

Déprotection semi-automatique de binaire

Yoann Guillot et Alexandre Gazet

Sogeti - ESEC

Résumé Que ce soit sur des binaires malicieux ou non, les protections dont le but est de freiner la rétro-ingénierie sont d'une complexité toujours croissante. Les analyser, et éventuellement les contourner, requiert des outils de plus en plus intelligents. Au travers de deux cas concrets, nous illustrerons d'abord le fonctionnement de quelques familles de protections particulièrement intéressantes, et tenterons de montrer leurs limites ainsi que les moyens de les supprimer pour revenir à un binaire proche de l'original. Toutes nos approches sont assistées par le framework de manipulation de binaire *Metasm*. Celui qui fond dans la bouche et pas dans la main.

1 Les machines virtuelles

Ces dernières années, l'augmentation des performances des processeurs a permis l'utilisation de techniques de protection toujours plus consommatrices de ressources, parmi lesquelles nous trouvons l'utilisation de machines virtuelles.

Dans le domaine de la protection logicielle, le terme *machine virtuelle* désigne une brique logicielle simulant le comportement d'un processeur. Nous pourrions parler de *processeur virtuel*. Ce dernier est pourvu de son propre jeu d'instructions et exécute un programme spécialement écrit dans le code machine lui correspondant. Dans ce papier, le terme machine virtuelle fera donc toujours référence à ce type de protection logicielle, et non à des méthodes ou logiciels plus évolués de virtualisation d'architectures existantes comme peuvent l'être *VMWare* ou *VirtualPC*.

Cette technique de protection est maintenant mise en œuvre dans bon nombre de protections commerciales largement répandues telles que *VMProtect*, *StarForce*, *Themida* ou encore *SecuROM*. Au delà de ces protections commerciales, nous avons aussi pu constater l'utilisation de machines virtuelles par certains *malwares*, tels que certains membres de la famille *Trojan.Win32.Krotten*.

En terme de protection logicielle, implémenter une machine virtuelle revient à ajouter un **niveau d'abstraction** entre le code machine tel qu'il est perçu lors de l'analyse (à l'aide d'un débogueur ou d'un désassembleur) et sa sémantique, c'est-à-dire la fonction qu'il remplit. Analyser ce niveau d'abstraction représente le plus souvent un défi de taille, relativement coûteux en temps. Le processeur simulé possède un jeu d'instructions et donc un code machine qui lui est propre. Dans la plupart des cas, l'analyste va devoir développer des outils pour être en mesure de les interpréter et donc de surmonter le niveau d'abstraction apporté par la protection.

Plusieurs éléments sont à prendre en compte pour évaluer la résistance à l'analyse d'une machine virtuelle. Une machine virtuelle peut être considérée selon deux modèles.

- Un **modèle concret** : c'est le code natif dans lequel est implémentée la machine virtuelle. Classiquement, nous trouvons ici toutes les primitives de manipulation de la mémoire, les registres virtuels, l'implémentation d'un cycle *fetch-decode-execute* et, bien entendu, les *handlers* d'instructions. La complexité de l'analyse du modèle concret peut être très élevée si le

code a été obfusqué. L'obfuscation est une autre technique de protection logicielle que nous étudierons par la suite.

- Un **modèle abstrait** : une machine virtuelle simule le comportement d'une architecture (ou processeur) donnée. Plus cette architecture abstraite est complexe et éloignée de l'architecture concrète, plus l'analyse est ralentie. D'une part le processus de traduction est plus lent et d'autre part, le manque de repères induit par la nouvelle architecture peut être source de confusion. Au niveau d'abstraction le plus haut, la difficulté est aussi induite par la complexité du programme exécuté sur le processeur virtuel.

1.1 Détection

La détection d'une machine virtuelle est liée à son implémentation. La plupart du temps, nous allons trouver des schémas très caractéristiques. L'implémentation d'un cycle *fetch-decode-execute* va se traduire par une boucle d'exécution. Un handler d'instruction n'est ni plus ni moins qu'une fonction prenant en entrée un certain nombre d'arguments, typiquement un ou deux, puis renvoie un résultat. Les machines virtuelles font souvent référence à ces *handlers* via un tableau de pointeurs de fonctions. D'un point de vue structurel, tracer le graphe d'appel d'un binaire peut être extrêmement révélateur.

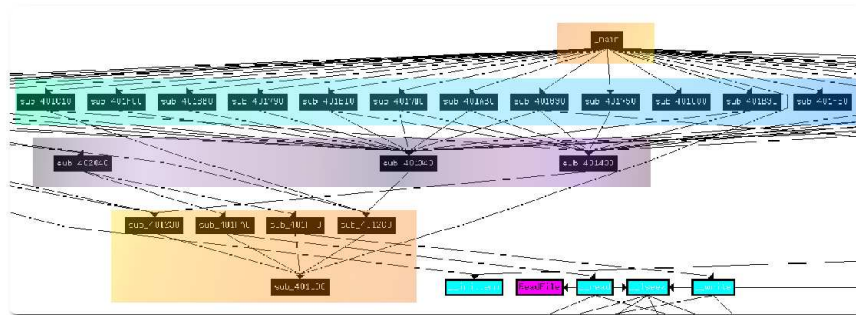


FIG. 1: Structure classique d'une machine virtuelle.

Nous retrouvons sur cet exemple (Fig. 1) tous les éléments que nous venons de citer :

- Au sommet en orange : la fonction *main*, c'est l'implémentation du cycle *fetch-decode-execute*. Cette boucle aiguille le flot d'exécution vers le handler responsable du traitement de l'instruction courante.
- Nous retrouvons les *handlers* d'instructions, tous situés au même niveau logique signalé en bleu.
- Ces derniers peuvent faire appel à des *fonctions-outils*, par exemple pour l'accès aux opérandes. Ces fonctions sont ici teintées en violet.
- Enfin au niveau hiérarchique le plus bas, de nouveau en orange, toutes les primitives de manipulation des éléments matériels virtuels : mémoire, registres, ports d'entrées/sorties, etc.

Ce cas de figure représente l'exemple idéal. En pratique la structure est souvent moins évidente à visualiser, et, dans ce cas, seules l'expérience et l'intuition joueront en faveur de l'analyste.

1.2 Analyse

L'analyse d'une machine virtuelle passe dans un premier temps par la compréhension de l'architecture abstraite. Une fois l'architecture conceptualisée, il est alors possible d'analyser les handlers d'instructions afin d'identifier le jeu d'instructions dont est pourvue la machine virtuelle. Cette analyse est, le plus souvent, fondée sur une approche comportementale, par exemple la perception d'un transfert de données entre un registre et une zone mémoire. La force d'une machine virtuelle réside dans son modèle abstrait et sa faiblesse dans son modèle concret. Le modèle concret contient traditionnellement toutes les traces qui permettent l'analyse : un contexte, un tableau de handlers d'instructions, les primitives de décodage des instructions et des opérandes.

La seconde étape est la traduction du code machine de la machine virtuelle, en un langage plus aisément intelligible dans lequel nous possédons un minimum de repères : typiquement un pseudo-assembleur *x86*. Cette étape de traduction est incontournable et précède systématiquement des phases de rétro-ingénierie éventuellement plus avancées telles que la décompilation.

1.3 Blindage et résistance à l'analyse

La complexification des techniques de protection à base de machines virtuelles passe par deux axes. Le premier, le plus évident, consiste à complexifier la machine virtuelle elle-même en utilisant par exemple une architecture simulée particulièrement *exotique*, un jeu d'instructions très étendu, ou encore un travail de déstructuration. Par travail de déstructuration, nous entendons tout processus susceptible de dissimuler, fractionner et, d'une manière générale, retarder l'analyse du modèle concret.

Le second axe de travail s'oriente vers la multiplication des machines virtuelles et donc du travail d'analyse. Une nouvelle fois, deux axes sont possibles et peuvent être combinés entre eux :

- Une multiplication *horizontale* : au sein d'un binaire, n parties sont protégées chacune par une machine virtuelle unique. C'est ainsi que fonctionne *VMPProtect* : l'insertion de marqueurs autour d'une portion de code se matérialise par la génération locale d'une machine virtuelle protégeant cette section. Si nous considérons d le facteur de dégradation des performances induit par une machine virtuelle, et si nous supposons le programme uniformément protégé (toutes les parties sont couvertes), alors quel que soit le nombre de machines virtuelles, le facteur de dégradation pour le programme entier est égal à d .
- Une multiplication *verticale* : il s'agit cette fois de concevoir des machines virtuelles, exécutant elles-mêmes des machines virtuelles. Si nous considérons n , la hauteur maximale de machines *empilées*, le facteur de dégradation (local cette fois) atteint alors un pic égal à d^n . Cela entraîne un effondrement dramatique des performances, même sur un processeur puissant avec des machines virtuelles simples.

Le but de ces techniques est de créer une asymétrie la plus grande possible entre le coût de la protection et le coût de l'analyse. Néanmoins, cette complexification, si elle paraît attirante, repose intégralement sur la capacité de l'auteur à générer des machines virtuelles uniques et originales. Entendons par là que l'analyse d'une instance doit révéler le moins possible d'information pour l'analyse de la suivante. Dans l'idéal, l'auteur obligera l'analyste à étudier chacune des instances séparément. En pratique, c'est l'analyse à la chaîne des machines virtuelles qu'il faudra prévenir au mieux. Des notions de poly/meta-morphisme reviennent alors sur le devant de la scène pour apporter un degré de complexité satisfaisant.

Cette évolution vers des protections par machines virtuelles toujours plus lourdes et massives fait clairement sentir le besoin d'automatisation de l'analyse et d'outils réalisant des abstractions puissantes sur le code.

2 L'obfuscation

Comme nous venons de le décrire précédemment, la force d'une technique de protection logicielle réside dans l'asymétrie qu'il existe entre le coût de la protection et le coût nécessaire pour contourner cette protection. L'obfuscation est une technique de protection consistant à augmenter la difficulté d'analyse en diminuant la lisibilité du code. Elle n'a cependant de réel intérêt que sur des portions stratégiques, qu'il est impossible d'ignorer ou contourner, mais ne doit pas non plus servir de marqueurs pour lesdites portions. De plus, une problématique évidente est celle de la définition de la résistance et de la capacité de nuisance d'une fonction d'obfuscation[1].

Un processus, ou fonction d'obfuscation peut se définir comme étant une **transformation** appliquée au code qui préserve sa **sémantique**.

2.1 La sémantique

La sémantique est l'interprétation que nous avons du code, son rôle, c'est-à-dire la fonction qu'il remplit. Prenons p une portion de code, p' cette même portion de code obfusquée et \mathbb{E} l'ensemble des états initiaux possibles, la préservation de la sémantique pourrait s'exprimer sommairement ainsi :

$$\forall x \in \mathbb{E}, p(x) = p'(x)$$

À un niveau local, c'est-à-dire au niveau des instructions, il est évident que les contextes seront différents au moins en partie. C'est là qu'il nous faut bien saisir la notion de sémantique. Prenons par exemple une portion de code dont la fonction est de calculer une addition. Une fois obfusquée, elle devra retourner le résultat correct quelles que soient les étapes intermédiaires par lesquelles elle passe. Pour retrouver des notions concrètes, certains registres, ou plus généralement des cases mémoire, seront essentiels, car ils contiennent le résultat ; d'autres non. Ceci définit ce que nous nommerons le **contexte significatif**.

La préservation de la sémantique sur le contexte significatif est un élément essentiel car elle seule garantit la bonne exécution du programme une fois la fonction d'obfuscation appliquée.

2.2 Les transformations

Bien que les processus d'obfuscation diffèrent suivant leurs objectifs et le niveau d'abstraction auquel ils sont appliqués, tous peuvent se modéliser sous la forme d'une transformation. Nous allons tout d'abord illustrer ce concept à l'aide de quelques exemples que nous avons relevés dans différents binaires.

Élément neutre. L'obfuscation à l'aide d'éléments neutres (Fig. 2) est relativement faible et nous allons expliquer pourquoi.

```

ror eax, 0dh ; @948c6d c1c80d
xchg eax, edx ; @948c70 92
ror edx, 13h ; @948c71 c1ca13
xchg eax, edx ; @948c74 92

rol eax, 0ah ; @948bd7 c1c00a
xchg eax, ebx ; @948bda 93
rol ebx, 16h ; @948bdb c1c316
wait ; @948bde 9b
xchg eax, ebx ; @948bdf 93

rol esi, 9 ; @948a94 c1c609
pushfd ; @948a97 9c
add eax, esi ; @948a98 01f0
wait ; @948a9a 9b
sub eax, esi ; @948a9b 29f0
lea esi, dword ptr ds:[esi] ; @948a9d 8d36
push eax ; @948a9f 50
pop eax ; @948aa0 58
popfd ; @948aa1 9d
rol esi, 17 ; @948aa2 c1c617

```

FIG. 2: Obfuscation par élément neutre

Nous nous trouvons en réalité dans le cas de figure le plus simple qu'il soit possible de rencontrer en terme d'obfuscation : les contextes significatifs avant et après exécution sont identiques. La conception de chaque pattern, pris de manière unitaire, est triviale. Tout effet induit est par la suite annulé, ainsi la sémantique est préservée. Dans le dernier des trois exemples ci-dessus, les *sub* annulent les *add*, les *pop* annulent *push*, les *rol 9* et *rol 17h* se complémentent arithmétiquement, etc. Cette propriété sera la base d'une méthode de détection automatique. Elle induit de plus une seconde propriété : une certaine symétrie visuelle dans les blocs d'obfuscation. Cette symétrie est très utile pour détecter ces patterns dans le cadre d'une analyse manuelle.

Maintenant que nous avons défini les patterns insérés comme des éléments neutres, nous savons qu'il nous est possible de réduire l'expression du code en masquant ces patterns. En pratique, nous

remplaçons le plus souvent ces séquences alambiquées par des *nop*, car finalement un *nop* n'est qu'une forme tout à fait remarquable de l'élément neutre : nous réalisons la substitution d'un élément neutre par un autre. Cette attaque contre la protection a l'avantage d'être très simple. Une fois la forme des patterns connus, une reconnaissance au niveau hexadécimal peut suffire, sans même avoir besoin d'interpréter le code. Plus une attaque est menée à un bas niveau d'abstraction, plus elle est simple à mener.

```

mov eax, var1
xor eax, 2c1a83c1h

mov ebp, var2
xor ebp, 0f91628c5h
xor ebp, 0d50cab04h

```

FIG. 3: Double XOR

Expansion de constantes. Ce terme désigne une famille de transformations dont le but est de complexifier l'expression d'une constante. Prenons l'exemple de la construction suivante (Fig. 3).

Les paramètres *var1* et *var2* sont *XORés* à l'aide d'une même clé de 32 bits. Elle apparaît clairement pour le premier paramètre, en revanche pour le second l'opération semble moins évidente jusqu'à ce que nous réalisons que : $0f91628c5h \oplus 0d50cab04h == 2c1a83c1h$. L'idée consiste à exprimer la clé sous la forme du résultat du *XOR* de deux constantes.

Les constantes utilisées dans certains algorithmes sont très significatives, elles peuvent identifier immédiatement une fonction comme étant un hashage MD5 par exemple. Il est alors intéressant de chercher à les dissimuler, et c'est précisément ce que fait l'expansion de constantes.

Le procédé employé dans l'exemple, à savoir le double XOR, est basique, mais d'autres systèmes plus ardues, tant sur la forme (manipulations sur les registres, sur la pile, en mémoire) que sur le fond (expression de la constante comme le résultat d'un polynôme ou encore d'une formule trigonométrique) sont envisageables.

Obfuscation structurelle. Un autre type de transformation fréquemment rencontré peut prendre la forme des portions de code représentées dans la Fig. 4.

Le premier exemple consiste à pousser une adresse sur la pile pour ensuite utiliser l'instruction *ret* comme un saut. Cette forme d'obfuscation est différente de la simple insertion de patterns car la structure du flot d'exécution du binaire est modifiée : nous pouvons parler d'obfuscation structurelle.

Le second exemple en particulier est une figure de style bien connue des processus d'obfuscation structurelle, à savoir celui des *faux* sauts conditionnels. Pour ces sauts, le calcul de la condition renvoie toujours vrai ou toujours faux. Une des deux branches n'est jamais utilisée, c'est la raison

```

push loc_403f84h          ; @403f07  68843f4000
ret                      ; @403f0c  c3

push 89h                 ; @21730  6889000000
add dword ptr [esp], 179h ; @21735  81042479010000
popf                     ; @2173c  669d
jnz loc_2173e            ; @2173d  75ff

```

FIG. 4: Obfuscation structurelle

pour laquelle nous parlons de *faux* saut conditionnel. Nous détaillerons plus en détail la nature de cette protection plus loin dans ce document. Dans cet exemple, à l'aide des constantes 89h et 179h suivi de *popf*, qui pour rappel recharge les flags du processeur à l'aide du *dword* situé au sommet de la pile, l'auteur contrôle la condition du saut situé en *@2173f*.

D'une manière générale, l'apparente condition du saut est utilisée afin de complexifier de manière artificielle le graphe de contrôle du programme. En outre, cette technique possède la particularité intéressante de perturber certains moteurs de désassemblage. Nous avons dit que l'une des deux branches était une branche morte, il est donc possible d'y insérer des constructions visant uniquement à polluer le listing produit par le désassembleur.

2.3 Complexité

La complexité résultant de l'application d'une fonction d'obfuscation est liée à la compréhension par l'analyste de la transformation utilisée. Plus elle est facilement identifiable, plus sa mise en échec sera simple et à un niveau d'abstraction bas. Tout élément fixe, donc prédictible, représente de l'information gratuite pour l'analyse. Ceci est encore plus vrai dans le cadre d'une protection logicielle. L'insertion de patterns statiques est donc conceptuellement faible, et bien qu'une grande variété de ces derniers remonte quelque peu le niveau de difficulté, ce n'est généralement pas suffisant pour garantir une protection efficace. Ce caractère statique n'est pas une faille en soit, mais il concourt à abaisser le niveau de sécurité général de la protection.

Du point de vue du développeur, la solution la plus efficace consiste à concevoir un jeu de transformations simples (f, g, \dots), qu'il maîtrise aisément et à les appliquer successivement, la sortie d'une fonction étant l'entrée de la suivante. Ces fonctions de base devront être suffisamment variées : insertion de patterns, utilisation de trappes, modification du graphe de contrôle, expansion de variables, mise en échec des moteurs de désassemblage, etc... Le développement de chaque fonction est ainsi plus aisé, et la préservation de la sémantique est plus facilement prouvable.

Suivant le principe de la composition, nous obtenons au final une fonction d'obfuscation résultante f_{res} telle que $f_{res} = f \circ g \circ \dots$; a priori bien plus résistante à l'analyse que chacune prise individuellement.

Pour renforcer encore la résistance, il sera de bon ton de faire varier la fonction finale d'obfuscation en *randomisant* par exemple l'ordre de composition ou le nombre de composées. Nous

pouvons aussi penser à des fonctions paramétrables. Les seules limitations sont alors l'imagination de l'auteur et la dégradation des performances qu'il pourra tolérer.

La contrainte technique, au grand dam des petits artisans de la rétro-ingénierie, est en passe de devenir secondaire tant les capacités des processeurs augmentent rapidement.

3 Metasm

*Metasm*¹ est un framework open-source permettant d'interagir avec du code machine sous ses différentes formes (hexadécimal, assembleur, C). Ce framework est intégralement écrit en Ruby, ce qui en fait le choix idéal pour la tâche qui nous incombe : il sera en effet facile de modifier les différentes actions effectuées, notamment lors du désassemblage de nos cibles. *Metasm* est également multi-plateformes et multi-OS, ce qui laisse présager de la possibilité de développer une classe pour interagir avec le processeur virtuel que nous pourrions avoir à étudier.

Ce framework a fait l'objet d'une présentation au SSTIC[2] en 2007 et à Hack.lu[3] la même année.

3.1 Désobfuscation de code

Une machine virtuelle utilisée comme méthode de protection logicielle inclut généralement des mécanismes d'obfuscation du code natif qui l'implémente. Afin de faciliter l'analyse préliminaire du code, il est important de pouvoir passer outre cette couche de la protection.

L'approche pour circonvenir l'obfuscation consiste à analyser quelques séquences de code à la main, d'en dégager les patterns d'obfuscation, puis d'opérer une substitution, soit en supprimant les instructions dans le cas de *junk*, soit en restaurant l'opération originale dans le cas d'une obfuscation comportementale.

Ceci peut être fait à différents moments :

- sur le binaire avant désassemblage,
- dynamiquement pendant le désassemblage,
- ou sur le code source assembleur une fois le désassemblage terminé.

La première option n'est applicable que si nous arrivons à dégager une signature binaire pour chaque pattern, et présente de plus le risque d'être appliquée au mauvais endroit (si par exemple ce pattern apparaît par hasard au sein d'une section de données).

La dernière solution est plus sûre, mais suppose que le désassemblage a pu être mené à terme sur le code obfusqué. Or, il est possible que ce code camoufle un saut ou un appel de fonction de telle sorte que le désassembleur passe à côté : dans ce cas il pourra nous manquer tout un pan de code.

Aussi nous préfererons la deuxième approche.

Désassemblage standard. De base, le moteur de désassemblage de *Metasm* fonctionne de la manière suivante :

1. désassemblage de l'instruction binaire au pointeur courant,

¹ <http://metasm.cr0.org/>

2. analyse des effets de l'instruction,
3. mise à jour du pointeur d'instruction.

L'analyse des effets de l'instruction consiste à déterminer si celle-ci réalise des accès mémoire, et/ou modifie le flot d'exécution. Si cette analyse indique qu'un des effets fait intervenir un registre, nous rentrons alors dans une phase de backtracking pour tenter de déterminer la valeur des registres incriminés.

Backtracking. Le backtracking consiste en l'émulation symbolique de chaque instruction en remontant tous les flots d'exécution ayant mené à l'adresse courante, jusqu'à détermination de la valeur de l'expression tracée. Les chemins suivis sont marqués, de sorte que si nous trouvons plus tard un nouveau flot d'exécution qui rejoint un flot déjà examiné, le backtracking reprenne en suivant cette nouvelle branche de code.

La méthode de backtracking accepte en entrée une expression arithmétique arbitraire, une ou plusieurs adresses de début d'exécution et une adresse de fin.

L'expression peut faire intervenir de manière symbolique la valeur des registres du processeur, valeur valable à l'adresse finale. La méthode renvoie alors la même expression, mais exprimée en fonction de la valeur symbolique des registres à l'adresse de début.

Par exemple, si nous cherchons la valeur du registre *eax* en emulant "*add eax, 4*", nous trouverons *eax+4* : la valeur de *eax* à la fin de ce bloc est égale à la valeur de *eax* au début du bloc, plus quatre.

Désassemblage modifié. L'approche que nous allons appliquer ici est assez peu intrusive, mais ne sera pas capable de gérer les obfuscations modifiant le flot d'exécution (sauf quelques cas très simples, utilisant des sauts de quelques octets).

Nous allons modifier la première étape de la boucle décrite précédemment, c'est-à-dire le désassemblage d'une instruction binaire. Le cœur de ce processus est la méthode *CPU#decode_instr_op*, sur laquelle nous allons intervenir. Lorsqu'une instruction est décodée, elle sera comparée au début d'une série de patterns définis en fonction des observations faites sur le binaire. Si le résultat est concluant, alors l'instruction suivante est désassemblée pour continuer la comparaison. Si un pattern est trouvé en intégralité, alors l'instruction correspondante à l'intégralité du pattern est retournée, en spécifiant que son encodage binaire correspond à la totalité du pattern. Ceci supprime effectivement la couche d'obfuscation.

Cette méthode a le bénéfice d'être récursive, et va donc résoudre les imbrications de patterns automatiquement ; ce qui réduit accessoirement la liste des patterns à spécifier.

Les instructions *nop* seront systématiquement intégrées à l'instruction qui les suit, de manière à éliminer complètement le *junk code* du listing final.

Les figures suivantes (Fig. 5) montrent le résultat de la désobfuscation d'une séquence de code : le junk est intégré à l'instruction suivante (*pop eax*), ce que l'on peut voir dans l'encodage binaire de celle-ci.

En pratique, nous allons commencer par rentrer les patterns les plus évidents, regarder le résultat obtenu, puis affiner notre liste jusqu'à obtenir un code suffisamment clair.

Une approche plus sophistiquée consisterait à analyser automatiquement toute séquence de code rencontrée, afin de déterminer l'effet de celle-ci, et voir si cet effet serait exprimable de manière plus

```

push 42h      ; @21d38h 6a42
ror ebp, 0dh  ; @21d3ah c1cd0d
xchg edx, ebp ; @21d3dh 87d5
ror edx, 13h  ; @21d3fh c1ca13
xchg edx, ebp ; @21d42h 87d5
pop eax      ; @21d44h 58
inc eax      ; @21d45h 40

```

FIG. 5: Code original

```

ror %1, X
xchg %1, %2
ror %2, 0x20-X
xchg %1, %2

```

FIG. 6: Le pattern de junk code

```

push 42h      ; @21d38h 6a42
pop eax      ; @21d3ah c1cd0d87d5c1ca1387d558
inc eax      ; @21d45h 40

```

FIG. 7: Après désobfuscation

simple en terme de nombre d'instructions. Cela s'applique particulièrement au junk, si celui-ci est totalement neutre du point de vue du processeur. En revanche si le junk se contente de ne préserver que le strict nécessaire au fonctionnement du programme, et se permet de modifier par exemple des registres qui ne sont pas utilisés par la suite, il faudra ajouter manuellement la notion de *contexte significatif*, introduite précédemment.

Cette approche est en fait assez similaire à la décompilation de code, que nous aborderons dans la suite.

3.2 Analyse automatique des handlers de machine virtuelle

Cette phase ne peut être réalisée qu'après une analyse manuelle préliminaire visant à déterminer l'architecture du processeur virtuel.

Cette architecture est caractérisée par :

- la forme générale des instructions,
- l'implémentation des registres (en mémoire ou dans un registre matériel),
- la gestion du flot de contrôle virtuel.

Cette analyse répondra aux question suivantes :

- Comment passe-t-on d'une instruction à la suivante ?
- Comment appelle-t-on une sous-fonction ?
- Comment retourne-t-elle à l'appelant ?

Il est alors envisageable d'analyser automatiquement les handlers, du moins ceux implémentant des fonctions simples, en comparant l'état du processeur virtuel avant et après l'exécution du handler. Nous pourrions ainsi identifier la fonction de transition qu'il implémente.

Pour cela, nous allons utiliser le moteur de backtracking de *Metasm* pour modéliser les transformations subies par tous les registres lors de l'exécution de ce handler. Nous reparcourrons ensuite la liste des instructions du handler pour tracer les accès mémoire réalisés.

Ces deux informations contiennent l'ensemble des actions du handler, que nous appellerons *binding*. Nous pouvons alors comparer ces transformations à un ensemble de formes connues, afin de déterminer le rôle du handler, par exemple "addition de deux registres dans un troisième". Cela nous permettra principalement de reconnaître automatiquement les fonction très basiques, notamment celles réalisant des opérations arithmétiques et des accès simples à la mémoire (charger un registre depuis la mémoire, indirections etc). Les handlers non reconnus de cette manière devront être analysés à la main, du moins si nous souhaitons afficher de manière synthétique le listing de l'assembleur virtuel.

Notons que cette analyse peut théoriquement se faire sans passer par une phase de désobfuscation complète des handlers, ce qui peut constituer un gain de temps appréciable.

3.3 Désassemblage du pseudo-code

Une fois chaque handler identifié, nous pouvons alors construire une nouvelle classe CPU représentant le processeur virtuel, et l'intégrer à *Metasm*. Nous obtenons ainsi rapidement un désassembleur avancé, capable de manipuler directement le pseudo-code binaire. Cette classe peut même être construite automatiquement à partir du résultat de l'analyse des handlers.

Les handlers de machine virtuelle sont souvent très basiques, rendant cette modélisation aisée.

Une fois en possession de cette classe représentant le processeur virtuel, si nous nous donnons la peine d'écrire les méthodes nécessaires au parsing d'instructions, nous serons même capable de compiler notre propre code exécutable sur le processeur virtuel.

3.4 Décompilation du pseudo-code

En général, le jeu d'instructions offert par la machine virtuelle est très réduit, et le programme principal est écrit par l'auteur à l'aide de macro-instructions, voire parfois d'un compilateur C rudimentaire.

```
mov reg1, addr_op1
load reg1, [reg1]
mov reg2, addr_op2
load reg2, [reg2]
add reg1, reg2
mov reg3, addr_result
stor [reg3], reg1
```

FIG. 8: Macro d'addition de deux cases mémoire

Dans ce cas le pseudo-code est facilement caractérisable, et rend possible la remontée automatique à ce niveau d'abstraction supérieur. Nous pouvons alors transformer le listing très bas niveau,

obtenu par désassemblage des pseudo-instructions, en un code source qui sera très proche de celui qu'a manipulé l'auteur de la protection.

Arrivé à ce stade, la protection appliquée au binaire est complètement circonvenue, et il ne reste qu'à lire le code pour analyser l'algorithme.

4 Application au T2

Nous nous proposons d'utiliser les capacités de *Metasm* sur le code du défi lié à la conférence T2 de l'année 2007. Ce défi a retenu notre attention comme support de nos travaux car il représente un cas concret très pertinent pour illustrer les notions que nous venons de développer.

4.1 Le challenge 2007

Le challenge est matérialisé par un simple exécutable Windows. Celui-ci nous demande de rentrer un mot de passe, puis nous indique si celui-ci est valide ou non. Le but est, bien entendu, de trouver un mot de passe qui soit accepté par le logiciel.

Un rapide désassemblage du code révèle immédiatement les actions réalisées par le programme :

1. extraction d'un fichier contenant un driver sur le disque, depuis les ressources du programme,
2. chargement du driver,
3. lecture du mot de passe de l'utilisateur,
4. transmission du mot de passe au driver au moyen d'un *IOCTL*,
5. récupération de la réponse du driver,
6. affichage du résultat.

L'algorithme qui nous intéresse se trouve donc dans le driver, dont l'analyse se montre beaucoup plus prometteuse. Celui-ci répond à l'*IOCTL* en passant par une fonction massivement obfusquée, qui met rapidement le désassembleur en échec.

Note : le format du listing généré par *Metasm* consiste en l'instruction suivie d'un commentaire contenant l'adresse de l'instruction (précédée d'un @), son encodage hexadécimal et les éventuels accès mémoire et/ou indirections du flot d'exécution. La notation *010203..j+37;* signifie que l'encodage de l'instruction commence par les octets *010203* et se poursuit sur 37 octets.

4.2 Désobfuscation

L'analyse manuelle de cette séquence montre que nous sommes en présence d'une obfuscation principalement de type junk code.

Concerto pour détraqués. Cette obfuscation est un joyeux bric-à-brac des multiples techniques d'obfuscation.

Voici un pattern des plus intéressants (Fig. 10).

Nous avons ici un petit récital d'obfuscation en quelques lignes :

```

// Xrefs: 1101ch
loc_215f8:
push esi                ; @215f8h 56
push ebx                ; @215f9h 53
lea esi, dword ptr [esi] ; @215fah 8d36
ror edi, 0dh           ; @215fch c1cf0d
xchg ebx, edi          ; @215ffh 87df
ror ebx, 13h           ; @21601h c1cb13
xchg ebx, edi          ; @21604h 87df
push ebx                ; @21606h 53
push ecx                ; @21607h 51
lea ecx, dword ptr [ebx+4] ; @21608h 8d4b04
xor ecx, edx           ; @2160bh 31d1
xchg ecx, dword ptr [esp] ; @2160dh 870c24
push edx                ; @21610h 52
mov edx, dword ptr [esp+4] ; @21611h 8b542404
rol edx, 0fh           ; @21615h c1c20f
mov dword ptr [esp+4], edx ; @21618h 89542404
pop edx                ; @2161ch 5a
pop dword ptr [esp+(-8)] ; @2161dh 8f4424f8
pop ebx                ; @21621h 5b
rol eax, 2             ; @21622h c1c002
rol eax, 1eh           ; @21625h c1c01e
pushfd                 ; @21628h 9c

```

FIG. 9: Le code brut du driver

```

pushfd                 ; @2164fh 9c
push edi               ; @21650h 57
call loc_21656         ; @21651h e8.. noreturn x:loc_21656
loc_21656:
add dword ptr [esp+0], 24h ; @21656h 81842400000002400000
pop edi                ; @21661h 5f
push 89h               ; @21662h 6889000000
add dword ptr [esp], 179h ; @21667h 81042479010000
popfd                  ; @2166eh 9d
jnz loc_21670          ; @2166fh 75ff x:loc_21670

// ----- overlap (1) -----
// Xrefs: 2166fh
loc_21670:
jmp edi                ; @21670h ffe7 x:loc_2167a

// ----- overlap (1) -----
out eax, 90h           ; @21671h e790
nop                    ; @21673h 90
mov eax, dword ptr [ebp+8] ; @21674h 8b4508
adc eax, edi           ; @21677h 11f8
int 3                  ; @21679h cc

// Xrefs: 21670h
loc_2167a:
pop edi                ; @2167ah 5f
popfd                  ; @2167bh 9d

```

FIG. 10: *Overlapping* et faux saut conditionnel.

- Tout d’abord nous remarquons l’obfuscation structurelle : le jeu autour du *call* (l. 3), suivi de la modification de l’adresse de retour sur la pile (l. 5).
- Ensuite nous avons un faux saut conditionnel : à l’aide des constantes 89h et 179h suivi de *popf* — qui pour rappel recharge les flags du processeur depuis la valeur au sommet de la pile — l’auteur s’assure que le *jnz* (l. 10) est pris.
- Une touche d’*overlapping*. Cette technique consiste à encoder deux instructions de telle sorte que les derniers octets de la première instruction soient réutilisés comme premiers octets de l’encodage d’une autre instruction. Elle trouve son origine dans le fait que la taille d’une instruction x86 est variable (de un à quinze octets) et que le processeur n’impose pas de contraintes d’alignement pour les adresses des instructions. Ainsi *jmp edi* (en binaire *ffe7*) est encodé en utilisant les octets des instructions *jnz loc_21670* (*75ff*) et *out eax, 90h* (*e790*). C’est le *jnz* qui réalise le saut sur l’instruction fantôme. Ce type d’obfuscation est impossible à implémenter par exemple sur une architecture ARM², où les instructions sont de taille fixe (16 ou 32 bits), et doivent être alignées.

Au final ce schéma est tout à fait charmant, mais présente plusieurs grosses faiblesses :

- les patterns d’obfuscation ne sont quasiment jamais imbriqués,
- et ils ne sont que très peu polymorphiques, uniquement au niveau des registres utilisés.

Ceci nous permettrait en fait de supprimer assez efficacement la majorité du junk code en utilisant simplement des patterns sur le binaire (par exemple pour les remplacer par des *nops*). Dans un souci de correction, nous utiliserons toutefois l’approche présentée précédemment, à savoir l’intégration au désassembleur.

Nous trouvons rapidement une vingtaine de patterns de junk code, dont une partie est constituée de séquences relativement triviales (ex : rotation de 32bits d’un registre). L’autre partie est constituée de séquences plus complexes, mais ces séquences sont toujours encadrées entre *pushfd ... popfd*³, et sont de ce fait assez facile à détecter : ces instructions ne se trouvent que très rarement dans du code “normal”.

Une fois cette première modification faite au désassembleur, nous obtenons un code déjà bien plus compact et agréable à lire.

Remettre les pendules à l’heure. Nous découvrons ensuite une autre construction intrigante.

Ce code (Fig. 11) lit la valeur du compteur du processeur⁴, et se sert de celle-ci comme variable pour un saut conditionnel. Or, cette valeur est très volatile, et difficilement prévisible, même pour les auteurs du programme ! D’une part, il faudrait être foncièrement optimiste pour jouer à pile ou face dans le code d’un driver et d’autre part, comme disait Albert, “*Dieu ne joue pas aux dés*”.

Prenons du recul pour visualiser la structure.

Un peu d’analyse manuelle nous révèle que les deux branches qui suivent le saut conditionnel dépendant du *rdtsc* sont parfaitement équivalentes : les deux séquences obfusquées *B* et *B'* ont la

² Advanced RISC Machine

³ Les instructions *x86 pushfd* et *popfd* sauvent et restaurent respectivement l’état des flags du processeur, ce qui permet d’insérer ces patterns entre l’évaluation d’une condition et le saut conditionnel associé.

⁴ L’instruction *RD TSC* signifie *Read TimeStamp Counter*.

```

// Xrefs: 1101ch
loc_215f8:
push esi           ; @215f8h 56
push ebx          ; @215f9h 53
pushfd            ; @215fah 8d36c1cf0d87dfc1cb13..<+37>
rdtsc             ; @21629h 9c5031c0668cc83d0900..<+15>
imul ecx, ebx     ; @21642h 0fafcb
cmp cl, 7fh       ; @21645h 80f97f
jnb loc_21aba     ; @21648h 0f836c040000 x:loc_21aba

popfd             ; @2164eh 9d
pop ebx          ; @2164fh 9c57e800000000818424..<+48>
pop esi          ; @21689h c1c7099c01fa9b29fa8d..<+8>
[...]

loc_21abah:
popfd             ; @21abah 9d
pop ebx          ; @21abbh 57870c245f87cf5b
pop esi          ; @21ac3h c1ce0d87cec1c91387ce..<+9>
    
```

FIG. 11: Le code désobfusqué

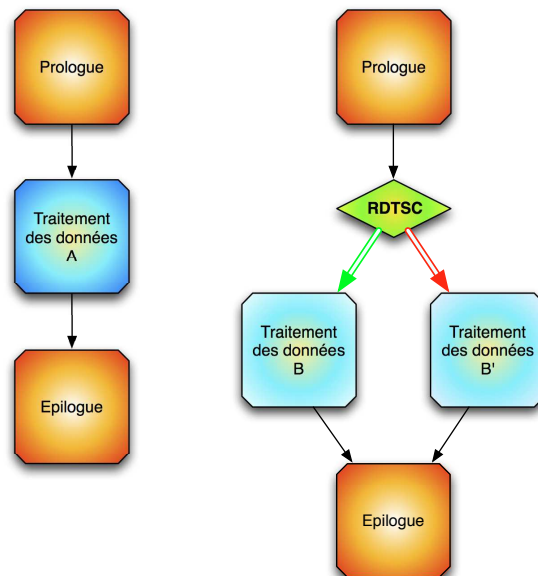


FIG. 12: Obfuscation structurelle.

même sémantique A (Fig. 12).

Cette forme d’obfuscation est différente de la simple insertion de patterns car la structure du flot d’exécution du binaire est modifiée : c’est une obfuscation structurelle. L’utilisation de *rdtsc* est astucieuse : deux exécutions du binaire ne produiront pas la même trace d’exécution, le comportement étant à priori non reproductible. Ainsi un point d’arrêt posé sans précautions lors d’une première revue du code, pourra ne pas être activé lors d’une seconde exécution. D’un autre côté, *rdtsc* fait partie des instructions que nous ne trouverons jamais naturellement dans du code compilé, et elle ne manquera pas d’attirer l’attention d’un attaquant.

Notons que la forme des deux branches diffère : les deux chemins B et B' sont obfusqués aléatoirement par les patterns que nous avons déjà rencontrés, de manière indépendante ; ils ne sont donc pas semblables bit à bit.

Nous avons choisi de considérer cette séquence également comme pattern de junk code, et nous suivrons toujours la branche correspondant à la non-prise du saut. L’autre branche est tout de même désassemblée, au cas où nous souhaiterions nous assurer de la conformité des deux chemins, mais elle ne sera pas affichée dans les exemples que nous donnerons ici.

Le saigneur des anneaux. Nous tombons alors sur une séquence originale.

```

pushfd          ; @131c0h  9c
push eax       ; @131c1h  50
xor eax, eax   ; @131c2h  31c0
mov ax, cs     ; @131c4h  668cc8
cmp eax, 9     ; @131c7h  3d09000000
jle loc_131d5h ; @131cch  7e07  x:loc_131d5h
rdtsc         ; @131ceh  0f31
imul eax, ecx  ; @131d0h  0fafc1
jmp eax        ; @131d3h  ffe0  x:unknown

// Xrefs: 131cch
loc_131d5h:
pop eax        ; @131d5h  58
popfd         ; @131d6h  9d

```

FIG. 13: Test d’exécution en ring 0

Ce code (Fig. 13) vérifie s’il est en train de s’exécuter dans le contexte du noyau Windows (ring 0), ou dans le contexte d’un processus normal. Dans le noyau, *cs* vaudra 8, alors que dans un processus utilisateur il vaudra *0x1b*. L’unique but de cette séquence, que nous retrouverons un peu partout dans le code du driver, est d’interdire l’exécution dans un contexte de processus standard.

Une des contraintes assez gênante du challenge, pour quelqu’un qui voudrait l’analyser de manière dynamique, est que le code intéressant est intégralement exécuté en mode noyau ; or cet environnement est assez peu propice au débogage. En effet, peu de débogueurs sont adaptés aux interactions avec du code ring 0, et la plupart des outils standards sont orientés vers l’analyse de code utilisateur (ring 3). Il serait donc tentant d’exécuter ou d’émuler le code du driver dans un

processus normal, afin de pouvoir étudier son comportement. Lors de l'exécution de cette séquence de code, la valeur du sélecteur de segment *cs* va alors conduire à ne pas prendre le saut conditionnel (l. 6 dans l'exemple). Dans ce cas, les trois instructions exécutées ensuite construisent une adresse pseudo-aléatoire et y déroutent le flot d'exécution (par le saut l. 9), ce qui garantit un plantage immédiat du programme : dans le meilleur des cas l'adresse en question sera invalide pour l'exécution de code, au pire elle contiendra du code qui sera exécuté en dehors de son contexte prévu, et plantera tôt ou tard.

```
loc_173a7h:
  call loc_173ach          ; @173a7h  e8..  noreturn x:loc_173ach
loc_173ach:
  pop  edx                ; @173ach  89ed9c873424569b9d5e5a
  cmp  edx, 7fffffffh     ; @173b7h  c1c10a5156e800000000..<+18>
  jnb  loc_17436h         ; @173d3h  0f835d000000  x:loc_17436h

  rdtsc                  ; @173d9h  558d6d0811c5872c2450..<+17>
  add  edx, eax           ; @173f4h  01c2
  jmp  edx                ; @173f6h  c1..  x:unknown
// [45 data bytes]

// Xrefs: 173d3h
loc_17436h:
  mov  edx, dword ptr [ebp+0] ; @17436h  8b5500
```

FIG. 14: Autre test d'exécution en ring 0

Une autre séquence (Fig. 14) a le même effet.

Celle-ci teste l'adresse du code au lieu de tester la valeur du segment *cs* ; si cette adresse est inférieure à *80000000h* (cas du code utilisateur⁵) un saut aléatoire est effectué de la même façon.

Ces deux séquences rejoignent la liste des patterns de junk code, à masquer par le désassembleur.

When you're pushed. . . À ce point de l'analyse, le code visualisé est bien réduit, mais nous trouvons encore de longues et pénibles séquences. Nous ne sommes toutefois plus en présence de simple junk code, mais de séquences obfusquées ayant un effet non nul : de l'obfuscation comportementale.

En général, ces séquences vont pousser et manipuler la valeur d'un registre sur la pile au lieu de la modifier directement (Fig. 15).

Nous trouvons également des références à de la mémoire, toujours de la forme $[ebx+ioffset\delta]$, qui sont très intrigantes ; les offsets ne correspondent à rien et semblent être choisis au hasard (Fig. 16 l. 5, 13, 15). Un examen plus poussé révèle qu'une grande zone de mémoire est utilisée pour stocker des valeurs temporaires, uniquement pour obfusquer le code : en effet, les deux branches suivant un *rdtsc*, donc ayant la même sémantique, n'utilisent pas les mêmes offsets et n'y stockent pas forcément les mêmes valeurs. Nous pouvons donc ignorer les écritures qui y sont faites.

Nous pouvons encore élaguer une grande partie du code source, et obtenons un résultat extrêmement concis.

⁵ Certes, en démarrant Windows avec le switch /3G...

```

push esi                ; @150eah 56
lea esi, dword ptr [ebp+8] ; @150ebh 8d7508
adc esi, ecx            ; @150eeh 11ce
xchg esi, dword ptr [esp] ; @150f0h 873424
push ecx               ; @150f3h 51
mov ecx, dword ptr [esp+4] ; @150f4h 8b4c2404
shl ecx, cl           ; @150f8h d3e1
mov dword ptr [esp+4], ecx ; @150fah 894c2404
pop ecx               ; @150feh 59
pop dword ptr [esp+(-0ch)] ; @150ffh 8f4424f4

push dword ptr [ebx+0e92h] ; @176beh ffb3920e0000
push ecx              ; @176c4h 51
mov cl, 11h          ; @176c5h b111
rol dword ptr [esp+4], cl ; @176c7h d3442404
pop ecx              ; @176cbh 59
pop dword ptr [ebx+0e92h] ; @176cch 8f83920e0000

```

FIG. 15: Obfuscation comportementale

```

loc_175e6h
mov eax, dword ptr [ebp+0ch] ; @175e6h 9c600f3101c8c1c20a5
xor eax, 1749c891h          ; @1767eh c1ce0d87cec1c91387c
push dword ptr [ebx+eax]    ; @1769bh ff3403
pop dword ptr [ebx+0e92h]   ; @1769eh 50e8000000009c81842
[...]

rdtsc_17882h:
mov ecx, dword ptr [ebp+0ch] ; @1788eh c1c00a5053e80000000
xor ecx, 1749c891h          ; @178a7h 6089f0d3c2619c81f19
push ecx                    ; @178c6h 9c55e80000000081842
mov ecx, dword ptr [ebx+ecx] ; @178f4h 9c5031c0668cc83d090
mov dword ptr [ebx+25b8h], ecx ; @1790eh 568d730431de8734245
pop ecx                     ; @1792eh c1c102c1c11e59
push dword ptr [ebx+25b8h]   ; @17935h 9c5031c0668cc83d090
pop ecx                     ; @17952h 59

```

FIG. 16: Junk en mémoire

4.3 La machine virtuelle

Nous pouvons commencer l'analyse du code clarifié dans la partie précédente.

La première opération réalisée est l'allocation d'une grande zone mémoire (106000 octets), dont l'adresse est stockée dans le registre *ebx*.

Le code exécuté ensuite est une séquence de blocs ayant une structure très semblable. Chacun commence par la séquence d'obfuscation vue précédemment, avec le *rdtsc* suivi de deux branches équivalentes.

Des valeurs de type *dword* sont manipulées. Elles sont récupérées à partir de l'adresse stockée dans *ebp*, et xorées avec une constante dépendante du bloc.

Ensuite elles servent parfois d'index dans la table allouée au début (basée à *ebx*).

Enfin, un épilogue constant va mettre à jour le registre *ebp* à partir de la valeur à *[ebp+4]*, et exécuter le bloc dont l'adresse est stockée à *[ebp]* (après un *xor* utilisant une seconde constante, elle aussi dépendant du bloc).

Nous allons interpréter ce schéma d'exécution comme la séquence des instructions d'un processeur virtuel : les blocs seront les handlers, et les données pointées par *ebp* seront les instructions virtuelles.

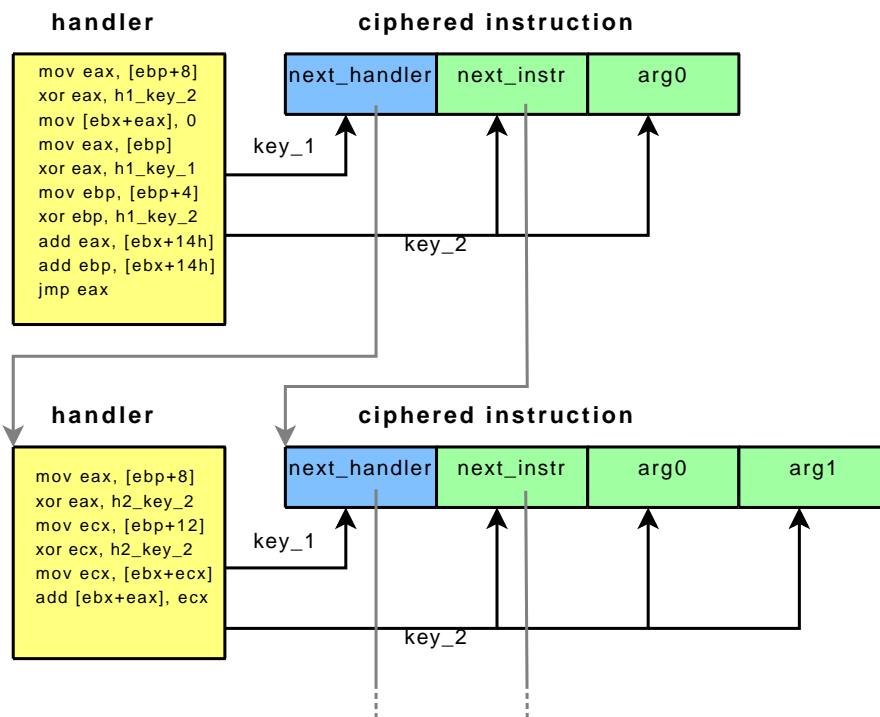


FIG. 17: Architecture du T2

In the belly of the beast. Le registre *ebx* pointe en permanence sur le début de la zone allouée lors de l'initialisation, qui constitue le contexte d'exécution du processeur. Le registre *ebp* pointe lui sur l'instruction virtuelle en cours d'exécution.

Les instructions de la machine virtuelle sont une séquence de 2 à 6 mots mémoire contigus. Chacun de ces mots est chiffré par un *xor* au moyen d'une clé 32bits, qui change pour chaque handler. Le premier mot est chiffré au moyen d'une seconde clé spécifique, elle aussi variable d'un handler à l'autre (Fig. 17).

Ce premier mot contient l'offset du handler à utiliser pour interpréter l'instruction suivante ; l'offset doit être ajouté à l'adresse de base de la section *.data* du driver pour obtenir l'adresse réelle en mémoire. Le second mot contient l'offset de la prochaine instruction, toujours par rapport à la base de la section *.data*. Enfin, les mots suivants sont interprétés de façon différente selon le handler. La plupart du temps ce sera un entier (une valeur *immédiate*) ou un index par rapport à *ebx* (un registre virtuel).

Les registres virtuels sont quelques mots mémoire situés à un offset fixe dans la zone allouée lors de l'initialisation de la machine virtuelle (celle pointée par *ebx*). La méthode d'accès par les handlers permettrait de référencer l'intégralité de la mémoire de l'hôte, mais en pratique nous ne trouvons qu'une poignée d'offsets utilisés.

Chacun des registres possède un rôle bien précis ⁶ :

Offset	Nom	Rôle
0x4	<i>esp</i>	pointeur de pile
0x8	<i>ebp</i>	pointeur de frame
0x64	<i>r64</i>	registre générique
0x68	<i>r68</i>	registre générique
0x78	<i>r78</i>	indirection mémoire
0x0	<i>esp_init</i>	valeur initiale du pointeur de pile
0xc	<i>host_esp</i>	pointeur de pile de l'hôte
0x18	<i>retval</i>	valeur de retour d'une sous-fonction

Initialisation. Revenons sur la séquence d'initialisation de la machine virtuelle.

Les premières instructions calculent l'adresse du début de la section *.data*, et la stockent dans *esi* (l. 1-5).

Un appel est fait pour allouer 0x19e10 (106000) octets, et l'adresse de ce buffer (le contexte) est stockée dans *ebx* (l. 7-9).

Enfin plusieurs champs du contexte sont initialisés :

- L'adresse de base de la section *.data* est stockée à $[ebx+14h]$ (l. 11).
- L'adresse du sommet de la pile est stockée dans *host_esp* ($[ebx+0ch]$) (l. 12).
- La pile virtuelle est initialisée à $ebx+101d0h$ ($ebx+66000$), et cette adresse est placée dans les registres virtuels *esp* et *esp_init* (l. 13-15).

Exemple de handler. Pour illustrer le fonctionnement des instructions virtuelles, regardons le début d'un handler additionnant deux registres.

⁶ Les registres dans la seconde partie du tableau sont utilisés de manière très sporadique.

```

loc_215f8h:
  call loc_216bch          ; @215f8h  56538d36c1cf0d87dfc
loc_216bch:
  pop esi                 ; @216bch  5e
  sub esi, 0e6bch         ; @216bdh  81eebce60000

  push 19e10h             ; @216c3h  68109e0100
  call dword ptr [esi]    ; @216c8h  ff16 r4:xref_13000
  mov ebx, eax            ; @216cah  89e489c3

  mov dword ptr [ebx+14h], esi ; @216ceh  c1c602c1c61e5089f08
  [...]
  mov dword ptr [ebx+0ch], esp ; @218f7h  89ff9c871c24539b9d5
  lea eax, dword ptr [ebx+101d0h] ; @21904h  8d83d0010100
  mov dword ptr [ebx], eax ; @2190ah  558704245d955089c09
  mov dword ptr [ebx+4], eax ; @21969h  56873c245e87fe89430

```

FIG. 18: Initialisation de la VM

```

loc_15336:
  mov ecx, dword ptr [ebp+0ch] ; @15336h  9c600f3101f131cbc1c1..<+102>
  xor ecx, 842b1208h          ; @153a6h  528d550811ca87142451..<+49>
  mov ecx, dword ptr [ebx+ecx] ; @153e1h  516089d0d3c2618b0c0b..<+30>
  mov eax, dword ptr [ebp+8]   ; @15409h  51e800000009c818424..<+19>
  xor eax, 842b1208h          ; @15426h  9cc1c10a5155e8010000..<+37>
  add dword ptr [ebx+eax], ecx ; @15455h  c1c60a5651e800000000..<+15>

```

FIG. 19: Addition de deux registres

1. Les deux premières instructions récupèrent et déchiffrent l'index du registre source à partir du champ *+0ch* de l'instruction virtuelle (2^{ème} argument).
2. Cet index est ensuite utilisé pour lire la valeur du registre virtuel correspondant.
3. Les deux instructions suivantes récupèrent et déchiffrent l'index du registre destination (champ *+8* de l'instruction, i.e. 1^{er} argument).
4. Enfin l'addition est réalisée et stockée dans le contexte.

Ensuite le contrôle est passé au couple handler/instruction suivant, grâce à un code que nous retrouvons dans tous les handlers.

```

mov ecx, dword ptr [ebp+0] ; @1546eh 8b4d00
xor ecx, 149f0c63h ; @15471h c1ce0d87cec1c91387ce..<+12>
mov ebp, dword ptr [ebp+4] ; @15487h c1c20a92c1c0169b928b6d04
xor ebp, 842b1208h ; @15493h 6089f0d3c26181f55204..<+12>
add ebp, dword ptr [ebx+14h] ; @154a9h 036b14
add ecx, dword ptr [ebx+14h] ; @154ach c1c11287d1c1c20e9087..<+4>
jmp ecx ; @154bah 89ff9c871424529b9d5affe1

```

FIG. 20: Transition entre handlers

Les offsets du handler (l. 1-2) et de l'instruction (l. 3-4) suivante sont déchiffrés à partir de l'instruction courante (notez l'utilisation de la clé spécifique pour l'offset du handler), puis convertis en adresses absolues par addition de l'adresse de base de la section *.data*, stockée à *[ebx+14h]* (l. 5-6). Enfin le contrôle est passé au handler suivant, *ebp* pointant bien sur l'instruction virtuelle à interpréter.

4.4 Modélisation

Le principal problème de cette architecture est que nous ne possédons ni la liste des handlers, ni la liste des instructions : il nous faut suivre le flot d'exécution afin de retrouver les clés de chiffrement de chaque handler, et déchiffrer l'instruction pour pouvoir trouver l'adresse du couple handler/instruction suivant, et ainsi de suite.

Cette opération est extrêmement fastidieuse à faire à la main, c'est pourquoi nous allons la réaliser de manière automatique.

À la trace. Pour cela, nous utilisons le moteur de backtracking de *Metasm*, qui nous permet de trouver les valeurs de *eip* et de *ebp* à la fin du handler en fonction de leur valeur initiale.

Nous pouvons ainsi retrouver les deux clés de chaque handler : la clé pour l'offset du handler suivant est le résultat de $(backtrace(eip) - [ebx+14]) \oplus [ebp]$, et la clé pour les arguments est $(backtrace(ebp) - [ebx+14]) \oplus [ebp+4]$.

Ces deux clés en main, il nous est possible de suivre le flot d'exécution de la machine virtuelle.

```

loc_19aa0h:
mov ecx, dword ptr [ebp+10h] ; @19aa0h 8b4d10
xor ecx, 2ce6fc22h          ; @19aa3h 9c81f122fce62cc1c60a..<+20>
cmp dword ptr [ebx+ecx], 0  ; @19ac1h c1ce0d87cec1c91387ce..<+7>
jz loc_19b57h               ; @19ad2h 0f847f000000 x:loc_19b57h

mov ecx, dword ptr [ebp+0]  ; @19ad8h 538d5d0811c3871c2450..<+18>
xor ecx, 3c606446h         ; @19af4h 8d12565e9c81f1466460..<+8>
mov ebp, dword ptr [ebp+4] ; @19b06h 9c5031c0668cc83d0900..<+16>
xor ebp, 2ce6fc22h         ; @19b20h 538d5b0431eb871c2455..<+30>
jmp loc_19b92h              ; @19b48h 89c09c873c24579b9d5f..<+5> x:

// Xrefs: 19ad2h
loc_19b57h:
mov ecx, dword ptr [ebp+8] ; @19b57h 8b4d08
xor ecx, 2ce6fc22h         ; @19b5ah 9c6089d0d3c26181f122..<+10>
mov ebp, dword ptr [ebp+0ch] ; @19b6eh c1c80d92c1ca13928b6d0c
xor ebp, 2ce6fc22h         ; @19b79h c1c11287d1c1c20e9087..<+15>

// Xrefs: 19b48h
loc_19b92h:
add ebp, dword ptr [ebx+14h] ; @19b92h 036b14
add ecx, dword ptr [ebx+14h] ; @19b95h 034b14
jmp ecx                       ; @19b98h 518d4b109c19d99d870c..<+21>

```

FIG. 21: Saut conditionnel

Cette technique fonctionne bien jusqu'au 18^{ème} handler où survient une erreur : les deux clés ont chacune deux valeurs possibles.

L'analyse manuelle de ce handler (Fig. 21) révèle qu'il s'agit d'un saut conditionnel : si un des registres virtuels contient une valeur non nulle, le contrôle est passé normalement au handler suivant, par contre si cette valeur est nulle, l'adresse virtuelle à exécuter ensuite est prise dans les champs 2 et 3 de l'instruction.

Ces deux champs étant chiffrés avec la clé *key_args* du handler.

Deux choix s'offrent alors à nous :

- refaire un algorithme de suivi de chemin,
- réutiliser le moteur de désassemblage de *Metasm*.

En survolant les premiers handlers que nous venons de détecter, la simplicité de ceux-ci nous incite à choisir la deuxième solution.

Le décapsuleur automatique. L'élément fondamental pour cette approche est une méthode nous permettant d'analyser un handler quelconque.

Cette méthode est composée des étapes suivantes :

1. Désassemblage du handler, en utilisant un désassembleur natif sous-jacent.
2. Examen de la forme générale du handler obtenu :
 - De combien de blocs basiques est-il constitué ?
 - Comment ces blocs sont agencés ?
 - Combien y a-t-il de points de sortie au handler ?

3. Analyse des effets du handler :

- Quelles sont les modifications apportées aux registres du processeur natif ?
- Quelles sont les modifications apportées à la mémoire ?

Le traçage des modifications des registres réels par le handler est fait directement à partir de la fonctionnalité de backtrack de *Metasm* : pour chaque point de sortie, nous listons les modifications apportées à chaque registre par rapport au début du handler.

L'analyse des effets sur la mémoire est moins directe : il faut reparcourir le handler, instruction par instruction, et backtracer chaque accès lorsque l'on trouve une instruction qui écrit en mémoire.

Nous obtenons ainsi un tableau qui liste l'ensemble des éléments modifiés par le handler, et y associe la valeur par laquelle ils seraient remplacés lors de son exécution. Nous l'appellerons le *binding* du handler.

L'architecture de la machine virtuelle nous permet de définir plusieurs raccourcis simplifiant radicalement l'expression de ce binding.

Si le handler conserve la valeur du registre *ebx* (qui, pour rappel, contient l'adresse de base du contexte de la machine virtuelle), et si le binding de *ebp* et *eip* correspond à la séquence de transition entre handlers que nous avons vue précédemment, alors nous récupérerons les deux clés de déchiffrement du handler.

La clé de déchiffrement des arguments est alors utilisée pour définir les entités symboliques suivantes :

- *arg0* qui correspond au premier argument de l'instruction, considéré comme un entier ($[ebp+8] \oplus arg_key$),
- *reg0* qui correspond au premier argument utilisé comme index de registre virtuel ($[ebx+arg0]$),
- *reg0b* qui correspond au premier argument utilisé comme index de registre virtuel (*byte ptr* $[ebx+arg0]$), dont on n'utilise que les 8 bits de poids faible (l'équivalent d'*al* pour *eax* sous *x86*),
- on réitère l'opération pour les arguments restants : *arg1*, *reg1* etc.

Ces informations permettent d'identifier le handler par comparaison avec une série de modèles que nous aurons définis. Si aucun modèle ne correspond, le handler est marqué comme inconnu ; il faudra alors déterminer un nouveau modèle couvrant son cas, après analyse manuelle.

Les modèles triviaux correspondent aux handlers constitués d'un seul bloc : nous sommes alors sûrs d'avoir la sémantique complète du handler dans le binding.

Ces handlers correspondent aux les opérations arithmétiques standard, et aux opérations de lecture/écriture en mémoire (*indirections*).

D'autres handlers font un appel à une fonction native. Ces appels sont toujours réalisés par l'intermédiaire d'un tableau de pointeurs, initialisé au chargement du driver.

Les fonction référencées dans ce tableau sont :

- *ExAllocatePool* : allocation mémoire,
- *ExFreePoolWithTag* : libération mémoire,
- une zone du driver remplie de zéros (jamais appelée),
- une fonction affichant une chaîne de debug, utilisant *vsprintf* et *DbgPrint* (jamais appelée),


```

handler_13491h:
// handler type: add reg, reg
// "reg0" <- Expression["reg0", :+, "reg1"]
mov eax, dword ptr [ebp+0ch] ; @13491h 8b450c
xor eax, 8d3f5d8bh ; @13494h 9c358b5d3f8d9d
mov eax, dword ptr [ebx+eax] ; @1349bh 6089e8d3c26150c1c502..<+79>
mov ecx, dword ptr [ebp+8] ; @134f4h 89ed9c873424569b9d5e..<+3>
xor ecx, 8d3f5d8bh ; @13501h c1c10a5156e800000000..<+29>
add dword ptr [ebx+ecx], eax ; @13528h 6089c0d3c26101040b
mov eax, dword ptr [ebp+0] ; @13531h 558734245d87f58b4500
xor eax, 6f9078cch ; @1353bh c1c30a5350e801000000..<+69>
mov ebp, dword ptr [ebp+4] ; @1358ah c1c3099c01de9b29de8d..<+10>
xor ebp, 8d3f5d8bh ; @1359eh 9c52e800000000818424..<+57>
add ebp, dword ptr [ebx+14h] ; @135e1h 558d6b109c19c59d872c..<+22>
add eax, dword ptr [ebx+14h] ; @13601h c1e620034314
jmp eax ; @13607h ffe0

```

FIG. 22: Résultat de l'analyse automatique d'un handler : addition

- *MmGetSystemRoutineAddress* : récupération de l'adresse d'une fonction du système d'exploitation à partir de son nom,
- une fonction du driver réalisant un hashage MD5.

Nous connaissons la sémantique de chacune de ces fonctions, et donc la sémantique complète du handler.

Le cas particulier de *MmGetSystemRoutineAddress* pourrait être problématique. Mais il s'avère à l'usage que les handlers y faisant référence récupèrent tous l'adresse de la fonction native *KdDebuggerEnabled*, et s'en servent pour planter le processeur s'ils détectent la présence d'un débogueur.

La séquence responsable du plantage (Fig. 23, l. 20), a été réduite par le désobfusicateur.

Il s'agissait initialement d'un saut aléatoire sur le résultat d'un *rdtsc*.

Le modèle vérifie que la fonction dont nous récupérons l'adresse correspond bien à l'adresse de *MmGetSystemRoutineAddress*. Si tel est le cas, ce handler est marqué comme *trap*, sinon il est traité comme non reconnu.

Parmi les handlers restant, quatre sont plus complexes à analyser, car ils font intervenir des sauts conditionnels.

- Un saut conditionnel virtuel, qui saute à une adresse virtuelle ou une autre en fonction de la nullité d'un registre virtuel.
- Trois catégories de handlers qui définissent la valeur d'un registre virtuel à 0 ou 1 selon que sa valeur initiale est respectivement supérieure, inférieure ou égale à celle d'un autre registre virtuel.

Dans ces cas, nous utilisons une petite heuristique, afin de garder un code d'analyse concis : on va tout simplement regarder l'instruction native utilisée pour le saut conditionnel que l'on retrouve dans l'implémentation du handler.

Enfin, pour les deux derniers types de handler, l'un est un saut indirect inconditionnel qui charge l'offset du prochain handler et de la prochaine instruction à partir de la valeur de deux registres virtuels.

```

handler_13fb6h:
// handler type: trap
// "call_arg0" <- Expression[81929]
// "call" <- Expression[Indirection[[Indirection[[:ebx, :+, 20]..
jmp loc_1401bh          ; @13fb6h  9c6..  x:loc_1401bh
db "KdDebug"          ; @14009h
db "gerEnabled", 0    ; @14010h
loc_1401bh:
call loc_14020h        ; @1401bh  e80..  noreturn x:loc_14020h
loc_14020h:
pop eax                ; @14020h  58
sub eax, 17h           ; @14021h  2d17000000
push eax               ; @14026h  50
mov eax, dword ptr [ebx+14h] ; @14027h  508b4314508b44240458..<+4>
call dword ptr [eax+18h] ; @14035h  ff5018
cmp byte ptr [eax], 1  ; @14038h  803801  r1:unknown
jnz loc_14043h        ; @1403bh  7506  x:loc_14043h

loc_1403dh:
jmp loc_1403dh        ; @1403dh  0f3101c8ffe0  x:loc_1403dh

loc_14043h:
mov eax, dword ptr [ebp+0] ; @14043h  8b4500
xor eax, 45f341a7h       ; @14046h  351465ef9335b3241cd6
mov ebp, dword ptr [ebp+4] ; @14050h  8b6d04
xor ebp, 0b7048be8h      ; @14053h  81f5d4401e619b81f53c..<+3>
add ebp, dword ptr [ebx+14h] ; @14060h  036b14
add eax, dword ptr [ebx+14h] ; @14063h  034314
jmp eax                 ; @14066h  ffe0

```

FIG. 23: Test de présence d'un débogueur kernel

Le dernier est plus complexe que les autres : il fait intervenir une boucle, et semble implémenter une sorte de routine de chiffrement (Fig. 24).

```

handler_23c8fh:
// handler type: decryptcopy reg, imm, imm, imm
mov edi, dword ptr [ebp+8] ; @23c8fh 50578d00578d1b50585f..<+138>
xor edi, 2c1a83c1h ; @23d23h 9c81f7c1831a2c9d
push dword ptr [ebx+edi] ; @23d2bh ff343b
pop edi ; @23d2eh 5f
mov esi, dword ptr [ebp+0ch] ; @23d2fh 8b750c
xor esi, 2c1a83c1h ; @23d32h 9c81f6c1831a2c9d
add esi, dword ptr [ebx+14h] ; @23d3ah 037314
mov eax, dword ptr [ebp+10h] ; @23d3dh 8b4510
xor eax, 2c1a83c1h ; @23d40h 9c35c1831a2c9d
mov ecx, dword ptr [ebp+14h] ; @23d47h 8b4d14
xor ecx, 2c1a83c1h ; @23d4ah 9c81f1c1831a2c9d

loc_23d52h:
mov edx, dword ptr [(esi+(4*ecx))+(-4)] ; @23d52h
xor edx, eax ; @23d61h 31c2
mov dword ptr [(edi+(4*ecx))+(-4)], edx ; @23d63h
rol eax, cl ; @23d67h 505188c9d34424045958
add eax, ecx ; @23d71h 01c8
loop loc_23d7ah ; @23d73h e205 x:loc_23d7ah
jmp loc_23d7fh ; @23d75h e905000000 x:loc_23d7fh

loc_23d7ah:
jmp loc_23d84h ; @23d7ah e905000000 x:loc_23d84h

loc_23d7fh:
jmp loc_23d89h ; @23d7fh e905000000 x:loc_23d89h

loc_23d84h:
jmp loc_23d52h ; @23d84h e9c9ffffff x:loc_23d52h

loc_23d89h:
mov eax, dword ptr [ebp+0] ; @23d89h 8b4500
xor eax, 0a9c47c96h ; @23d8ch 351d38ce7c358b440ad5
mov ebp, dword ptr [ebp+4] ; @23d96h 8b6d04
xor ebp, 2c1a83c1h ; @23d99h 81f5c52816f99b81f504..<+3>
add ebp, dword ptr [ebx+14h] ; @23da6h 036b14
add eax, dword ptr [ebx+14h] ; @23da9h 034314
jmp eax ; @23dach ffe0

```

FIG. 24: Handler de déchiffrement

Ce handler accepte en fait 4 arguments : un registre contenant l'adresse d'un buffer destination, un entier qui est un offset dans la section *.data*, un autre entier représentant une taille (en *dwords*), et un dernier utilisé comme clé de déchiffrement.

Il recopie alors les données depuis la section *data* vers le buffer destination, après les avoir xorées avec la clé. La clé est modifiée à chaque round, par une rotation et une addition faisant intervenir l'index du prochain *dword* à décoder (l.19 et 20).

À l'assaut ! Forts de ces informations, nous pouvons à présent associer à chaque handler :

- un nom d’opcode virtuel, pour pouvoir afficher le listing des instructions de la VM,
- une liste d’arguments symboliques, pour décoder et interpréter les arguments de chaque instruction virtuelle,
- un binding traduisant les effets de l’instruction sur le contexte du processeur virtuel,
- et les deux clés de chiffrement (quand elles existent), pour décoder les arguments et suivre le flot d’exécution.

Ces données peuvent être calculées une fois pour toutes : nous les sauvegardons dans un fichier de cache afin d’accélérer le script, car la partie prenant le plus de temps actuellement est le désassemblage et la désobfuscation du code natif des handlers.

À titre d’information, initialiser le cache prend 15 minutes sur une machine standard pour tous les handlers (au nombre de 112), alors que le traitement intégral décrit dans ce papier, avec le cache déjà peuplé, prend moins de 30 secondes (y compris les phases que nous décrirons par la suite).

L’idée est alors de construire dynamiquement une classe ruby utilisant cette méthode d’analyse automatique pour interpréter les handlers d’instructions virtuelles à la volée.

Cette classe servira de *CPU* pour le moteur de désassemblage standard de *Metasm*, afin de pouvoir l’utiliser de manière transparente sur le code virtuel.

Le fonctionnement sera le suivant.

Nous commençons par définir un espace de code virtuel, où l’adresse d’une instruction est le couple (*adresse du handler*, *adresse de l’instruction*). Un tel objet, représentant la première instruction virtuelle, est passé à *Metasm* comme point d’entrée d’un programme, dont le *CPU* est une instance de ladite classe, *T2CPU*.

Ce cpu contient une référence à une instance de *Disassembler* standard, la même que nous avons utilisé pour générer les listings utilisés en exemples dans cet article.

Comme nous l’avons vu dans la partie introduisant *Metasm*, le désassembleur va demander au cpu de décoder et d’analyser l’instruction à l’adresse courante, mettre à jour cette adresse, et recommencer. C’est là que commence la partie intéressante.

Lorsqu’il va recevoir la requête de décodage d’instruction, notre processeur virtuel utilise l’analyse automatique pour déterminer l’instruction à renvoyer, ainsi que ses effets ; notamment l’adresse de l’instruction suivante.

Ainsi, de manière totalement transparente, *Metasm* va désassembler chacune des instructions virtuelles comme un programme classique, en nous fournissant notamment les fonctionnalités de backtracking des registres virtuels.

Le listing obtenu suite à cette étape est fort remarquable (Fig. 25).

Nous remarquons que l’assembleur est de très bas niveau, prenant par exemple plusieurs instructions pour faire l’équivalent d’un *push*. Les instructions semblent également ne manipuler que des variables sur la pile.

Chronologiquement, c’est à ce moment que nous avons pu assigner un nom et un rôle à chacun des registres virtuels.

En regardant le fichier de cache des handlers, nous nous apercevons en outre que la plupart sont dupliqués : il y a par exemple quatre handlers capables de réaliser une addition entre deux registres virtuels, sans aucune différence sémantique.

```

entrypoint_219feh_21ea6h:
  nop ; @219feh_21ea6h
  mov r68, 28h ; @138fah_35748h
  add r68, host_esp ; @1501dh_2adc7h
  mov r64, dword ptr [r68] ; @175e6h_38670h r4:dword_host_esp+28h
  mov dword ptr [esp], r64 ; @156d1h_368a2h w4:dword_ctx+101d0h
  mov r64, 4 ; @13184h_34e02h
  add esp, r64 ; @15e23h_34d28h
  mov r68, 2ch ; @138fah_37a96h
  add r68, host_esp ; @15336h_3a431h
  mov r64, dword ptr [r68] ; @18231h_31642h r4:dword_host_esp+2ch
  mov dword ptr [esp], r64 ; @16f1bh_36aa2h w4:dword_ctx+101d4h
  mov r64, 4 ; @13626h_2fca1h
  add esp, r64 ; @13491h_3594eh
  trap ; @18c00h_35b6eh
  mov ebp, esp ; @22d86h_34262h
  mov r64, 234h ; @13184h_1c968h
  trap ; @1bf14h_36f9ah
  add esp, r64 ; @1501dh_2b162h
  trap ; @15a3ch_2a44fh
  mov r78, 200h ; @138fah_2f305h
  trap ; @14121h_3a473h
  add r78, ebp ; @1501dh_3402ah
  mov r64, dword ptr [r78] ; @175e6h_35d98h r4:dword_ctx+103d8h
  xor r64, 1 ; @17f53h_37befh
  jrz loc_2d630h_2d8ffh, r64 ; @19aa0h_1c6ffh x:loc_2d630h_2d8ffh
  mov r68, 0ch ; @13184h_394ebh
  syscall_alloc_ptr r64, r68 ; @25d49h_35b2fh
  mov r78, 200h ; @138fah_2926ch
  add r78, ebp ; @15e23h_32249h
  mov dword ptr [r78], r64 ; @15fc3h_3119ch w4:dword_ctx+103d8h
loc_2d630h_2d8ffh:
  decryptcopy r64, 100h, 751734b1h, 3 ; @2d630h_2d8ffh w4:unknown
  mov r78, 0 ; @13184h_1aab2h
  add r78, ebp ; @15336h_2d1abh
  mov dword ptr [r78], r64 ; @156d1h_2a854h w4:dword_ctx+101d8h
  mov r78, 0 ; @13626h_3a2ebh
  add r78, ebp ; @15336h_33a37h
  mov r64, dword ptr [r78] ; @175e6h_2e159h r4:dword_ctx+101d8h
  [...]

```

FIG. 25: Le code de la machine virtuelle

4.5 Macro assembleur

En filtrant les *nops* et autres *trap*, nous nous rendons assez vite compte que l'assembleur virtuel que nous parcourons est le résultat d'une programmation par macro-instructions. Nous retrouvons en effet en permanence des séquences d'instructions identiques, à quelques constantes près ; séquences qui sont contiguës et découpent parfaitement le texte en blocs élémentaires.

Le travail nécessaire pour reconstituer les macro-instructions à partir de ce listing est très similaire à ce que nous avons déjà fait pour la désobfuscation du code du driver : concaténer plusieurs instructions successives en une autre instruction exprimant la même sémantique de manière plus concise.

Ici les patterns sont très simples : il s'agit principalement de construire une adresse dans un registre puis de faire une indirection ; en pratique ces opérations ne font intervenir que les instructions *mov* et *add*.

Comme les instructions n'ont pas de définition précise, nous sommes libres d'utiliser des constructions peu courantes, par exemple en faisant plusieurs références à la mémoire dans une même instruction, ou en utilisant des références imbriquées ; chose classiquement interdite dans les langages de processeurs réels.

Nous disposons également de toutes les informations nécessaires au déchiffrement des portions de la section *.data* par les instructions *decryptcopy*, nous pouvons donc profiter de cette passe supplémentaire pour expliciter ces chaînes.

Le listing gagne ainsi encore un niveau de concision et de lisibilité (Fig. 26).

```

entrypoint_219feh_21ea6h:
mov dword ptr [esp], dword ptr [host_esp+28h] ; @219feh_21ea6h r4:d
add esp, 4 ; @13184h_34e02h
mov dword ptr [esp], dword ptr [host_esp+2ch] ; @138fah_37a96h r4:d
add esp, 4 ; @13626h_2fca1h
mov ebp, esp ; @18c00h_35b6eh
add esp, 234h ; @13184h_1c968h
mov r64, dword ptr [ebp+200h] ; @15a3ch_2a44fh
xor r64, 1 ; @17f53h_37befh
jrz loc_2d630h_2d8ffh, r64 ; @19aa0h_1c6ffh x:loc_2d630h_2d8f
syscall_alloc_ptr r64, 0ch ; @13184h_394ebh
mov dword ptr [ebp+200h], r64 ; @138fah_2926ch
loc_2d630h_2d8ffh:
decryptcopy r64, "\000\000\000\000\000\000\000\000\000\000\376\324\004"
mov dword ptr [ebp], r64 ; @13184h_1aab2h

```

FIG. 26: Le même listing en macro-instructions

Appels de fonction. Nous voyons alors apparaître un pattern intéressant : l'instruction de saut indirect est systématiquement utilisée pour exécuter une instruction dont l'adresse a été poussée sur la pile un peu plus tôt, ce qui n'est pas sans rappeler la structure d'un appel de fonction classique.

Celui-ci est bizarrement conçu : tout d'abord le code pousse sur la pile manuellement l'adresse de retour, sauvegarde le pointeur de frame, puis pousse les arguments ; le flot d'exécution continue ensuite naturellement dans la sous-fonction.

En général, les arguments sont poussés avant l'adresse de retour, et le flot est détourné pour entrer dans la fonction.

Le retour est également particulier : le code commence par supprimer l'espace réservé sur la pile pour les variables locales et les arguments, puis il vérifie si le pointeur de pile est à sa valeur initiale (comme lors du démarrage de la VM) en le comparant avec la valeur de *esp_init*. Si c'est le cas, le code sort de la VM et rend le contrôle au code non-obfusqué du driver, qui va renvoyer la réponse à l'utilisateur ; dans le cas contraire le pointeur de frame et l'adresse de retour sont dépilés et le contrôle est rendu à l'appelant (Fig. 27).

```

mov r68, 4 ; @138fah_353c6h
mov dword ptr [esp], 2507dh ; @13626h_2fa7eh
add esp, r68 ; @15e23h_1e097h
mov dword ptr [esp], 14cch ; @13184h_3302ch
add esp, r68 ; @1501dh_29749h
mov dword ptr [esp], ebp ; @16a01h_22b17h
add esp, r68 ; @15e23h_32514h
mov dword ptr [esp], dword ptr [ebp] ; @13184h_2be36h
add esp, r68 ; @13491h_351c1h
mov dword ptr [esp], dword ptr [ebp+4] ; @13bf0h_38c9eh
add esp, r68 ; @15336h_39e68h
mov ebp, esp ; @1d735h_386abh
add esp, 204h ; @13626h_323b7h

[corps de la fonction]

add esp, -20ch ; @13626h_32000h
mov r64, esp ; @22174h_34eeah
seteql r64, esp_init ; @1fc6eh_3629ah
xor r64, 1 ; @19357h_21febh
jrz loc_272dah_35a76h, r64 ; @1ab4fh_32340h
add esp, -4 ; @13184h_369a0h
mov ebp, dword ptr [esp] ; @187beh_251a3h
add esp, r64 ; @15e23h_378dch
mov r68, dword ptr [esp] ; @175e6h_39ee2h
add esp, r64 ; @1bf14h_1c1edh
mov r6c, dword ptr [esp] ; @175e6h_39419h
jmp r68+13000h_r6c+13000h ; @23fb0h_381e2h

loc_272dah_35a76h:
syscall_free_exitvm ; @272dah_35a76h

```

FIG. 27: Macros d'appel et de retour d'une sous-fonction

Le fait d'intégrer ces macros permet à *Metasm* de gérer correctement les sous-fonctions ; nous couvrons dès lors une séquence de code virtuel beaucoup plus longue.

Nous retrouvons ainsi une organisation standard du code : quelques petites fonctions utilitaires, par exemple pour calculer la taille d'une chaîne de caractères, et trois fonctions plus conséquentes.

En affichant le code complet du driver ainsi désassemblé, après avoir marqué les octets occupés par les instructions virtuelles, il nous est possible vérifier que nous avons bien interprété l'intégralité du binaire ; à l'exception de quelques très courtes séquences d'octets aléatoires entre les handlers.

Cette opération est donc un succès.

```

syscall_alloc_ptr r64, 24h          ; @138fah_2f2f0h
mov var20c, r64                    ; @13184h_3924eh
decryptcopy r64, "powered by T2 - http://www.t2.fi\000?\365\256"
mov var4, r64                      ; @13bf0h_354a1h
call sub_13626h_323b7h, var0, var4 ;@138fah_353c6h x:sub_13626h_323b7h
mov var0, var21c                   ; @144cch_3807dh
call sub_138fah_33176h, var0       ; @138fah_23c63h x:sub_138fah_33176h
mov var0, retval                   ; @168f4h_38615h
mov var4, 10h                     ; @13184h_399eeh
mov r64, var0                     ; @13184h_376f2h
seteql r64, var4                   ; @1486dh_22be1h
xor r64, 1                         ; @17f53h_181f5h
mov var0, r64                     ; @13bf0h_37a11h
jrz loc_13184h_3276fh, var0       ; @14121h_2c78dh x:loc_13184h_3276fh
add esp, -23ch                    ; @138fah_1fbei1h
syscall_free var200                ; @138fah_311e6h

```

FIG. 28: Code virtuel avec macro-instructions et variables locales

Nous pouvons enfin commencer à étudier le code pour se donner une idée de l'algorithme implémenté.

Le code en question ressemble énormément à du C compilé sans optimisation (voire désoptimisé...). Toutes les opérations sont faites sur la pile, dans l'espace réservé aux variables locales.

Le code est également très redondant : beaucoup de valeurs sont recopiées inutilement dans plusieurs variables temporaires sur la pile, avant d'atteindre leur destination finale.

Nous renommons les variables suivant leur offset dans la *stack frame* : ainsi *dword ptr [ebp+128h]* devient *var128*, toujours dans un souci de lisibilité (Fig. 28).

4.6 Décompilation

À cause de la multiplication des variables temporaires, le code est toujours fastidieux à lire (d'accord, un peu moins qu'au début).

L'idée est maintenant de s'abstraire du code lui-même, pour essayer d'en exprimer la sémantique globale, et non plus simplement au niveau des instructions individuelles. Ceci semble d'autant plus réalisable que chacune de ces instructions est très simple, sans effet de bord aucun, et que le nombre d'instructions est assez réduit.

Il se trouve par ailleurs que *Metasm* inclut un compilateur C, et dispose donc de tous les objets nécessaires à la manipulation de code dans ce langage. Nous allons essayer de générer une retranscription du listing en C.

Dans l'idéal, le but est de retranscrire le programme fonctionnellement, en ignorant au maximum les détails d'implémentation ; nous devons donc commencer par définir les actions qui sont significatives. Nous allons complètement ignorer les modifications des registres, et des variables sur la pile.

Seuls vont nous intéresser :

- les appels de fonctions, aussi bien internes qu'externes, avec leurs arguments,
- les écritures en mémoire (hors pile),
- et les prédicats associés aux sauts conditionnels.

Cette approche très simplifiée est inapplicable à un langage concret, car les données sur la pile sont significatives : nous y trouverons des buffers, des structures (au sens C). Nous pouvons aussi imaginer les problématiques liées aux interactions avec le système d'exploitation, notamment les threads, ou les signaux. . .

Sans parler de la gestion d'exceptions et autres joyeusetés.

En pratique, nous allons nous contenter de travailler bloc d'instruction par bloc d'instruction, ce qui est beaucoup plus simple qu'une approche globale du programme, et donne des résultats déjà très intéressants.

Nous allons utiliser une approche récursive, fonction par fonction, au fur et à mesure du parcours du code à partir du point d'entrée : si au cours de l'analyse nous tombons sur un appel de sous-fonction, cette sous-fonction sera analysée à son tour.

Cela suppose que le segment de code principal est une fonction, ce qui est généralement le cas.

Le principal objectif est de réduire toutes les affectations intermédiaires aux variables de la pile. Pour cela, il nous faut déterminer lesquelles sont significatives.

Nous allons faire une première passe sur le code, en marquant pour chaque bloc quelles sont les variables lues et quelles sont les variables écrites. Ensuite, en utilisant le graphe de contrôle de la fonction, nous pouvons déterminer quelles sont les affectations à conserver : ce sont celles qui sont suivies par une lecture dans un autre bloc sans avoir été réécrites entre temps.

Canif. Nous transformons alors chaque bloc basique en son équivalent en langage C (Fig. 29) : les opérations arithmétiques sont regroupées de manière à traduire le binding du bloc complet par rapport à une variable significative donnée en une seule expression C ; les sauts sont transformés en *goto* et les sauts conditionnels en *if (..) goto label;*

Nous ne nous intéressons pas au type des variables pour l'instant, que l'on considérera comme des entiers ; mais nous conserverons le type des indirections : pour les références à un seul octet, le type sera *char*, pour les autres *int*.

Sequel. L'étape suivante, et probablement la plus importante, est la reconnaissance des structures de contrôle C standard : *if*, *if/else* et *while*.

Commençons par le plus simple : le *if*.

En parcourant le flot d'exécution, si nous rencontrons un saut conditionnel (c'est-à-dire un *if (..) goto label;*) dont le label destination se situe plus loin dans la fonction, nous allons le transformer

```

void sub_13626h_323b7h(int arg4, int arg0)
{
    int var0;
    int var200;
sub_13626h_323b7h:
    var200 = 0;
    var0 = var200;
label_13184h_31e59h:
    if (*((char*)(var0 + arg0)) == 0)
        goto loc_13bf0h_1aa3bh;
    *((char*)(arg4 + var200)) = *((char*)(var200 + arg0));
    var200 += 1;
    var0 = var200;
    goto label_13184h_31e59h;
loc_13bf0h_1aa3bh:
    *((int*)(arg4 + var200)) = 0;
    return;
}

```

FIG. 29: Phase préliminaire de la décompilation

en inversant la condition du *if* et en remplaçant le *goto* par tout le code entre le *if* et le label. Il faudra ensuite réappliquer le même traitement à ce qui constitue maintenant le bloc *then*, afin de gérer les tests imbriqués.

On voit assez rapidement comment gérer les *if/else* : si le bloc *then* nouvellement trouvé se termine par un *goto label* dont la destination est dans le code qu'il nous reste à examiner, on peut supprimer le *goto* et repositionner la séquence entre la fin du *if* et le label dans le bloc *else*.

Le code obtenu est plus clair, mais un pattern nous apparaît comme un *if/else* (Fig. 30) : un *then* contient un label et se termine par un *goto* ; et le code suivant le *if* saute sur ce label. Un test est ajouté pour gérer ce cas et le traiter correctement (Fig. 31).

Le code sous-jacent est très répétitif ; ce qui se traduit entre autre par des *if/else* dont les dernières expressions sont identiques entre le *then* et le *else*. Nous profitons de cette phase pour factoriser ce code, et le sortir du *if*.

Une fois tout le code traité de cette façon, la gestion du *while* est très simple : c'est tout simplement un label suivi d'un *if* dont la dernière instruction est un *goto* sautant sur ce label. Quelques tests supplémentaires permettent également de reconnaître les *continue* et *break* associés.

Enfin une dernière passe cosmétique est effectuée, de manière à effacer les labels inutilisés dans le code.

Le résultat est très satisfaisant (Fig. 32).

Project II. La dernière étape est la détermination du type des variables.

Un premier parcours du code nous permet de noter quelles sont les variables qui se voient affecter des entiers immédiats : celles-la seront de type *int*.

```
if (cond) {
    a;
label:
    b;
    goto anywhere;
}
c;
goto label;
```

FIG. 30: Pattern de if/else

```
if (cond) {
    a;
} else {
    c;
}
label:
b;
goto anywhere;
```

FIG. 31: Pattern de if/else résolu

```
void sub_13626h_323b7h(int arg4, int arg0)
{
    int var0;
    int var200;
    var200 = 0;
    var0 = var200;
    while (*((char*)(var0 + arg0)) != 0) {
        *((char*)(arg4 + var200)) = *((char*)(var200 + arg0));
        var200 += 1;
        var0 = var200;
    }
    *((int*)(arg4 + var200)) = 0;
    return;
}
```

FIG. 32: Phase intermédiaire de la décompilation

Ensuite nous recherchons les indirections avec *type casting*, qui, couplées à la liste des entiers, nous permettent de déterminer les variables qui sont de type pointeur. La taille des données déréférencées nous permet en outre de connaître le type de données pointées.

Il ne reste plus alors qu'à effectuer une passe finale, pour rectifier les séquences de *cast* ; il faut également penser à rectifier les additions/soustractions de pointeurs.

Le résultat final est au-dessus de nos espérances (Fig. 33).

```
void sub_13626h_323b7h(char *arg4, char *arg0)
{
    int var0;
    int var200;
    var200 = 0;
    var0 = var200;
    while (arg0[var0] != 0) {
        arg4[var200] = arg0[var200];
        var200 += 1;
        var0 = var200;
    }
    *((int*)(arg4 + var200)) = 0;
}
```

FIG. 33: Phase finale de la décompilation

On obtient de cette manière un listing complet de l'algorithme obfusqué, qui n'occupe que 352 lignes, que l'on peut réduire facilement à environ 200 lignes à la main.

C'est le moment d'avoir une petite pensée émue pour les psychopathes qui ont résolu ce programme de manière artisanale...

Pour rappel, le code natif de cet algorithme comporte environ 40000 instructions obfusquées, qui implémentent 112 handlers utilisés par 3000 instructions virtuelles, le tout conçu pour être pénible à lire.

4.7 It's a trap !

L'examen final du code C montre que même à ce niveau, les concepteurs du challenge pensent encore à nous.

Nous voyons par exemple quelques séquences étranges (le code est légèrement retraité à la main) (Fig. 34).

Decryptcopy est utilisé pour désobfusquer deux chaînes depuis le binaire original (on les voit ici en clair), pointées par *var20c* et *var210*. *var210* est visiblement une chaîne chiffrée, qui est déchiffrée dans un buffer passé en argument.

La clé de déchiffrement, par contre, est calculée à la ligne 9 de manière fort étrange : la chaîne dans *var20c* est appelée comme s'il s'agissait du corps d'une fonction.

```

void sub_13626h_38b14h(char *arg4, int arg0)
{
    var20c = malloc(20);
    decryptcopy(var20c, "Q\213D$\b\213L$\f\323\350Y"
        "\302\b\000\000\000\006\271\004");
    var210 = malloc(20);
    decryptcopy(var210, "X\244{\022\322\246\023\\|"
        "\350\350\201\210\000\000\000\000_\335\271");
    var214 = 0;
    while (var214 < 13) {
        r64 = ((int*)(int, int))var20c)(arg0, var214);
        arg4[var214] = var210[var214] ^ r64;
        var214 += 1;
    }
    *((int*)(arg4 + 13)) = 0;
    [...]
}

```

FIG. 34: Shellcode natif

```

entrypoint_0:
// function binding: eax -> (dword ptr [esp+4]>>dword ptr [esp+8]), esp -> esp+0ch
// function ends at 0ch
push ecx                ; @0 51
mov eax, dword ptr [esp+8] ; @1 8b442408
mov ecx, dword ptr [esp+0ch] ; @5 8b4c240c
shr eax, cl             ; @9 d3e8
pop ecx                 ; @bh 59
ret 8                   ; @0ch c20800 endsub entrypoint_0
db 0, 0, 6, 0b9h, 4    ; @0fh

```

FIG. 35: Le shellcode pointé par *var20c*

Il s'agit en fait d'un shellcode natif très simple (Fig. 35), qui retourne son premier argument décalé vers la droite d'un nombre de bits spécifiés dans son deuxième argument.

En l'occurrence, le premier argument est un des paramètres de la fonction que nous étudions, et le deuxième est le compteur indiquant l'index dans la chaîne en cours de décodage.

L'utilisation de code natif de manière si détournée laisse entendre que de sales coups sont à l'étude.

```

*((int*)(arg4 + 13)) = 0;
var208 = malloc(8);
decryptcopy(var208, "1\300f\214\310\303\000");
r64 = ((int*)(void))var208();
*((int*)(arg4 + 14)) = r64 + -8;
free(var20c);
free(var210);
free(var208);
}

```

FIG. 36: La roche...

Cela se vérifie précisément dans l'épilogue de cette même fonction (Fig. 36).

Une fois encore un shellcode est utilisé dans l'algorithme, ici la valeur renvoyée va servir à remplir une partie de la chaîne déchiffrée par la fonction.

```

entrypoint_0:
// function binding: eax -> cs, esp -> esp+4
// function ends at 5
xor eax, eax ; @0 31c0
mov ax, cs ; @2 668cc8
ret ; @5 c3 endsub entrypoint_0
db 0, 30h ; @6

```

FIG. 37: ... et l'anguille

Le shellcode est toujours très simple, mais le but ici semble beaucoup moins bon enfant : il s'agit à nouveau d'un test d'exécution en ring 0 (Fig. 37), comme nous en avons trouvé dans la partie désobfuscation.

Cette fois-ci le test est beaucoup plus pervers : si le code ne s'exécute pas en ring 0 (où *cs* vaut 8), l'action entreprise n'est pas un plantage franc comme nous en avons l'habitude. Non, ici on va simplement modifier subtilement le résultat d'une fonction, probablement cruciale dans l'algorithme.

Ce genre d'interventions est toujours très pénible à détecter, et encore plus à contourner, car les effets ne sont visibles que beaucoup plus tard dans l'exécution du programme.

```
int sub_13bf0h_2b788h(int arg0)
{
    register int r64;
    register int retval;
    int *var200;
    int var204;

    r64 = malloc(16);
    md5(r64, arg0, 8);
    var200 = r64;
    var204 = var200[0] ^ var200[1] ^ var200[2] ^ var200[3];
    if (r64 > 0)
        var204 += 1;

    free(var200);
    retval = var204;
    return retval;
}
```

FIG. 38: Autre test ring 0

Une autre fourberie du même accabit est présente dans une autre fonction du programme (Fig. 38).

Cette fonction prend en argument une chaîne, la passe au hashage *MD5* et renvoie une valeur dérivée de ce résultat. Or une toute petite variation, apparemment insignifiante, vient là aussi nous mettre des bâtons dans les roues : il s'agit des deux lignes 12 et 13 du listing. Celles-ci testent l'adresse renvoyée par le malloc, et si celle-ci est positive, le résultat de la fonction est légèrement biaisé.

Il s'agit encore une fois du même type de tests que ceux présents dans l'obfuscation, mais les effets sont bien plus pernicieux.

4.8 So long, and thanks for all the fish

Ces derniers écueils franchis, nous pouvons à présent reconstituer l'algorithme du programme qu'il nous faut défaire.

1. La longueur du mot de passe soumis doit être égale à 16 caractères.
2. 3 entiers ($h1, h2, h3$) sont déduits de celui-ci. Il s'agit en fait d'un encodage en base64, utilisant une base personnalisée.
3. Ces 3 entiers doivent remplir certaines conditions : on doit obtenir 'T2' en xorant le mot de poids faible et le mot de poids fort d'un d'eux, et le troisième doit être égal au condensat du MD5 des deux premiers, xorés par lui-même ($sub_13bf0h_2b788h(md5(h1 \oplus h3, h2 \oplus h3)) == h3$).
4. Le mot de passe ne doit pas contenir les caractères $+$ et $/$.

5. Une fois ces conditions vérifiées, le premier entier est utilisé comme clé pour déchiffrer une phrase stockée en dur.
6. La valeur du hash MD5 de cette phrase est testée comme vérification finale.

En étudiant comment est déchiffrée la phrase finale, nous voyons que seulement 20 bits de la clé sont utilisés. Cette valeur est largement bruteforceable : un petit programme en assembleur teste toutes les clés en une fraction de seconde.

On ne trouve qu'un seul entier pour lequel la phrase déchiffrée passe le teste du MD5; cette phrase est "t207@owned.by".

On connaît ainsi 20 bits du premier entier.

Le test par rapport à la valeur 'T2' nous donne également 16 bits pour le second entier; et la relation avec les xor et le md5 nous permet de déduire le 3^{ème} des deux premiers.

Le test réalisé par cet algorithme est donc très lâche, et autorise un grand nombre de solutions possibles (2^{28} en ignorant le test sur les caractères invalides).

Épitaphe. Au final, le challenge était d'un très bon niveau, et extrêmement intéressant.

Le résoudre de manière purement statique fut un défi passionnant; qui a accessoirement engendré plusieurs améliorations à *Metasm*.

Nous attendons la cuvée 2008 avec impatience!

5 Securitech 2006 : vers une approche structurelle

La dernière partie de cet article sera consacrée à une technique d'obfuscation que nous avons déjà abordée très rapidement, l'obfuscation structurelle, ou obfuscation du graphe de contrôle. Pour illustrer notre propos, nous avons choisi le défi n°10 du challenge **Securitech 2006**. Ce défi a été proposé par Fabrice Desclaux.

Le binaire Il s'agit d'un exécutable Windows qui prend en entrée une chaîne de caractères, qu'il utilise pour générer une sorte de hash. L'objectif du challenge est de trouver la chaîne qui produira une sortie donnée. Le corps du binaire est massivement obfusqué, interdisant ou compliquant de manière critique toute tentative de rétro ingénierie.

L'objectif initial était d'obliger les participants à le résoudre en boîte noire, sans accès à l'implémentation. Notre objectif est d'éliminer la couche d'obfuscation pour retrouver l'algorithme exact utilisé.

5.1 Définition du graphe de contrôle

Le graphe de contrôle d'un binaire est une structure fondamentale, utilisée tant lors de la phase de compilation que lors d'une éventuelle phase d'analyse et désassemblage. Nous allons brièvement revenir sur les principales notions qui lui sont associées.

– Control Transfert Instruction (CTI)

Une *CTI*, comme son nom l'indique, est une instruction qui, par sa nature même, agit sur le flot d'exécution pour éventuellement le modifier. Dans cette catégorie nous retrouvons les sauts (conditionnels ou non), les appels (*call*) et leurs pendants : les retours (*ret*), ou encore les interruptions (*int*).

Il est important d'insister sur le fait que le but premier de ces instructions est précisément la maîtrise du flot d'exécution, à l'opposé d'une instruction, comme *mov*, qui peut éventuellement provoquer une exception (ex : pointeur nul) et donc un transfert du flot d'exécution, mais seulement comme effet de bord.

– Basic Block

Un bloc basique est une suite contiguë d'instruction dont seule la première peut être la cible d'une *CTI* et seule la dernière peut être une *CTI*. C'est l'unité atomique du graphe de contrôle, nous pourrions faire le parallèle avec les sections critiques qu'il est impossible de préempter en cours d'exécution.

Le graphe de contrôle est la mise en forme de ces deux notions. Les noeuds du graphe sont constitués par les blocs basiques. Les arcs représentent les relations existant entre ces différents blocs. Ces relations correspondent aux transferts d'exécution : saut, appel de fonction, retour, etc. Le graphe de contrôle est le niveau d'abstraction privilégié pour visualiser rapidement la logique du code : boucles *while*, *do-while*, *if-then-else*. . . C'est par exemple le mode de rendu sous forme de graphe que propose IDA⁷ depuis sa version 5.0.

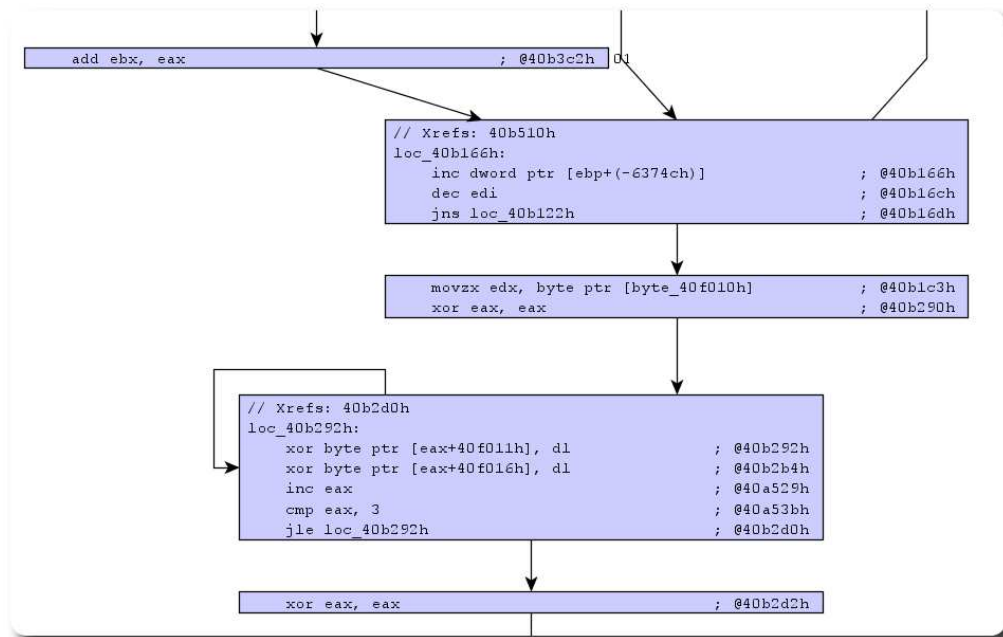
Lors du désassemblage, *Metasm* construit implicitement ce graphe de contrôle. En effet, en interne, l'objet *Disassembler* construit et gère des objets de type *InstructionBlock* qui sont la traduction de la notion que nous venons d'évoquer. Les arcs sont gérés avec une granularité très fine ainsi nous distinguons un arc *normal*, par exemple un saut, d'un arc *indirect* comme par exemple celui lié au retour d'une fonction. Afin de profiter de manière plus visuelle de ce niveau d'abstraction, un script a été développé afin de réaliser une passerelle entre *Metasm* et l'éditeur de graphe yEd⁸. Cet éditeur accepte en entrée un fichier de type *graphml*, c'est un format de fichier basé sur *XML* et dédié à la description de graphes. Le script génère un fichier *graphml* à partir de la représentation du graphe interne à *Metasm*.

5.2 Définition des prédicats

D'un point de vue fonctionnel, un saut conditionnel est un prédicat suivi d'un branchement. Dans le cadre d'un saut *légitime*, un premier ensemble d'états initiaux (nommons le *A*) provoque l'exécution de l'une des deux branches et un second ensemble (*B*), complémentaire, provoque l'exécution de l'autre branche. *A* et *B* couvrent l'ensemble des états possibles et sont parfaitement définis par le prédicat. Un saut inséré lors d'un processus d'obfuscation peut être vu comme corrompu, et, le plus souvent, nous allons constater un biais dans le prédicat.

⁷ Interactive DisAssembler, <http://www.hex-rays.com/idapro/>

⁸ http://www.yworks.com/en/products_yed_about.html

FIG. 39: Couplage de yEd et *Metasm*

Les prédicats obscurs. Dans ce cas de figure le prédicat est une fonction booléenne renvoyant toujours *vrai* ou toujours *faux*. La fonction prédicat pourra elle-même être suffisamment complexe et/ou obfusquée pour ne pas être déchiffrable de manière triviale[4].

```
if( x^4*(X-5)^2 >= 0) {
    goto real_code;
} else {
    goto no_man's_land;
}
```

FIG. 40: Prédicat obscur (*pseudo-code*)

Ces quelques lignes de *pseudo-code* (Fig. 40) reflètent tout à fait ce que nous venons de dire. Le prédicat est, dans le cas présent, constitué par un polynôme dont la valeur est toujours positive ou nulle. La fonction prédicat renvoie donc toujours *vrai*. Décrit de cette manière, le biais est très facile à deviner, en revanche regardons la compilation de cette même fonction par *GCC 4.1.2* (Fig. 41).

```
loc_8048403h:
fstp qword ptr [esp+8]          ; @8048403h dd5c2408
fstp qword ptr [esp]           ; @8048407h dd1c24
call thunk_pow                 ; @804840ah e8e5feffff
fstp qword ptr [ebp+(-20h)]    ; @804840fh dd5de0
mov eax, dword ptr [ebp+(-0ch)] ; @8048412h 8b45f4
sub eax, 5                     ; @8048415h 83e805
push eax                       ; @8048418h 50
fld dword ptr [esp]            ; @8048419h db0424
lea esp, dword ptr [esp+4]     ; @804841ch 8d642404
fld qword ptr [xref_8048590h]  ; @8048420h dd0590850408
fstp qword ptr [esp+8]        ; @8048426h dd5c2408
fstp qword ptr [esp]          ; @804842ah dd1c24
call thunk_pow                 ; @804842dh e8c2feffff
fld qword ptr [ebp+(-20h)]    ; @8048432h dd45e0
fmulp ST(1)                    ; @8048435h dec9
fldz                           ; @8048437h d9ee
fxch ST(1)                     ; @8048439h d9c9
fucompp                         ; @804843bh dae9
fnstsw                          ; @804843dh dfe0
sahf                            ; @804843fh 9e
jnbe loc_8048444h              ; @8048440h 7702
jmp loc_80484452h              ; @8048442h eb0e
```

FIG. 41: Assembleur produit lors de la compilation de la fonction prédicat

Cette implémentation fait un usage massif de calculs en virgule flottante (*FPU*). Retrouver la sémantique d'une telle portion de code lors de sa lecture est loin d'être immédiat. En revanche, un émulateur dans une approche dynamique, ou un outil d'analyse statique[5], pourront être à même

d’automatiser la découverte et la caractérisation d’un tel prédicat. Cet exemple est volontairement basique, et il est relativement aisé d’atteindre des niveaux de difficulté bien supérieurs.

Enfin, dans notre exemple de prédicats obscurs, la branche correspondant au *else* est une branche morte : elle n’est jamais exécutée. Pour ajouter encore de la confusion à cette structure, il est possible (et même recommandé) de dupliquer d’importantes portions de code dans cette branche, et/ou de créer de fausses pistes pour fourvoyer les moteurs désassemblage — par exemple en la faisant pointer au beau milieu d’une instruction réelle.

L’aléa total. Maintenant considérons que l’une ou l’autre des branches est empruntée indifféremment. Le prédicat se transforme alors en une simple source d’aléa. Afin de préserver la sémantique du code, les deux branches seront sémantiquement équivalentes. La duplication du code peut être vue comme un facteur négatif, notamment par la croissance de la taille du binaire ; pour cette raison, les portions de code ainsi protégées sont généralement limitées, ce qui fait apparaître très rapidement des structures de type *diamant*.

```
if(rand()%2) {
    real_code_A
} else {
    real_code_B
}
```

FIG. 42: Aléa total (*pseudo-code*)

La seule condition imposée par cette structure (Fig. 42) est que **real_code_A** et **real_code_B** soient sémantiquement équivalents. Cette technique est d’autant plus efficace que deux exécutions du même binaire pourront donner deux traces différentes. Nous avons déjà abordé cette technique, mais sans la formaliser, dans la partie consacrée au challenge *T2* (ch 4.2) où elle ne constituait qu’une infime partie de la protection.

L’arroseur arrosé. Les sauts conditionnels insérés à des fins d’obfuscation font très souvent apparaître des prédicats biaisés, que ce soit sous la forme d’un prédicat obscur ou d’un aléa. À ce titre, ils présentent eux-mêmes des caractéristiques susceptibles d’être détectées puis analysées. Ainsi un outil d’interprétation capable de modéliser la fonction mathématique sous-jacente à un prédicat sera à même de déduire la nature du saut.

Du point de vue d’un développeur de protection, il est intéressant de masquer le plus possible la véritable nature du prédicat. Ainsi, concernant l’aléa total, il pourra prendre soin de faire intervenir des variables significatives dans le calcul du prédicat fictif afin de tromper l’analyste (ou un outil) sur sa nature. Cette précaution a, en partie, été prise sur le *T2*, mais de manière beaucoup trop faible. Le simple fait de retrouver une instruction *RDTSC* dans le calcul d’un prédicat, qui plus en en mode privilégié, suffit à faire naître le doute quant à la finalité du calcul et par ricochet du saut.

De la même manière dans l'utilisation des prédicats obscurs, les fonctions seront généralement suffisamment complexes et variées, tant sur le papier que dans leur implémentation, pour circonvenir ou retarder toute forme d'analyse.

5.3 Portrait of a man

Avant d'aller plus loin il convient de rappeler un point essentiel. Nous avons pu reconstruire le graphe de contrôle intégral du binaire grâce à la pertinence du désassemblage proposé par *Metasm*. Nous pouvons dire que *Metasm* se sert des données, et donc du *dataflow*, pour améliorer sa connaissance du *controlflow* ou graphe de contrôle, qui lui même revient enrichir le *dataflow*... La boucle est bouclée. Pour étudier un binaire protégé, un moteur de désassemblage ne peut plus se contenter d'une simple traduction de l'opcode en son équivalent textuel, la mnémonique. Un des atouts de *Metasm* est sa capacité à exprimer de manière abstraite la sémantique de l'instruction et de *backtracker* les effets des instructions afin de d'enrichir son désassemblage. Cela étant dit, nous pouvons nous concentrer sur le challenge lui même. Il utilise massivement l'obfuscation pour protéger l'algorithme qu'il renferme. En réalité il met principalement en oeuvre cinq méthodes différentes :

Insertion de sauts inconditionnels. Les blocs basiques sont réordonnés et des sauts sont insérés pour préserver la logique du code. Un parallèle immédiat peut se faire avec un *round* de permutation dans un algorithme cryptographique. Cette technique est efficace contre un analyste qui essaie de tracer pas-à-pas le code à l'aide d'un debugger par exemple. Il sera "promené" d'un bout à l'autre de l'exécutable et toute reconstruction d'une logique de plus haut niveau sera très perturbée. En revanche avec des outils de visualisation du graphe de contrôle (IDA (quand il y arrive;)), *Metasm* couplé à yEd...), elle se révèle tout à fait inopérante.

Émulation de saut. Cette technique peut s'envisager dans le prolongement de celle évoquée précédemment, elle est seulement un tout petit peu plus évoluée dans son implémentation. Elle consiste à pousser une adresse sur la pile et à sauter à cette adresse à l'aide de l'instruction *ret*.

```
push loc_4042fch ; @4027adh 68fc424000
ret             ; @4027b2h  c3
```

FIG. 43: *push-ret* utilisé comme un saut

L'intérêt de cette construction est que seule la connaissance de la sémantique des deux instructions permet de poursuivre la construction du graphe de contrôle lors du désassemblage.

Insertion de faux appels. Sans revenir sur les principes de l'appel, nous pouvons dire qu'une particularité de ces appels insérés est de modifier leur adresse de retour sur la pile. Cette propriété

```

push esi           ; @401873h 56
push ebx           ; @401874h 53
call loc_403592h   ; @401875h e8181d0000 x:loc_403592h
push ebx           ; @403595h 53
add dword ptr [esp+4], 4 ; @403598h 8344240404
add esp, 4         ; @409d3eh 83c404
ret 8              ; @40c17bh c20800 x:loc_40187e

```

FIG. 44: Squelette d'un faux appel

est très efficace car elle met en échec certains moteurs de désassemblage qui font souvent l'hypothèse qu'un appel retourne à l'instruction suivant le *call*.

Nous avons épuré cet exemple (Fig. 44) afin de ne montrer que le code spécifique au pattern. Dans le binaire, ces instructions sont entremêlées avec d'autres patterns et des instructions *réelles*. Ce pattern est polymorphe tant au niveau du *delta* appliqué sur l'adresse de retour que du nombre de registres poussés sur la pile, mais reste néanmoins très caractéristique. Il est à noter qu'il existe du code *mort* entre l'instruction *call* et l'adresse de retour effective, 4 octets dans cet exemple.

Duplication de flot. Voici la mise en application d'un prédicat biaisé, avec une forme d'aléa total. Le moteur d'obfuscation sélectionne une portion de code, généralement limitée, et la duplique dans les deux branches d'un saut conditionnel.

Comme les deux branches attenantes au saut conditionnel sont sémantiquement équivalentes, peu importe la source d'aléa, l'essentiel étant de la faire apparaître comme "plausible" pour perturber l'analyste.

L'aléa apparent. Cette technique est la conséquence directe de la précédente. Couplée à l'insertion de sauts conditionnels, nous allons retrouver l'insertion d'aléa, qui va ici prendre la forme d'instructions *test* et *cmp*. Pour rappel, ces deux instructions mettent à jour les flags en comparant les deux opérandes qui leur sont passées.

L'exemple (Fig. 46) résume les forces et les faiblesses de cette technique. À première vue, il est tentant de chercher l'origine des registres *edi* et *ebp* utilisés par l'instruction *test*, d'où une perte de temps et une source confusion. Or, sur l'architecture *IA32*, et à l'inverse d'une architecture *ARM* par exemple, de nombreuses instructions mettent implicitement à jour les flags, en particulier les instructions arithmétiques. En reprenant l'exemple, nous voyons alors que les flags qui déterminent le branchement du saut *jnz* (l. 6) seront réécrits deux fois par les instructions *add* et *and*. Ceci constitue la principale faiblesse de cette technique, il est relativement aisé de déterminer si l'instruction de comparaison est légitime ou non.

5.4 Analyse et réduction du graphe de contrôle

Pour se représenter l'ampleur du problème, voici le graphe de contrôle de l'épilogue de la fonction principale, tel qu'on le trouve dans le binaire protégé (Fig. 47).

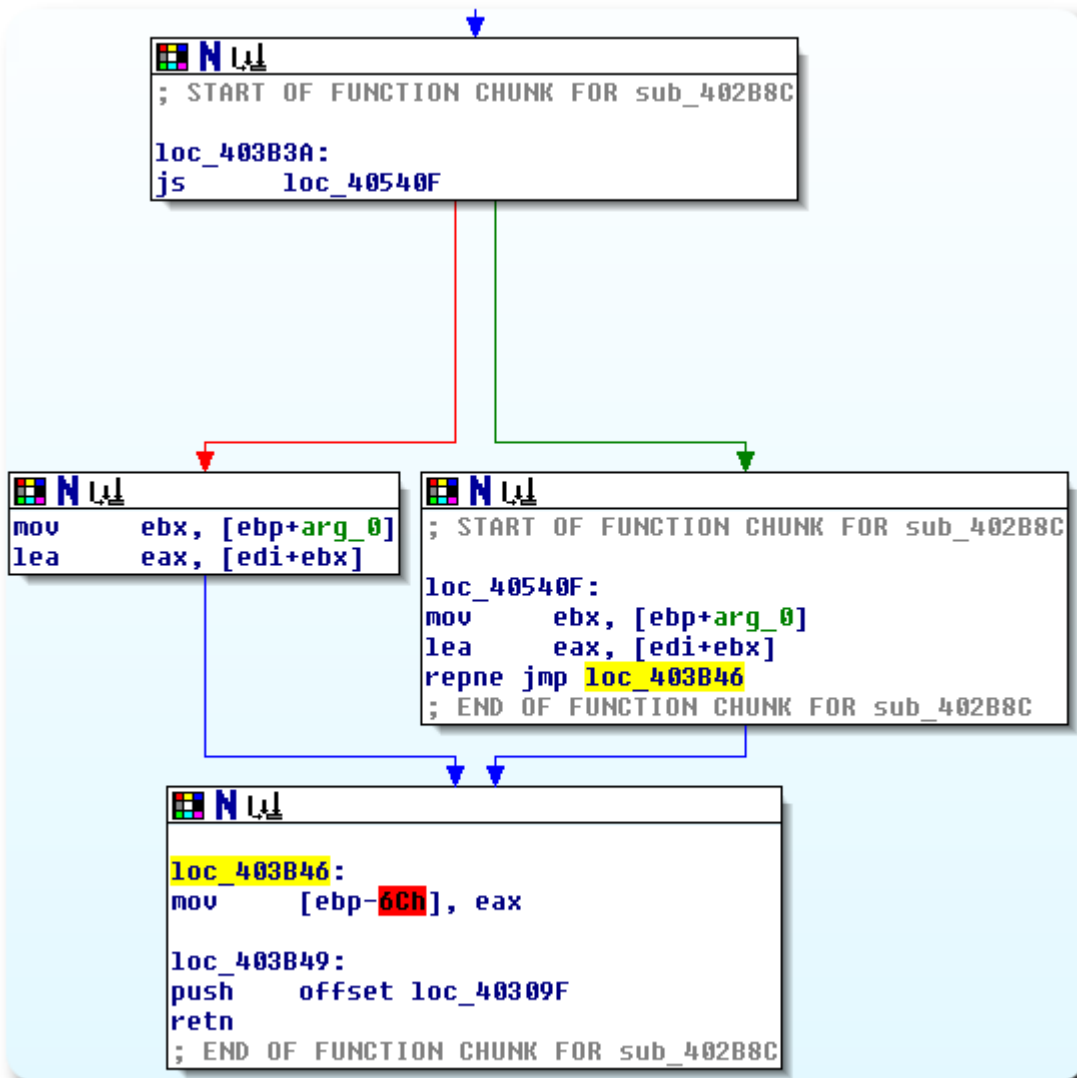


FIG. 45: Duplication de flot et insertion de faux prédicat, vu par IDA

```

402C44h test    edi, ebp
402C46h mov    ebx, [ebp+arg_C]
402C49h mov    esi, edi
402C4Bh add    [ebx], edi
402C4Dh and    esi, 3Fh
402C50h jnz    short loc_402C5A

```

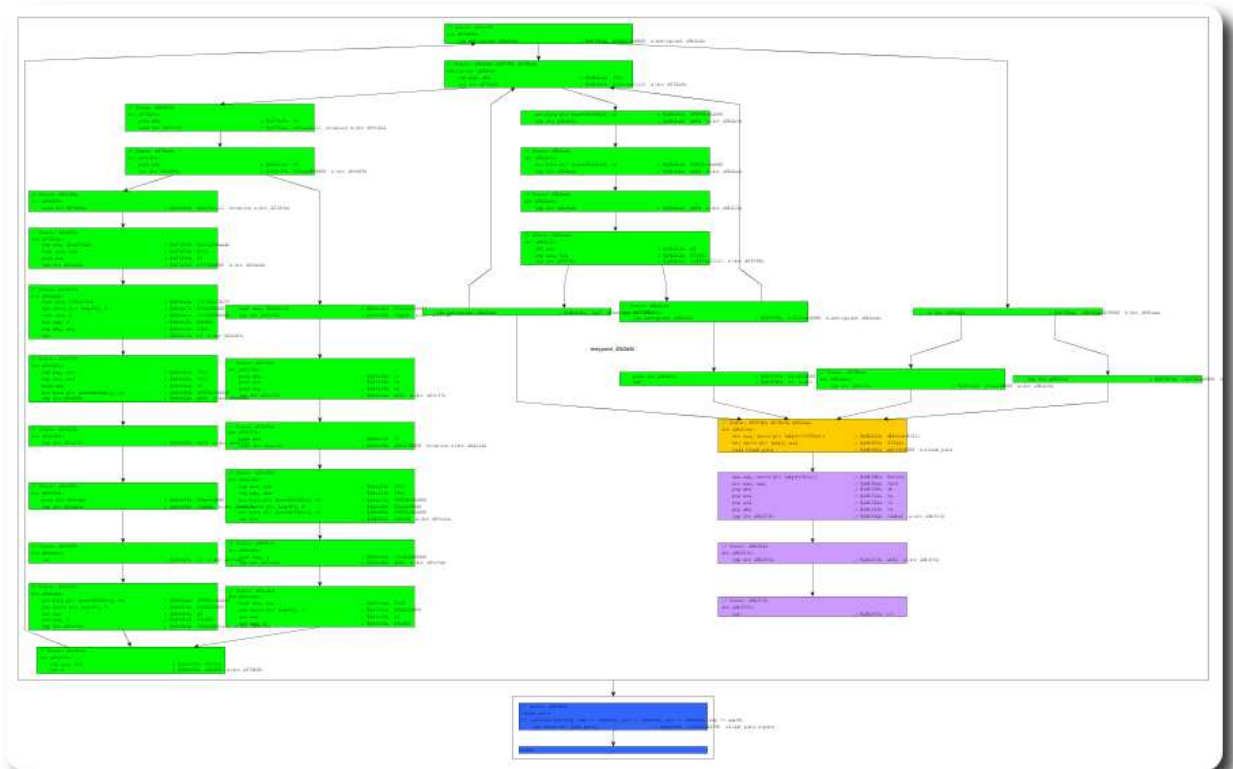
FIG. 46: Insertion d'une instruction *test*

FIG. 47: Épilogue protégé de la fonction principale

Notre approche met à profit notre connaissance du graphe de contrôle complet du binaire. À partir d'un point d'entrée fixé, nous explorons linéairement les blocs jusqu'à rencontrer un saut conditionnel. Lorsque le cas se présente, nous construisons les flots d'exécution associés à chacune des deux branches. Ce processus est récursif afin de gérer le cas d'un code dupliqué plusieurs fois.

Une fois ces deux flots en notre possession, le traitement se passe en plusieurs étapes :

1. Suppression des instructions *test* et *cmp* abusives, suivant la méthode développée précédemment. *Metasm* associe à chaque instruction sa sémantique au travers d'une expression abstraite, qui permet de déterminer si une instruction lit et/ou écrit les flags du processeur.
2. Suppression des sauts inconditionnels insérés pour compenser la réorganisation des blocs.
3. Suppression des faux appels, leur structure est suffisamment caractéristique pour être reconnue. Le flot d'exécution est parcouru, et la présence d'un *ret* provoque le traitement des instructions intervenant potentiellement dans le pattern, que nous stockons au fur et à mesure du parcours dans une pile. Nous pouvons alors vérifier la cohérence de la structure, afin d'éviter les faux positifs (i.e. suppression de code réel).
4. Comparaison des deux flots nettoyés. Par souci de simplicité, nous avons implémenté une simple comparaison textuelle des deux flots, instruction par instruction. Dans le cas d'une protection plus évoluée, avec par exemple utilisation de poly/meta-morphisme au niveau des flots dupliqués, nous aurions pu mettre en oeuvre une analyse comportementale à la manière de ce qui a été fait sur le challenge T2 pour reconnaître le comportement de chacun des handlers. Dans le cas présent, l'option simple est tout à fait satisfaisante.

Si les deux flots sont équivalents, nous sommes en présence d'une duplication artificielle. Dans ce cas, nous supprimons le saut conditionnel et modifions le graphe de contrôle afin de ne conserver qu'une seule des deux branches. Il est aussi à noter que toutes les instructions taguées comme *illégitimes* lors du nettoyage des flots sont supprimées du listing final tel qu'il est présenté à l'utilisateur. Voici un résultat intermédiaire sur lequel nous pouvons constater que le nombre de blocs a été divisé par 5 environ (Fig. 48).

Le résultat est donc déjà agréable, mais pas encore optimal. Des portions du binaire ne sont toujours pas nettoyées : en effet, seuls les flots mis en jeu dans le cadre d'un test de duplication sont purgés des instructions de *junk code*. Pour répondre à cette problématique, nous faisons une passe finale, qui parcourt l'intégralité du graphe de contrôle et nettoie l'ensemble des flots rencontrés. Afin d'obtenir une présentation plus alléchante, nous agrégeons également les blocs contigus. Le résultat final est très satisfaisant : nous retrouvons le code original entièrement débarrassé de la protection. Le nombre de blocs est réduit d'un facteur 10 environ au total (Fig. 49).

Sur l'ensemble du graphe de contrôle du binaire, le facteur de réduction mesuré est environ égal à 7.7 concernant le nombre de blocs. À cela s'ajoute bien sûr une réduction très importante de la taille du code : sur l'ensemble du code protégé, c'est environ 70% des instructions qui sont éliminées du listing final.

5.5 La cerise sur le gâteau : l'interopérabilité

Nous disposons alors d'un code désassemblé très proche de l'original, nous allons l'exploiter et reconstruire un exécutable vierge de toute protection, à partir du binaire original et du listing

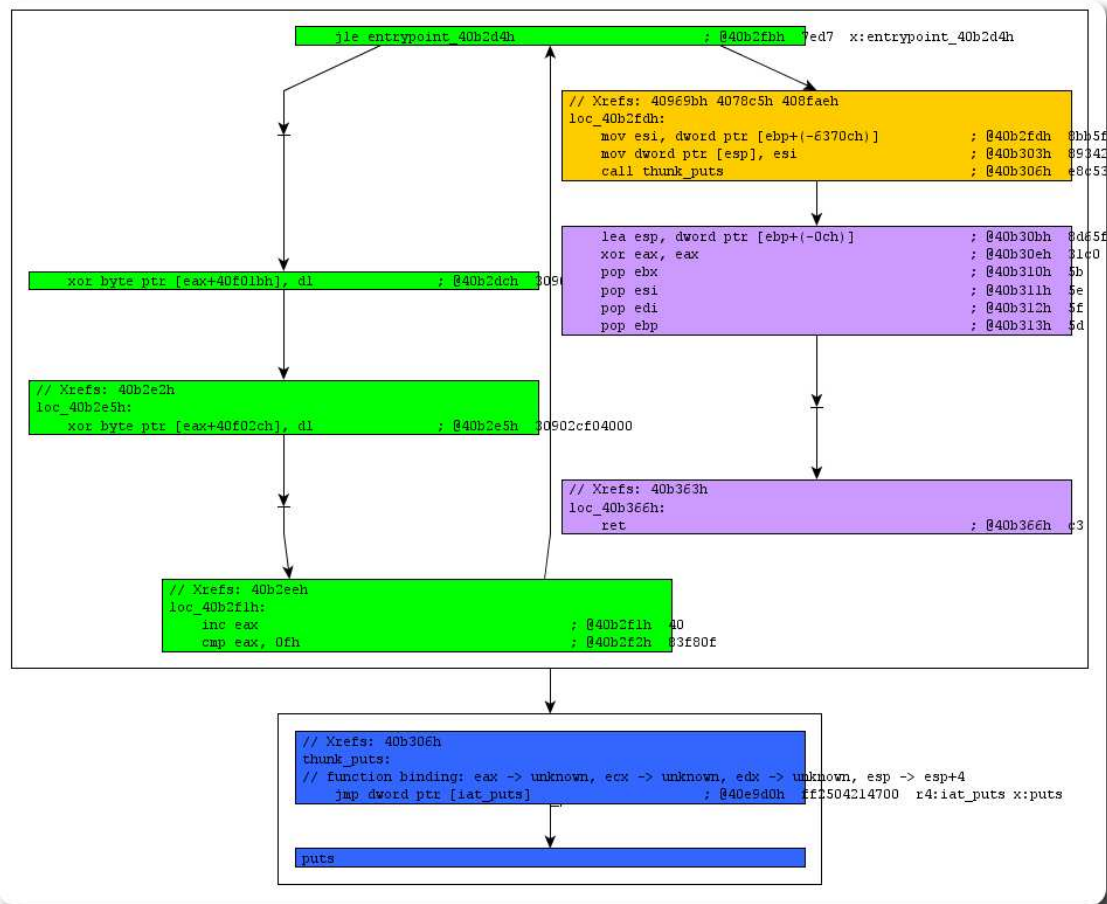


FIG. 48: Épilogue sans les flots dupliques

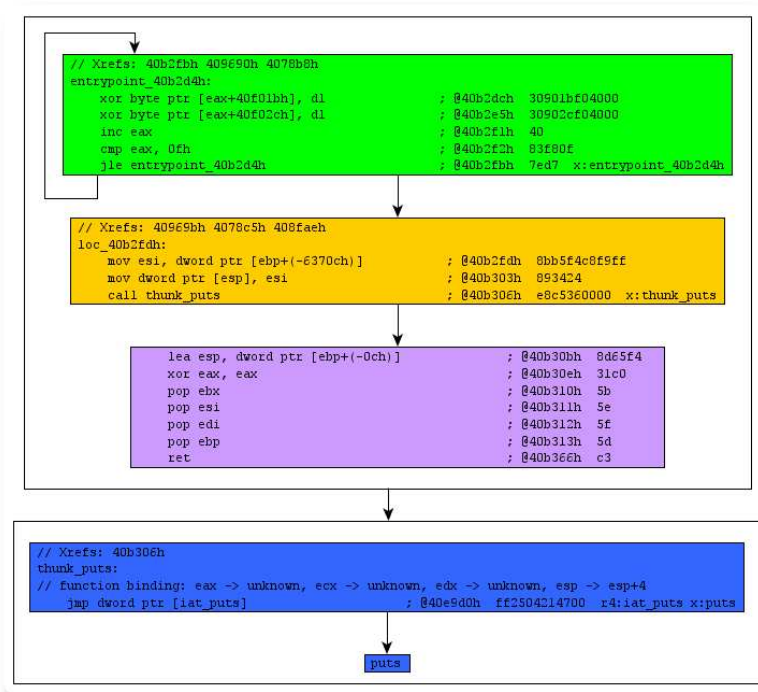


FIG. 49: L'épilogue entièrement nettoyé et réordonné

résultant de notre nettoyage.

```
require 'metasm'
include Metasm

# lecture du binaire
file = 'poeut.exe'
pe = PE.decode_file file
pe.cpu = pe.cpu_from_headers

# compilation de la section de code clean
src = File.read('poeut.asm').sub('entrypoint:', '')
pe.parse '.section ".clrtext" rx'
pe.parse '.entrypoint'
pe.parse src
pe.assemble

# resolution des labels vers .data etc
text = pe.sections.last.encoded
text.fixup! 'loc_0' => 0, 'loc_1' => 1
text.reloc.values.map { |r| r.target.reduce_rec }.grep(&:String).uniq.sort.each { |t|
  if t =~ /^(?:dword|byte)_(4\w+)h/ and not text.export[t]
    rva = $1.to_i(16) - pe.optheader.image_base
    s = pe.sections.find { |s| s.virtaddr <= rva and s.virtaddr + s.virtsize >= rva }
    s.encoded.add_export t, rva-s.virtaddr
  end
}

# generation du binaire clean
pe.encode_file 'unpoeut.exe'
```

FIG. 50: Script régénérant un exécutable “clean”

Le binaire protégé est une simple application console, qui ne fait appel qu'à très peu de bibliothèques systèmes : il est possible de le porter vers un format *ELF* sans trop de difficultés.

Nous remplaçons le *stub GCC* situé avant le point d'entrée par notre propre code (Fig. 51).

Le binaire original commence par allouer une très grande zone sur la pile, et la routine réalisant cette allocation n'est pas compatible avec Linux (où il faut d'abord décaler le pointeur de pile avant de pouvoir accéder aux pages souhaitées, alors que la routine Windows touche d'abord les pages avant de modifier *esp*, ce qui génère un *SEGFAULT*). Nous allons donc rajouter une séquence dans notre loader qui va transférer la pile dans le *heap* avant de donner la main au code original.

Metasm intégrant un compilateur, nous n'avons même pas besoin d'utiliser un programme extérieur pour disposer d'un exécutable utilisable sur une plateforme unix.

Bon, dans le fond, cela ne sert à rien, mais ça fait plaisir.

5.6 Résolution

Disposant d'un code en clair très proche de l'original, le défi se résout bien. Le mot de passe entré est découpé en plusieurs blocs de caractères manipulés afin de produire la sortie : dans les

```
.entrypoint hooked_entrypoint
hooked_entrypoint:
    ; alloc heap space for 'stack'
    push 0x80100
    call malloc
    add esp, 4
    lea ebp, [eax+0x80000]
    ; disable libc init (weird things w/ FindAtomA)
    or dword ptr [40fc00h], 1

    ; init argc/argv/envp as args for main
    mov ecx, [esp]
    mov [ebp], ecx
    lea eax, [esp+4]
    mov [ebp+4], eax
    lea eax, [eax+4*ecx+4]
    mov [ebp+8], eax

    ; call main w/ new stack
    mov esp, ebp
    call loc_403db0h

    ; exit
    mov eax, 1
    mov ebx, 0
    int 80h

; puts is not autoresolved to libc
puts:
    push [esp+4]
    push puts_format
    call printf
    add esp, 8
    ret
```

FIG. 51: Code nécessaire à la conversion du binaire en ELF

valeurs calculées, nous trouvons la somme et le produit d'un sous-groupe de caractères, un *CRC* d'une autre partie, ainsi que le *MD5* du reste.

Ces différentes conditions permettent, après un petit bruteforce, de trouver la solution au problème initial.

Le code comporte cependant de nombreux calculs qui ne sont pas utilisés pour générer la chaîne résultat, qu'il serait certainement intéressant d'analyser.

6 Conclusion

Au travers des exemples mis en avant, nous avons essayé de montrer l'utilité d'outils proposant toujours plus d'abstraction dans les approches de reverse-engineering, et en particulier l'étude de code protégé.

Le reverse-engineering consiste à remonter une chaîne de niveaux d'abstraction. À la base de cette chaîne, se situe l'unité élémentaire d'information qu'est l'instruction. Elle se caractérise avant tout par un comportement. C'est la modélisation abstraite du comportement qui permet à *Metasm* d'implémenter un backtracking efficace. L'instruction textuelle est remplacée par sa sémantique. Cette notion est la base de nos travaux, sur le challenge Securitech, cela nous a initialement permis d'obtenir le graphe de contrôle complet là où un désassembleur classique s'arrêterait sur les patterns d'obfuscation les plus basiques.

Amenée à un stade supérieur, cette capacité nous a conduit à aborder des notions d'analyse comportementale : s'abstraire de l'implémentation du code, pour ne s'intéresser qu'à sa sémantique. Ainsi nous avons identifié de manière automatique le comportement des handlers de la machine virtuelle, et enfin, modélisé un processeur virtuel pour résoudre le *T2*. L'aspect comportemental que nous avons effleuré dans cet article est très prometteur, et pourrait par la suite se développer en tirant parti de résultats obtenus dans le domaine de l'analyse statique[5].

Nous avons insisté sur le caractère semi-automatique des approches proposées, car toutes reposaient sur une part d'analyse manuelle : la prise de conscience de l'existence d'une machine virtuelle, la reconnaissance de schémas d'obfuscation, de la duplication de flots. . . Si la reconnaissance de patterns facilement automatisable, leur découverte et leur caractérisation reste un procédé manuel : l'automatisation permet surtout d'appliquer un très grand nombre de fois une technique générique trouvée artisanalement.

De nos approches se dégage une constante : pour comprendre une protection logicielle, il faut se placer à un niveau d'abstraction supérieur ou égal au sien. Nous concluons en disant que *Metasm* est un framework puissant de manipulation de binaires, qui permet d'interagir à tous les niveaux d'abstraction : depuis le plus bas, le matériel, jusqu'au niveau le plus haut, le code source.

Références

1. Collberg, C., Thomborson, C., Low, D. : A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland (July 1997)
2. Guillot, Y. : Metasm. In : 5ème Symposium sur la Sécurité des Technologies de l'Information et des Communicatins (SSTIC'07). (2007)

3. Guillot, Y. : Metasm, a ruby (dis)assembler. In : HACK.LU. (2007)
4. Collberg, C., Thomborson, C., Low, D. : Manufacturing cheap, resilient, and stealthy opaque constructs. In : Principles of Programming Languages 1998, POPL'98. (1998) 184–196
5. Allamigeon, X., Hymans, C. : Analyse statique par interprétation abstraite : Application à la détection de dépassement de tampon. In : 5ème Symposium sur la Sécurité des Technologies de l'Information et des Communicatins (SSTIC'07). (2007) 347–384