

LES GUIDES DE



LE MAGAZINE DE LA SÉCURITÉ INFORMATIQUE MULTIPLATEFORME!

HORS-SÉRIE
N°17

France MÉTRO. : 12,90 € — CH : 18,00 CHF — BEL/PORT. CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 \$ CAD

RECHERCHE DE

VULNÉRABILITÉS

DÉCOUVREZ DE NOUVEAUX OUTILS POUR AUDITER VOS APPLICATIONS

```
/* Enter response type, length and copy payload into the allocated memory for the response
message type, plus 2 bytes payload, plus padding
payload, plus padding
*/
buffer = OPENSSL_malloc(1 + 2 + payload);
bp = buffer;

/* Enter response type, length and copy payload into the allocated memory for the response
message type, plus 2 bytes payload, plus padding
payload, plus padding
*/
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

= ssl3_write_bytes(s, TLS1_RT_HEARTBEAT,
r >= 0 && s->msg_callback)
s->msg_callback(1, s->version,
buffer, 3 + payload,
s, s->msg_callback);
```

ANALYSE STATIQUE & DYNAMIQUE

Fuzzing, exécution symbolique, reverse engineering : d'afl à r2, reprenez les bases pour la recherche de vulnérabilités

CRYPTOGRAPHIE

De l'identification de primitives vulnérables dans des binaires aux tests automatisés : initiez-vous à grap et à CDF - Crypto Differential Fuzzing

IOT

Prenez en main l'ensemble des outils nécessaires pour tester la sécurité des objets connectés

WEB

XSS, injection SQL : mettez en œuvre les outils pour déceler les problèmes de sécurité dans vos applications web

Édité par Les Éditions Diamond

L 16844 - 17 H - F : 12,90 € - RD



www.ed-diamond.com

Retrouvez toutes nos publications



sur <http://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 : <https://www.miscmag.com>
<https://www.ed-diamond.com>

Directeur de publication : Arnaud Metzler

Chef des rédactions : Denis Bodor

Rédacteur en chef : Cédric Foll

Rédacteur en chef adjoint : Émilien Gaspar (gapz)

Secrétaire de rédaction : Aline Hof

Responsable service PAO : Kathrin Scali

Responsable publicité : Tél. : 03 67 10 00 27

Service abonnement : Tél. : 03 67 10 00 20

Impression : Loire Offset Titoulet, Saint-Etienne

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 France - PRINTED in France

Dépôt légal : A parution

N° ISSN : 1631-9036

Commission Paritaire : K 81190

Périodicité : Bimestrielle

Prix de vente : 12,90 Euros

Première parution : Mars 2018

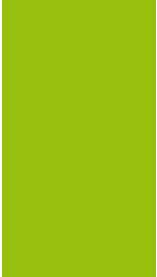


La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans Misc est interdite sans accord écrit de la société Les Éditions Diamond. Sauf accord particulier, les manuscrits, photos et 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

« Aucune règle n'existe, les exemples ne viennent qu'au secours des règles en peine d'exister. »

André Breton

La recherche de vulnérabilités est un domaine vaste, tout comme les outils dont dispose le chercheur en sécurité. Les colonnes de *MISC* ont ainsi déjà vu bon nombre de ces programmes, incontournables pour certains, mais l'évolution du domaine nous apporte chaque jour son lot de scripts, bouts de code, plugins, preuves de concept et frameworks en tout genre. Est en cause notamment la grande diversité des cibles et des méthodes. D'une part, que l'on s'attaque à du web, de la cryptographie ou à tout ce que l'on peut trouver dans l'Internet des objets, force est de constater qu'il va être nécessaire, pour chaque cas, d'avoir un outillage spécifique pour peu que celui-ci existe et qu'il ne faille pas en développer un. De la même manière, les techniques utilisées sont extrêmement variées : reverse, fuzzing, analyse statique automatisée, exécution symbolique ou simple lecture de code, pour ne prendre que quelques exemples.

Dans cette étendue de possibilités, l'approche du présent hors-série va être de s'intéresser aux outils récents et qui présentent un intérêt dans un domaine particulier de la recherche de vulnérabilités. Les pages qui vont suivre vont se concentrer essentiellement à décrire les principes sur lesquels reposent les différents programmes sélectionnés dans ce hors-série ainsi que leurs usages via des exemples pratiques. Les limites vont être que l'on ne parlera pas d'exploitation et que les connaissances théoriques nécessaires à l'approche de certains domaines seront abordés partiellement et nécessiteront d'autres lectures pour ceux qui seraient plus intéressés. Il est d'ailleurs important de noter que, quel que soit l'outil, quelle que soit la technique, les connaissances élémentaires restent l'arme la plus forte. À quoi bon trouver des dizaines de crashes en utilisant du fuzzing si l'on n'est pas capable d'interpréter minimalement les résultats ?

Le numéro va s'articuler autour de quatre thématiques. Il va s'agir dans un premier temps de découvrir ou d'approfondir des outils permettant de faire de la rétro-ingénierie (radare2), du fuzzing (afl) et de l'exécution concolique (angr). S'ensuivra une partie passionnante et accessible sur la cryptographie : initiation à CDF (*crypto differential fuzzing*) et Wycheproof pour tester la qualité de votre code cryptographique, et découverte de grap, un outil aux nombreuses possibilités et dont on verra une application pour détecter des algorithmes cryptographiques obsolètes dans des binaires. Une troisième partie sera consacrée à la prise en main des outils nécessaires à l'analyse des objets connectés. Enfin, dans la dernière partie sera traité le domaine du web avec la découverte de Wapiti et Snuffleupagus, un scanneur de vulnérabilités et un module de sécurité pour php7.

Bonne lecture,

Émilien Gaspar / gapz / eg@miscmag.com

Sommaire

MISC
Hors-Série

N°17

```
buffer = OPENSSL_malloc(1  
bp = buffer;
```

```
/* Enter response type, l  
*bp++ = TLS1_HB_RESPONSE;  
s2n(payload, bp);  
memcpy(bp, pl, payload);  
bp += payload;
```

```
/* Random padding */  
RAND_pseudo_bytes(bp, pad
```

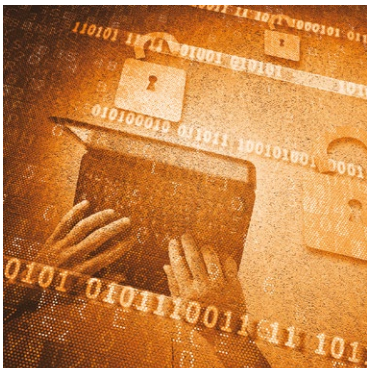
```
= ssl3_write_bytes(s, T
```



ANALYSE STATIQUE & DYNAMIQUE

Fuzzing, exécution symbolique, reverse engineering : d'afl à r2, reprenez les bases pour la recherche de vulnérabilités

- p.10 Analyse concolique de binaires avec angr
- p.22 Introduction au développement de plugins pour Radare2
- p.44 Usages avancés d'AFL



CRYPTOGRAPHIE

De l'identification de primitives vulnérables dans des binaires aux tests automatisés : initiez-vous à grap et à CDF - Crypto Differential Fuzzing

- p.66 Testons votre crypto !
- p.76 Algorithmes et implémentations cryptographiques vulnérables : détection avec grap

RECHERCHE DE VULNÉRABILITÉS

**DÉCOUVREZ DE NOUVEAUX OUTILS POUR
AUDITER VOS APPLICATIONS**



IOT

Prenez en main l'ensemble des outils nécessaires pour tester la sécurité des objets connectés

p.92 Toolkit 4.0 : revue des outils pour l'exploitation pour l'IoT



WEB

XSS, injection SQL : mettez en œuvre les outils pour déceler les problèmes de sécurité dans vos applications web

p.110 Recherche passive de vulnérabilités web avec Snuffleupagus

p.116 Wapiti : chasser les vulnérabilités web

**LISEZ LE DERNIER NUMÉRO PARU
EN LIGNE + 75 AUTRES NUMÉROS
ET 16 HORS-SÉRIES**



PROFESSIONNELS, ENSEIGNEMENT, R&D...
DÉCOUVREZ CONNECT
LA PLATEFORME DE LECTURE EN LIGNE !

**UN ACCÈS ILLIMITÉ À
+ DE 950 ARTICLES**

**DES ARTICLES TRIÉS
PAR DOMAINES**

**UN ACCÈS
MULTI-UTILISATEURS**

**UN AFFICHAGE PAR
NUMÉRO PARU**

**UN FILTRAGE PAR
AUTEUR**

**UN OUTIL DE
RECHERCHE**



ABONNEMENTS & RENSEIGNEMENTS



+33 (0)3 67 10 00 28



connect@ed-diamond.com



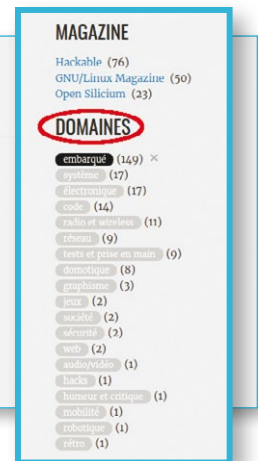
www.ed-diamond.com

SUR : connect.ed-diamond.com

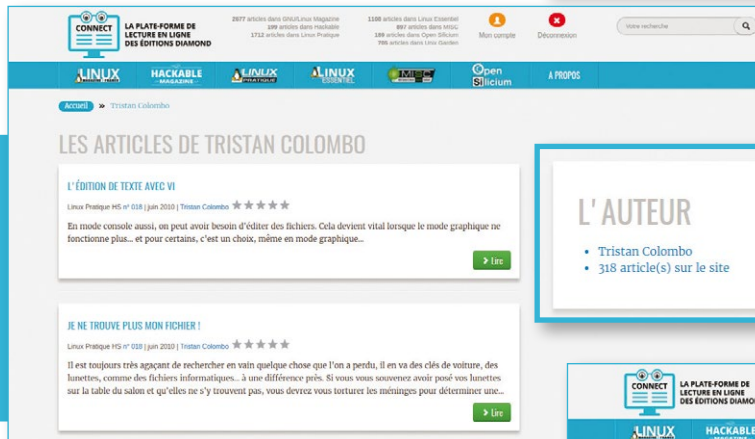
On retrouve la date de publication de chaque magazine, sa couverture ainsi qu'un sommaire cliquable des articles le composant.



Les articles peuvent être filtrés par domaine.



Un clic sur un auteur va lister l'ensemble des articles qu'il a rédigé.



Les résultats des recherches peuvent être filtrés par publication, domaine et tag.



» ET BIEN PLUS ENCORE...

- un sommaire cliquable de chaque article,
- l'affichage dans un nouvel onglet des sites référencés,
- la possibilité de copier-coller du code ou encore d'évaluer les articles...

BON À SAVOIR : l'abonnement à la plateforme de lecture en ligne CONNECT pourra être personnalisé en fonction du nombre de personnes composant son équipe grâce à nos diverses formules.



1

ANALYSE STATIQUE & DYNAMIQUE

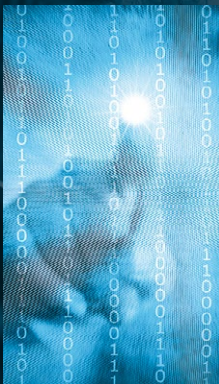
À découvrir dans cette partie...



ANALYSE / RÉTRO-INGÉNIERIE / PYTHON

Analyse concolique de binaires avec angr

L'exécution concolique ne vous dit rien ? L'analyse de binaires vous semble être une longue et pénible tâche impossible à automatiser ? Initiez-vous à angr, framework vous permettant de décortiquer vos binaires grâce à l'exécution concolique. p.10



RADARE2 / ANALYSE PAR TEINTE / PLUGIN / PYTHON

Introduction au développement de plugins pour Radare2

Envie d'essayer un outil de reverse-engineering autre qu'IDA et qui ne semble pas totalement étranger à votre distribution libre préférée ? Découvrez radare2 et les mécanismes pour développer des plugins pour ce dernier. p.22



AFL / FUZZING / CODE-COVERAGE / LLVM / VULNÉRABILITÉS

Usages avancés d'AFL

Déjà familier des techniques de fuzzing ainsi que du célèbre American Fuzzy Lop ? Cet article vous propose d'améliorer votre usage d'afl via quelques astuces, mais aussi d'aller plus en profondeur en découvrant le fuzzing différentiel. p.44

ANALYSE STATIQUE & DYNAMIQUE

ANALYSE / RÉTRO-INGÉNIERIE / PYTHON

ANALYSE CONCOLIQUE DE BINAIRES AVEC ANGR

Julien VOISIN

L'analyse de binaires est un processus long et fastidieux, qu'il serait tellement plaisant d'automatiser, totalement ou en partie. C'est précisément le but d'angr : automatiser, exposer, simplifier, pour voir de quoi nos binaires sont faits !

1. ANGR, QU'EST-CE QUE C'EST ? D'OÙ ÇA VIENT ?

Angr, c'est un framework, écrit en Python, sous licence BSD, pour analyser des binaires, en combinant de l'analyse statique avec de l'exécution concolique.

Il existe une quantité invraisemblable d'outils pour « analyser des binaires » : de la reine IDA Pro pour la rétro-ingénierie, suivie de près par Binary Ninja et Hopper. Des débogueurs comme le vénérable Ollydbg et son successeur x64dbg, GDB pour les Unix, Windbg chez Microsoft... des frameworks comme radare2 et miasm pour les casse-cous, des désassembleurs comme capstone et objdump... bref, il y a de quoi faire.

Alors pourquoi angr ? Qu'a-t-il à offrir de plus ou de mieux ?

Il est principalement écrit par les gens du Computer Security Lab de l'UC Santa Barbara. C'est à cette université qu'est rattachée l'équipe des Shellphish, qui s'est classée 3e au Cyber Grand Challenge [1] de la DARPA, justement grâce (en partie) à angr. Le CGC était une compétition de systèmes autonomes, capables de découvrir, prouver et corriger des vulnérabilités applicatives. Le système vainqueur a joué contre des humains lors du CTF de la Defcon, et son équipe a empoché deux millions de dollars. Comme l'analyse de binaires est une tâche notoirement fastidieuse, angr a été pensé pour être facile à utiliser, souvent en quelques lignes de Python. Il est toutefois possible de trifouiller tout un tas de paramètres au besoin, mais la plupart du temps, les réglages par défaut suffisent largement.

1.1 À quoi ça sert ?

Étant un framework, Angr est capable de faire de beaucoup de choses. Comme chacun l'utilise à sa sauce, une liste à la Prévert d'exemples s'impose :

- ⇒ Résolution (semi-)automatique de crackme, typiquement en reversant un peu le binaire pour trouver ce qu'il attend comme format d'entrée, puis en demandant à angr de se débrouiller pour aller à l'offset indiquant que le flag est le bon, ou simplement en cherchant le mot `flag{` dans la sortie standard.
- ⇒ Exploration des états possibles d'un binaire pour, par exemple, trouver des vulnérabilités : est-il possible d'obtenir un déréférencement de pointeur nul ? Si oui, avec quelle entrée ? Le binaire comporte-t-il des appels à des fonctions dangereuses ? Si oui, peut-on contrôler leurs paramètres ?
- ⇒ Aide à la création de code d'exploitation : est-il possible de contrôler le pointeur d'instruction ? À quel endroit du binaire peut-on injecter un shellcode ? Quels sont les gadgets pour faire du ROP dans ce binaire ? Peut-on les associer pour obtenir une primitive de type *write-what-where* (on contrôle ce que l'on veut écrire et à quel endroit) ?

2. COMMENT ÇA MARCHE ?

Pour analyser un binaire, il faut a minima résoudre les problématiques suivantes :

- 1 Le chargement du binaire dans le moteur d'analyse ;
- 2 La transformation du binaire dans une représentation intermédiaire utilisable par ledit moteur d'analyse ;
- 3 L'analyse proprement dite, qui peut être :
 - ⇒ Une analyse statique (complète ou partielle) : analyse de dépendances, découpage en blocs logiques...
 - ⇒ Une exploration symbolique de l'espace formé par l'ensemble des états possibles du programme : quel chemin prendre pour obtenir un overflow ici, ou arriver dans tel bloc ?
 - ⇒ Une combinaison des deux méthodes précédentes : quels blocs est-il pertinent d'analyser pour savoir s'il est possible d'obtenir un overflow ici ?

Dans l'introduction, les mots « exécution concolique » sont présents. Il s'agit d'un mot valise, issu de « concrète » et « symbolique ». Contrairement à ce qui se passe dans un ordinateur, où chaque registre, chaque écriture/lecture mémoire possède une valeur concrète, dans angr, tout a une valeur symbolique. Quand l'exécution d'un programme rencontre une branche (par exemple, durant une condition), les deux chemins peuvent être analysés. Ce qui signifie, très grossièrement, que chacun des états du programme peut être représenté par des formules mathématiques : savoir comment accéder à tel ou tel état revient simplement à trouver des solutions à la formule qui lui correspond. Angr dispose d'heuristiques pour éliminer les states n'ayant pas d'impact sur l'analyse demandée, mais elles ont leurs limites : il n'est pas vraiment possible d'analyser de gros binaires comme `jscript.dll` (l'implémentation d'un des moteurs JavaScript de Windows).

NOTE

- ⇒ Bloc : ensemble d'instructions ne comportant pas de sauts ou de branches.
- ⇒ Path : représentation de haut niveau d'un ensemble de blocs exécutés depuis un état initial.
- ⇒ Path group : ensemble de *path* regroupés en tas.
- ⇒ State : un état au cours de l'exécution du binaire.
- ⇒ Stash : un ensemble d'états.
- ⇒ Hook : primitive permettant de détourner des fonctions natives.

Le reste de cet article utilisera cette terminologie, en anglais. Le sujet est assez compliqué à expliquer sans ajouter des traductions approximatives, et inutilisable en dehors de cet article, empêchant le lecteur d'approfondir le sujet ailleurs.

Pour installer angr, rien de bien sorcier : `pip install angr` (de préférence dans un virtualenv), avec `python-dev` et `libffi-dev` d'installés. Pour les gens qui ne sont pas sous un *nix, aucune idée des manœuvres à effectuer, désolé.

La première étape est de charger un binaire, ici `/bin/ls`. Dans angr, c'est sous la forme d'un *Projet* que ça se fait.

Terminal

```
>>> import angr
>>> projet = angr.Project('/bin/ls')
>>>
```

2.1 Le loader

Le passage de la représentation d'un binaire à son aspect une fois chargé en mémoire est, comme évoqué précédemment, un problème compliqué. Le composant s'acquittant du chargement des binaires dans angr s'appelle CLE, pour *CLE Loads Everything*. Il est possible de le manipuler en passant par l'attribut `loader` d'un projet. Cet attribut représente l'ensemble des objets chargés et mappés dans l'espace mémoire.

Il est possible d'obtenir facilement la liste de ces objets :

Terminal

```
>>> projet.loader.all_objects
[<ELF Object ls, maps [0x400000:0x620447]>,
 <ELF Object libselinux.so.1, maps [0x1000000:0x12278cf]>,
 <ELF Object libc- 2.24.so, maps [0x2000000:0x23c699f]>,
 <ELF Object libpcre.so.3.13.3, maps [0x3000000:0x3272107]>]
```

```
<ELF Object libdl-2.24.so, maps [0x4000000:0x42030ef]>,
<ELF Object ld-2.24.so, maps [0x5000000:0x5227167]>,
<ELF Object libpthread-2.24.so, maps [0x6000000:0x621d427]>,
<ELFTLSObject Object cle##tls, maps [0x7000000:0x700d010]>,
<ExternObject Object cle##externs, maps [0x8000000:0x8008000]>,
<KernelObject Object cle##kernel, maps [0x9000000:0x9008000]>
]
>>>
```

On voit notre binaire, **ls**, chargé en premier, puis les bibliothèques partagées, un objet **ELFTLSObject** (pour *thread local storage*, contenant, comme son nom l'indique, des informations propres à chaque thread), un objet **ExternObject** (pour *external*, fournissant les symboles non résolus) et un objet **KernelObject**, pour les syscalls.

Il est possible d'obtenir différentes informations au sujet de ces objets, comme par exemple, leurs offsets, tailles, contenus...

Terminal

```
>>> projet.loader.all_objects[0].segments
<Regions: [<ELFSegment offset=0x0, flags=0x5, filesize=0x1e024,
vaddr=0x400000, memsize=0x1e024>, <ELFSegment offset=0x1e048, flags=0x6,
filesize=0x1220, vaddr=0x61e048, memsize=0x2400>]>
>>> projet.loader.find_symbol('malloc')
<Symbol "malloc" in libc.so.6 at 0x1086930>
>>>
```

2.2 Blocs

Comme évoqué dans l'introduction, une partie de l'analyse statique vise à identifier correctement des blocs, afin de réduire la quantité d'états possibles à analyser de manière dynamique.

Il est possible d'obtenir et de manipuler un bloc à l'aide de la méthode **projet.factory.block(offset)** :

Terminal

```
>>> bloc = projet.factory.block(projet.entry)
>>> bloc.pp()
0x4052c0: xor ebp, ebp
0x4052c2: mov r9, rdx
0x4052c5: pop rsi
0x4052c6: mov rdx, rsp
0x4052c9: and rsp, 0xfffffffffffffff0
0x4052cd: push rax
0x4052ce: push rsp
0x4052cf: lea r8, qword ptr [rip + 0x107ca]
0x4052d6: lea rcx, qword ptr [rip + 0x10753]
0x4052dd: lea rdi, qword ptr [rip - 0x18a4]
0x4052e4: call qword ptr [rip + 0x219b1e]
>>> bloc.instruction_addrs
[4215488L, 4215490L, 4215493L, 4215494L, 4215497L, 4215501L, 4215502L,
4215503L, 4215510L, 4215517L, 4215524L]
>>>
```

Ici, on vient de demander à angr d'assigner à la variable **bloc** le bloc dans lequel est compris le point d'entrée du programme, d'afficher le désassemblage du bloc (**pp** comme **pretty print**), ainsi que les offsets des instructions.

On reconnaît ici effectivement un point d'entrée classique d'un binaire x64, avec la fonction **main** se situant vraisemblablement à **rip - 0x18a4**.

2.3 États

Dans angr, l'objet **Projet** ne représente que l'état du programme lors de son initialisation. Dans le contexte de l'analyse statique, on va manipuler des objets de type **SimState**, qui contient la mémoire du programme, ses registres, ses accès au système de fichiers..., tout ce qui peut être changé au cours de son exécution :

Terminal

```
>>> state = projet.factory.entry_state()
>>> state.regs.rip
<BV64 0x4052c0>
>>> hex(projet.entry)
'0x4052c0'
>>>
```

Jusqu'ici, tout semble bon. Le type utilisé pour contenir **rip**, le registre d'instruction courante, n'est pas un entier Python classique, mais un *bit vector*, qui se comporte comme un *word*. Par exemple, en cas de dépassement, un modulo 2^{32} (32 étant la taille d'un mot en x86.) va être appliqué, au lieu de s'entendre comme un entier Python : $0xffffffff + 1 = 0x0$.

2.4 Gestionnaire de simulation

Alors que les *states* d'angr permettent de représenter l'état d'un programme, les gestionnaires de simulation gèrent les relations qui existent entre ces derniers. C'est-à-dire comment passer d'un état à un autre, exécuter le programme...

Les *SimulationManager* permettent de contrôler l'exécution symbolique d'un groupe de *states* de manière simultanée, pour par exemple, appliquer des stratégies de recherche à l'échelle d'un programme entier afin d'explorer ses états possibles.

Afin de faciliter la manipulation des *states*, ceux-ci sont organisés en *stashes*, qu'il est possible de fusionner, diviser, filtrer, déplacer... à sa guise. Ce qui permet, par exemple, de *step* à des vitesses différentes dans deux stash différents, pour ensuite fusionner le résultat, afin d'obtenir un mélange de deux traces. La **stash** par défaut est appelée « active » ; c'est là que sont placées les *states* à leur initialisation par un *SimulationManager*.

Un petit exemple pour voir plus clair :

Terminal

```
>>> import angr
>>> proj = angr.Project('/bin/ls', auto_load_libs=False)
WARNING | 2017-12-29 14:54:02,071 | cle.loader | The main binary is a position-
independent executable. It is being loaded with a base address of 0x400000.
>>> state = proj.factory.entry_state()
>>> simgr = proj.factory.simgr(state)
>>> simgr.active
[<SimState @ 0x4052c0>]
>>> simgr.active[0].regs.rip
<BV64 0x4052c0>
>>> simgr.step()
<SimulationManager with 1 active>
>>> simgr.active
[<SimState @ 0x10001a0>]
>>> state.regs.rip
<BV64 0x4052c0>
>>> simgr.active[0].regs.rip
<BV64 0x10001a0>
>>>
```

Pour l'instant, rien de très palpitant : ce genre de choses peut très bien se faire dans GDB. Là où ça devient intéressant, c'est quand une branche est rencontrée :

Terminal

```
>>> proj = angr.Project('/bin/tar', auto_load_libs=False)
WARNING | 2017-12-29 15:10:08,529 | cle.loader | The main binary is a
position-independent executable. It is being loaded with a base address of
0x400000.[
0m
>>> state = proj.factory.entry_state()
>>> simgr = proj.factory.simgr(state)
>>> while len(simgr.active) == 1:
...     simgr.step()
...
<SimulationManager with 1 active>
[...]
<SimulationManager with 2 active>
>>> simgr
<SimulationManager with 2 active>
>>> simgr.active
[<SimState @ 0x43f7f4>, <SimState @ 0x43f7d6>]
>>> simgr.run()
WARNING | 2017-12-29 15:10:39,328 | angr.manager | No completion state
defined for SimulationManager; stepping until all states deadend
<SimulationManager with 2 deadended>
>>> simgr
<SimulationManager with 2 deadended>
>>>
```

Quand un *state* ne produit aucun successeur, par exemple, quand il rencontre le syscall `exit` ou une instruction invalide, il est retiré de la stash **active**, et est placé dans la stash **deadended**.

Angr vient avec un ensemble prédéfini de *stashes* :

- ⇒ **active** pour l'état par défaut des stashes, qui seront donc step quand aucune stash particulière n'est précisée ;
- ⇒ **deadended** pour les stashes dont la continuation d'exécution n'est plus possible ;
- ⇒ **pruned** pour, quand l'option **LAZY_SOLVES** est utilisée, les stashes dont la satisfaisabilité n'est vérifiée qu'en cas de nécessité. Par exemple, quand un state est déterminé comme insatisfaisable, l'arbre des états précédents est traversé, afin d'identifier quand ce *state* a été marqué ainsi. Puis tous les états descendants de ce point sont mis dans cette stash.
- ⇒ **unconstrained**, si l'option **save_unconstrained** est fournie en paramètre au constructeur du SimulationManager, les états qui ne sont pas contraints, comme par exemple quand l'utilisateur peut directement contrôler le pointeur d'instruction, sont placés ici.
- ⇒ **unsat**, dans laquelle sont placés les états insatisfaisables, si l'option **save_unsat** est passée au SimulationManager.

Il y a également le stash **errored**, qui sert surtout à déboguer, et comme angr ne comporte aucun bugs, nous n'en parlerons pas ;-)

Il peut parfois arriver qu'angr (et plus particulièrement la partie solveur) s'embourbe dans une boucle, et prenne un temps fou à mener à bien la phase d'analyse. Pour trouver où angr est bloqué, il est possible de lancer son script dans deux consoles ipython imbriquées, d'interrompre avec un `[ctrl]+[c]` l'exécution, pour se retrouver dans la première instance d'ipython, et inspecter la première, par exemple via `print pg.active`.

3. EXPLORATION

Une opération extrêmement courante d'utilisation d'exécution symbolique est de trouver un état aboutissant à une certaine adresse. C'est pourquoi angr fournit une méthode dédiée à cette tâche : `.explore()`.

Lors de l'utilisation d'`explore`, l'exécution s'arrête dès qu'un état correspondant à la condition passée en argument est valide, ce dernier étant ensuite placé dans la stash `found`.

La condition peut être n'importe quoi, comme par exemple la présence d'une chaîne de caractères dans le buffer de sortie (`find=lambda s: "Good flag!" in s.posix.dumps(1)`), une adresse mémoire, une instruction particulière...

Il est également possible de passer des conditions à éviter, de même manière qu'avec `find`, mais en utilisant le paramètre `avoid`. Si on veut plusieurs résultats, il suffit de jouer avec `num_find`.

3.1 Techniques d'exploration

Il est possible de modifier le comportement des `SimulationManager`, afin d'utiliser des techniques d'exploration différentes. Par exemple, la technique par défaut consiste à itérer en parallèle les états, ce qui revient à faire un parcours horizontal du graphe. Il est possible d'en définir d'autres, ou d'utiliser celles fournies de base par angr :

Terminal

```
>>> dir(angr.exploration_techniques)
['AngrError', 'AngrExplorationTechniqueError', 'CallFunctionGoal', 'DFS', 'Director',
 'ExecuteAddressGoal', 'ExplorationTechnique', 'Explorer', 'LengthLimiter', 'LoopLimiter',
 'Oppologist', 'SimError', 'Spiller', 'Threading', 'Veritesting', '__builtins__',
 '__doc__', '__file__', '__name__', '__package__', '__path__', 'dfs', 'director', 'explorer',
 'lengthlimiter', 'looplimiter', 'oppologist', 'spiller', 'threading',
 'veritesting']
>>>
```

Les plus utiles/simples étant :

- ⇒ **Explorer** : la technique utilisée par la méthode `explorer`, permettant de remplir et d'éviter des conditions.
- ⇒ **DFS** : parcours du graph en profondeur (*Depth First Search* en anglais).
- ⇒ **LoopLimiter** : utilise une approximation pour calculer le nombre d'itérations dans une boucle, afin de mettre de côté les états qui semblent boucler trop souvent, pour revenir dessus une fois que le nombre d'autres états viables est épuisé.
- ⇒ **LengthLimit** : fixe une limite de profondeur dans le parcours du graphe.
- ⇒ **ManualMergepoint** : marque une adresse dans un programme comme un point de fusion. Chaque parcours arrivant sur ce point sera fusionné en un seul.
- ⇒ **Veritesting** : basé sur un papier éponyme [2], il cherche automatiquement des points de fusion.
- ⇒ **Oppologist** : quand cette technique est utilisée, et qu'angr rencontre une instruction inconnue, comme par exemple `PCLMULQDQ`, il fait appel à unicorn [3] pour l'émuler, puis reprend ses analyses.
- ⇒ **Spiller** : permet de mettre en cache sur les disques certains des états actifs en cas de besoin. Pratique si l'analyse utilise une quantité faramineuse d'états.

4. MOTEURS D'EXÉCUTION

Pour avancer dans l'exécution d'états en états, angr utilise différents moteurs d'exécution, tous des sous-classes de SimEngine. Le principe est bien plus simple que pour les techniques d'exploration : les moteurs sont essayés les uns après les autres, tant que la transition n'a pas été correctement effectuée. L'ordre des moteurs est choisi en fonction de la situation.

Par exemple, quand un état aboutit dans un `syscall`, c'est le moteur de syscall qui va l'émuler. Quand un état n'a pas de successeur possible, il s'agit du moteur d'échec. Quand une adresse est interceptée, il s'agit du moteur de hook. Et ainsi de suite.

4.1 Hooks

Plutôt que d'émuler les fonctions classiques, comme celles de la libc, angr possède des routines internes pour les gérer plus efficacement. Cette astuce peut également s'utiliser sur les binaires compilés statiquement. Angr peut évidemment émuler complètement le binaire, mais il est possible de poser des *hooks* pour lui faciliter la vie, et donc réduire le temps d'attente pour avoir le flag ;-)

Par exemple, si on sait que telle fonction à l'adresse `0x414012` est un `memcpy`, on peut le signaler avec `p.hook(0x422690, angr.SIM_PROCEDURES['libc']['memcpy']())`. Angr vient avec un bon nombre de procédures classiques, qu'il est possible d'énumérer comme suit :

Terminal

```
>>> import angr
>>> import pprint
>>> pprint.pprint(angr.SIM_PROCEDURES)
>>>
```

Il est possible de définir ses propres fonctions, comme par exemple, `strcpy` dans cet exemple :

Fichier

```
class strcpy(angr.SimProcedure):
    def run(self, dst, src):
        self.argument_types = {0: self.ty_ptr(SimTypeString()),
                                1: self.ty_ptr(SimTypeString())}
        self.return_type = self.ty_ptr(SimTypeString())

        strlen = angr.SIM_PROCEDURES['libc']['strlen']
        strncpy = angr.SIM_PROCEDURES['libc']['strncpy']
        src_len = self.inline_call(strlen, src)

        ret_expr = self.inline_call(strncpy, dst, src, src_len.ret_expr+1,
                                     src_len.ret_expr).ret_expr
        return ret_expr
```

Les arguments sont de type pointeur sur `SimTypeString`, tout comme la valeur de retour. Le reste de l'implémentation appelle `strncpy`, avec la longueur du paramètre `src`, et retourne la valeur de retour de celle-ci.

Angr va se débrouiller pour détecter les conventions d'appel, et automatiquement assigner les paramètres passés à la fonction dans le binaire aux paramètres de la méthode `run`, ce qui, en plus d'être impressionnant, est réellement pratique ! Il en va évidemment de même pour la valeur de retour de la fonction.

Il est également possible de poser des points d'interception sur autre chose que des fonctions, par exemple, pour modifier à l'offset **0x1234** la valeur du registre **rax**, puis reprendre l'exécution 5 octets plus loin :

Fichier

```
>>> @project.hook(0x1234, length=5)
... def set_rax(state):
...     state.regs.rax = 1
```

Angr résolvant les imports, il est possible de les intercepter directement, par exemple, pour rendre la fonction **rand** un peu moins aléatoire :

Fichier

```
>>> class NotVeryRand(SimProcedure):
...     def run(self):
...         return 1337

>>> project.hook_symbol('rand')
```

5. EXEMPLE

La théorie c'est bien, la pratique c'est mieux !

5.1 Exemple avec un petit crackme

Le programme suivant est tout bête : il attend une chaîne de 23 caractères de long, vérifie qu'ils soient en majuscules, et qu'ils remplissent bien certaines relations. Si toutes les conditions sont validées, le programme affiche « *Congratz* », suivi de notre entrée.

Fichier

```
// angr_test1.c

#include <stdio.h>
#include <unistd.h>

int main(int argc, char** arg) {
    char buf[24] = {0};

    if (23 != read (0, buf, 23)) {
        puts("nupe\n");
        return -1;
    }

    // only printable chars
    for (int i = 0; buf[i]; i++) {
        if ((buf[i] < 0x41) || (0x5a < buf[i])) {
            printf("nape: %c:\n", buf[i]);
            return -1;
        }
    }

    for (int i = 0; i < 22; ++i) {
        if (buf[i] != (buf[i+1] ^ 0x2) - 1) {
            puts("nope\n");
            return -1;
        }
    }
}
```

```

    }
    }
    printf("Congratz: flag{%s}", buf);
    return 0;
}

```

Pour résoudre ce crackme, on va donner un petit coup de main à angr, en partant du principe que l'entrée utilisateur n'est pas plus grande que 64 caractères, et qu'ils sont ASCII et imprimables (c'est-à-dire, compris entre `!` et `~`). Si le binaire attend une clef de moins de 64 caractères, ça n'est pas très grave, angr peut très bien ne pas remplir tout le buffer.

On lance ensuite une exploration à partir de notre état d'entrée, puis on cherche la chaîne `flag{` dans le buffer de sortie :

Fichier

```

import angr
import sys

b = angr.Project(sys.argv[1])
state = b.factory.blank_state()

for _ in range(64):
    c = state.posix.files[0].read_from(1)
    state.solver.add(c >= '!')
    state.solver.add(c <= '~')

state.posix.files[0].seek(0)
state.posix.files[0].length = 64

pg = b.factory.simulation_manager(state)
pg.explore()

for pp in pg.deadended:
    out = pp.posix.dumps(1)
    if 'flag{' in out:
        print out

```

Terminal

```

$ python solve_angr_test1.py angr_test1
WARNING | 2018-01-07 17:43:29,837 | cle.loader | The main binary is a
position-independent executable. It is being loaded with a base address of
0x400000.
WARNING | 2018-01-07 17:43:30,374 | angr.manager | No completion state
defined for SimulationManager; stepping until all states deadend
Congratz: flag{CFEDGJIHKNMLORQPSVUTWZY}
./angr_test1
CFEDGJIHKNMLORQPSVUTWZY
Congratz: flag{CFEDGJIHKNMLORQPSVUTWZY}

```

5.2 Exemple de détection de contrôle de l'eip

Souvent (mais pas toujours), l'exploitation d'une vulnérabilité due à une corruption mémoire passe par un contrôle du pointeur d'instruction. Angr est parfaitement capable de nous aider à trouver ce genre de choses. Après tout, si on y réfléchit, contrôler ce pointeur, ça revient à trouver un état dans lequel celui-ci n'est pas contraint.

Ce qui nous donne quelque chose comme ça :

MISC HORS-SÉRIE N°17 : RECHERCHE DE VULNÉRABILITÉS

```

EAX: 0x91
EBX: 0x0
ECX: 0xffffd290 --> 0x0
EDX: 0x100
ESI: 0x1
EDI: 0xf7fa7000 --> 0x1b5db0
EBP: 0x0
ESP: 0xffffd320 --> 0xf7fa730a --> 0xffff0000 --> 0x0
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10207 (CARRY PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x41414141
[-----stack-----]
0000| 0xffffd320 --> 0xf7fa730a --> 0xffff0000 --> 0x0
0004| 0xffffd324 --> 0x80481e8 --> 0x2e ('.')
0008| 0xffffd328 --> 0x804846b (add    ebx,0x1199)
0012| 0xffffd32c --> 0x0
0016| 0xffffd330 --> 0x1
0020| 0xffffd334 --> 0xf7fa7000 --> 0x1b5db0
0024| 0xffffd338 --> 0x0
0028| 0xffffd33c --> 0xf7e09276 (<__libc_start_main+246>: add    esp,0x10)
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$

```

CONCLUSION

En fin de compte, on peut voir angr comme un émulateur complètement instrumentable. Bien sûr, il vient avec tout un tas d'outils et de couches d'abstractions, d'analyse, d'inférences, de résolutions... mais tout ça n'est qu'un bel emballage. Il n'existe pas de problèmes qui ne soient pas solubles en appliquant des grands coups de `.step` dans un `SimState`.

REMERCIEMENTS

Merci à Julien Bachmann pour la relecture et l'astuce des instance d'ipython imbriquées. ■

RÉFÉRENCES

- [1] <http://archive.darpa.mil/cybergrandchallenge/>
- [2] https://users.ece.cmu.edu/~dbrumley/pdf/Avgerinos%20et%20al._2014_Enhancing%20Symbolic%20Execution%20with%20Veritesting.pdf
- [3] <http://www.unicorn-engine.org/>

ANALYSE STATIQUE & DYNAMIQUE

RADARE2 / ANALYSE PAR TEINTE / PLUGIN / PYTHON

INTRODUCTION AU DÉVELOPPEMENT DE PLUGINS POUR RADARE2

Ayman KHAMOUMA

Cet article a pour but d'introduire le lecteur au développement de plugins pour radare2 afin d'automatiser certaines tâches de rétro-ingénierie ou de recherche de vulnérabilités.

1. UNE INTRODUCTION AU SCRIPTING DE RADARE2

1.1 Installation

Étant donné que radare2 est en perpétuelle évolution, il est recommandé de l'installer directement depuis le dépôt git :

Terminal

```
$ git clone https://github.com/radare/radare2.git
```

La compilation et l'installation se font ensuite via le script `sys/install.sh`.

Afin de simplifier le processus de mise à jour, ce script n'installera que des liens symboliques vers les binaires générés sur le système de fichier cible. Ainsi, il suffira d'exécuter les commandes suivantes afin de mettre à jour une installation :

Terminal

```
$ git pull origin master:master
$ make
```

1.2 L'aide de radare2

Comme tout outil de reverse engineering digne de ce nom, radare2 (parfois nommé simplement r2) propose un nombre incalculable de commandes (désassemblage, analyse, débogage, etc.). Toutes sont documentées et accessibles depuis la ligne de commandes.

La commande `?` nous permet d'y accéder :

Terminal

```
[0x00000000]> ?
Usage: [.] [times] [cmd] [~grep] [[@@iter] addr!size] [!>pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
| %var =valueAlias for 'env' command
| * [?] off [=0x]value      Pointer read/write data/values (see ?v, wx, wv)
| (macro arg0 arg1)         Manage scripting macros
| . [?] [-] (m) | f | !sh | cmd Define macro or load r2, cparse or rlang file
| = [?] [cmd]               Send/Listen for Remote Commands (rap://, http://, <fd>)
| / [?]                    Search for bytes, regexps, patterns, ..
| ! [?] [cmd]              Run given command as in system(3)
| # [?] !lang [...]        Hashbang to run an rlang script
| a [?]                    Analysis commands
| b [?]                    Display or change the block size
| c [?] [arg]              Compare block with given data
| C [?]                    Code metadata (comments, format, hints, ..)
| d [?]                    Debugger commands
| e [?] [a [=b]]           List/get/set config evaluable vars
| f [?] [name] [sz] [at]   Add flag at current address
| g [?] [arg]              Generate shellcodes with r_egg
| i [?] [file]             Get info about opened file from r_bin
| k [?] [sdb-query]        Run sdb-query. see k? for help, 'k *', 'k **' ...
| L [?] [-] [plugin]       list, unload load r2 plugins
| m [?]                    Mountpoints commands
| o [?] [file] ([offset])  Open file at optional address
| p [?] [len]              Print current block with format and length
```



```

| P[?]          Project management utilities
| q[?] [ret]    Quit program with a return value
| r[?] [len]    Resize file
| s[?] [addr]   Seek to address (also for '0x', '0x1' == 's 0x1')
| S[?]          Io section manipulation information
| t[?]          Types, noreturn, signatures, C parser and more
| T[?] [-] [num|msg] Text log utility
| u[?]          uname/undo seek/write
| V            Visual mode (V! = panels, VV = fcngraph, VVV = callgraph)
| w[?] [str]    Multiple write operations
| x[?] [len]    Alias for 'px' (print hexadecimal)
| y[?] [len] [[[@]addr] Yank/paste bytes from/to memory
| z[?]          Signatures management
| ?[??][expr]  Help or evaluate math expression
| ??          Show available '$' variables and aliases
| ?@          Misc help for '@' (seek), '~' (grep) (see ~??)

```

Les commandes de r2 sont classifiées en plusieurs sous-types, la première lettre de la commande permet de définir sa catégorie : **d** comme debug, **p** comme print, **w** comme write, et ainsi de suite.

Il est possible d'obtenir plus de détails au sujet d'une commande en la suffixant d'un **?**.

Ainsi la commande **d?** nous permettra d'obtenir plus d'informations sur les commandes de type debug.

Terminal

```

[0x00000000]> d?
|Usage: d # Debug commands
| db[?]          Breakpoints commands
| dbt[?]         Display backtrace based on dbg.btdepth and dbg.btalgo
| dc[?]          Continue execution
| dd[?]          File descriptors (!fd in r1)
| de[-sc] [rwx] [rm] [e] Debug with ESIL (see de?)
| dg <file>      Generate a core-file (WIP)
| dH [handler]   Transplant process to a new handler
| di[?]          Show debugger backend information (See dh)
| dk[?]          List, send, get, set, signal handlers of child
| dL [handler]   List or set debugger handler
| dm[?]          Show memory maps
| do[?]          Open process (reload, alias for 'oo')
| doo[args]      Reopen in debugger mode with args (alias for 'ood')
| dp[?]          List, attach to process or thread id
| dr[?]          Cpu registers
| ds[?]          Step, over, source line
| dt[?]          Display instruction traces (dtr=reset)
| dw <pid>       Block prompt until pid dies
| dx[?]          Inject and run code on target process (See gs)

```

De la même manière, on peut ainsi récupérer encore plus de détails au sujet des sous-commandes associées.

Il est aussi possible de demander à radare2 de fournir récursivement toute l'aide ou uniquement celle correspondant à une sous-commande donnée, via l'opérateur **?*** :

⇒ **?*** permettra de récupérer l'intégralité de l'aide ;

⇒ **d?*** permettra de récupérer toute l'aide des sous-commandes de **d**.

Radare2 fournit aussi un grep interne (**~**) afin de filtrer la sortie d'une commande. On pourra par exemple l'utiliser entre autres pour faire de la recherche dans l'aide de radare2.

Ainsi si on souhaite chercher les commandes liées aux gadgets ROP, par exemple :

Terminal

```
[0x7feb2f4827d1]> ?*~+gadgets
| /R [grepopcode]      search for matching ROP gadgets, semicolon-separated
| Usage: /R Search for ROP gadgets
| /R [filter-by-string] Show gadgets
| /R/ [filter-by-regexp] Show gadgets [regular expression]
| /Rl [filter-by-string] Show gadgets in a linear manner
| /R/l [filter-by-regexp] Show gadgets in a linear manner [regular expression]
| /Rk [select-by-class] Query stored ROP gadgets
| Usage: /Rk Query stored ROP gadgets
| /Rk [nop|mov|const|arithm|arithm_ct] Show gadgets
| /Rkq List Gadgets offsets
```

Plus d'informations au sujet du grep interne de radare2 sont disponibles via la commande `~?`.

Il est également possible d'utiliser le caractère pipe (`|`) afin de rediriger la sortie standard d'une commande vers n'importe quelle commande shell. Avec les chevrons (`>` ou `>>`), il est possible de rediriger vers un fichier.

1.3 Commandes de base

Dans cette partie, nous allons aborder les commandes de base de r2. Ces commandes pourront aussi bien être utiles pour une analyse statique et manuelle que dans le cadre de développement d'outils.

1.3.1 Information sur le binaire

Lors de l'analyse d'un binaire, nous serons amenés à récupérer un certain nombre de métadonnées telles que l'architecture utilisée, le nombre de bits, la liste des imports, la liste des relocations, etc. Toutes ces informations sont disponibles via la commande `i`.

On pourra par exemple récupérer l'adresse du point d'entrée du binaire grâce à la commande `ie` :

Terminal

```
[0x00402c59]> ie
[Entrypoints]
vaddr=0x004049a0 paddr=0x000049a0 baddr=0x00400000 laddr=0x00000000 haddr=0x00000018
type=program

1 entrypoints
```

Ou encore afficher ses informations générales avec la commande `iI` :

Terminal

```
[0x00402c59]> iI
arch      x86
binsz     124726
bintype    elf
bits      64
canary     true
class      ELF64
crypto     false
endian     little
havecode   true
intrp      /lib64/ld-linux-x86-64.so.2
lang       c
```



```

linenum  false
lsyms     false
machine  AMD x86-64 architecture
maxopsz   16
minopsz   1
nx        true
os        linux
pcalign   0
pic        false
relocs    false
relro     partial
rpath     NONE
static     false
stripped  true
subsys    linux
va        true

```

La commande **i** permet également d'obtenir des informations liées aux symboles utilisés par le binaire, en effet, les commandes **iE**, **ii**, **is**, **iS** et **ir** permettront respectivement d'afficher la liste des exports, imports, symboles, sections et relocations du binaire en cours d'analyse.

1.3.2 La commande seek

Une des premières commandes à connaître de radare2 est la commande seek (**s**), cette commande va nous permettre de nous déplacer dans le fichier aux adresses/symboles de notre choix.

Terminal

```

[0x00400430]> s main
[0x00400526]>

```

En plus des paramètres de base (registres, flag, valeur), **s** accepte aussi bien des opérations mathématiques que des accès mémoire :

Terminal

```

$ r2 -d /bin/ls .bashrc
[0x7f4f9ecc1ed0]> dcu main
[0x564b8fec2c10]> dr rsi
0x7fff3c570948
[0x564b8fec2c10]> s [rsi + 8]
[0x7fff3c572331]>

```

Il est intéressant de savoir que radare2 propose aussi un moyen de se déplacer de manière temporaire avant d'exécuter une commande. Il suffit de préfixer les commandes d'un **@** suivi d'une adresse, d'un flag, d'une expression mathématique ou encore un d'accès mémoire.

1.3.3 Les flags

Les flags de radare2 peuvent être perçus comme un système de marque-page, ils permettent d'associer une chaîne de caractères (un *flag*) à un offset.

Il est possible de regrouper des ensembles de flags dans des groupes de flags appelés « flagospace ».

La commande **f** va nous permettre de les manipuler.

Sans argument, **f** permet de lister l'ensemble des flags ou des flagspaces sélectionnés.

Il est possible de créer un flag à l'adresse de son choix via la commande suivante :

Terminal

```
[0x00000000]> f mon_flag @ 0x1000
[0x00000000]> f~mon_flag
0x00001000 1 mon_flag
```

Les flags générés par r2 sont automatiquement rangés dans différents flagspaces. Il est possible de récupérer la liste de tous les flagspaces via **fs** :

Terminal

```
[0x00400526]> fs
0 2 * strings
1 33 * symbols
2 82 * sections
3 3 * relocs
4 3 * imports
```

Par défaut, tout les flagspaces sont sélectionnés. Afin de sélectionner un flagspace spécifique, il suffit de le donner en argument à **fs**.

fs permet aussi de créer ses propres flagspaces, il suffit de sélectionner un flagspace inexistant pour qu'il soit automatiquement créé et sélectionné.

Terminal

```
[0x00400526]> fs test
[0x00400526]> fs
0 2 . strings
1 33 . symbols
2 82 . sections
3 3 . relocs
4 3 . imports
5 0 * test
```

1.3.4 Interprétation des données en mémoire

Les sous-commandes de **p** vont nous permettre d'afficher et d'interpréter certaines informations.

On peut par exemple utiliser la commande **pd** (*print disassembly*) pour désassembler un certain nombre d'instructions :

Terminal

```
[0x00400527]> pd 4
0x00400527 4889e5 mov rbp, rsp
0x0040052a 4883ec10 sub rsp, 0x10
0x0040052e 897dfc mov dword [rbp - 4], edi
0x00400531 488975f0 mov qword [rbp - 0x10], rsi
```

Il est également possible d'afficher le contenu d'une adresse sous différents formats.

On peut par exemple utiliser les sous-commandes de **px** (*print hex*) afin de présenter l'affichage dans un format hexadécimal.

Terminal

```
0x00402c59> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00402c59 4531 ed45 31e4 904c 8d44 2430 b9a0 4d41 E1.E1..L.D$0..MA
0x00402c69 00ba 3889 4100 4889 ee89 dfc7 4424 30ff ..8.A.H.....D$0.
0x00402c79 ffff ffe8 9ff8 ffff 83f8 ff0f 84eb 0600 .....
0x00402c89 0005 8300 0000 3d12 0100 000f 8784 0600 .....=.....
```


Certaines commandes (comme ici **px**) acceptent une taille en paramètre, si on ne leur en fournit pas, radare va afficher/interpréter tout le contenu d'un bloc (256 octets par défaut). Il est possible de modifier la taille d'un bloc à l'aide de la commande **b** :

Terminal

```
[0x00402c59]> b 32
[0x00402c59]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00402c59 4531 ed45 31e4 904c 8d44 2430 b9a0 4d41 E1.E1..L.D$0..MA
0x00402c69 00ba 3889 4100 4889 ee89 dfc7 4424 30ff ..8.A.H....D$0.
[0x00402c59]> b 16
[0x00402c59]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D 0123456789ABCD
0x00402c59 4531 ed45 31e4 904c 8d44 2430 b9a0 4d41 E1.E1..L.D$0..MA
```

Les commandes **px** sont pratiques pour une analyse visuelle, mais dans le cadre du développement de plugins pour r2, nous serons plus amenés à récupérer des informations brutes (**pr**).

Il existe un nombre important de variantes de la commande **p**, je vous invite donc à en voir la liste complète via **p?**.

1.3.5 Les opérations d'écriture

La commande **w** va nous permettre aussi bien d'écrire en mémoire lorsque nous utiliserons le mode debug, que de modifier, de manière temporaire ou permanente, le fichier en cours d'analyse.

Par défaut, le binaire à analyser est ouvert en lecture seule, pour pouvoir le modifier, il faudra l'ouvrir à nouveau, en lecture/écriture, à l'aide de la commande **oo** :

Terminal

```
[0x00400430]> oo?
|Usage: oo[-] [arg] # map opened files
| oo      reopen current file
| oo+     reopen in read-write
| oob     reopen loading rbin info
| ood     reopen in debug mode
| oom     reopen in malloc://
| oon     reopen without loading rbin info
| oon+    reopen in read-write mode without loading rbin info
| oonn    reopen without loading rbin info, but with header flags
| oonn+   reopen in read-write mode without loading rbin info, but with
[0x00400430]> oo+
[0x00400430]> px 1
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00400430 31                                     1
[0x00400430]> wx 90
[0x00400430]> px 1
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00400430 90                                     .
[0x00400430]>
```

En lecture seule, il sera également possible d'activer un cache d'entrée/sortie qui permettra à la session en cours de procéder à des modifications sans altérer le fichier grâce l'option **io.cache** :

Terminal

```
[0x00400430]> e io.cache=true
```

La liste de toutes les options disponibles et de leurs descriptions est disponible via la commande **e??**.

En activant le cache, il est également possible d'obtenir la liste des modifications apportées grâce à la commande **wc** :

Terminal

```
[0x00400430]> wc
idx=0 addr=0x00400430 size=1 31 -> 90 (not written)
```

Ces modifications ne seront ensuite effectivement appliquées sur le fichier ou la mémoire du process débogué uniquement sur demande de l'utilisateur. La commande **wci** permet d'appliquer toutes les modifications apportées au domaine d'entrée-sortie utilisé (l'espace mémoire du process débogué en mode debug, le fichier en mode écriture, autres plugins d'io).

Il est également possible d'appliquer partiellement des modifications, ou encore d'annuler une modification locale à l'aide des commandes **wc+** et **wc-**.

De la même façon que les sous-commandes **p**, les sous-commandes **w** permettent d'écrire différents types de données en mémoire. On pourra par exemple écrire directement une valeur hexadécimale, ou du code assembleur, que r2 assemblera lui-même, avant écriture :

Terminal

```
[0x00400526]> wx 90
[0x00400526]> pd 1
      ;-- main:
      ;-- main:
0x00400526      90      nop
[0x00400526]> wa ret
Written 1 byte(s) (ret) = wx c3
[0x00400526]> pd 1
      ;-- main:
      ;-- main:
0x00400526      c3      ret
```

La commande **w** est tout aussi complète que la commande **p**, je vous invite à plonger dans sa documentation via **w?**.

1.3.6 Le mode debug

Nous avons rapidement vu le mode debug dans la section précédente, c'est donc l'occasion d'aborder plus en détail ses fonctionnalités.

Il est possible de lancer r2 directement en mode debug grâce au flag **-d** :

Terminal

```
$ r2 -d /bin/ls
```

Ou alors de demander à r2 de rouvrir le fichier courant en mode debug grâce à la commande **ood**.

Par défaut, radare2 nous donne la main au point d'entrée du binaire, juste après qu'il soit chargé en mémoire. Il est possible de changer ce comportement avec l'option **dbg.bep**. Les valeurs possibles sont **loader**, **main**, **entry** ou n'importe quelle adresse/symbole.

Il est aussi possible de récupérer la liste des constructeurs/destructeurs grâce à la commande **iee**.

1.3.6.1 Reprise d'exécution

Le mode debug de radare2 propose un certain nombre de commandes intéressantes. Il va par exemple nous permettre non seulement de continuer l'exécution avec **dc**, mais on peut également lui demander de continuer jusqu'à une adresse particulière avec **dcu** :

Terminal

```
[0x7f31e6f55c30]> dcu main
Continue until 0x00400566 using 1 bpsize
hit breakpoint at: 400566
[0x00400566]>
```

ou encore continuer jusqu'au prochain appel système via **dcs** :

Terminal

```
[0x00400566]> dcs write
Running child until syscalls:1
child stopped with signal 133
--> SN 0x7f755e78bc34 syscall 5 fstat (0x1 0x7ffde3c2ded0)
child stopped with signal 133
--> SN 0x7f755e791e19 syscall 12 brk (0x0)
child stopped with signal 133
--> SN 0x7f755e791e19 syscall 12 brk (0x193f000)
child stopped with signal 133
--> SN 0x7f755e78c2c0 syscall 1 write (0x1 0x191e010 0x5)
[0x7f755e78c2c0]>
```

De la même façon, **dcf**, **dcc**, **dcu**, **dcp** permettent respectivement de continuer l'exécution jusqu'à un fork, un appel de fonction, un appel indirect de fonction, l'entrée de la section du code du binaire. La sous-commande **dc** comporte un grand nombre de sous-commandes, il est vivement recommandé de consulter sa documentation via **dc?**.

1.3.6.2 Les points d'arrêt

Il est possible de poser des points d'arrêt logiciel avec la commande **db** :

Terminal

```
[0x7f757c25bc30]> db main
[0x7f757c25bc30]> dc
hit breakpoint at: 400566
[0x00400566]>
```

Ou encore des points d'arrêt matériels/watchpoints via la commande **drx** :

Terminal

```
[0x7f319ffa1c30]> drx 0 main 1 x
[0x7f319ffa1c30]> dc
[0x00400566]>
```

1.3.6.3 Exécution pas à pas

Radare2 nous fournit aussi un ensemble de commandes permettant de faire du débogage pas à pas et de tracer l'exécution du code.

Les commandes **ds** et **dso** vont respectivement permettre de *step-in* et *step-over*. D'autres commandes **ds** sont disponibles, en voici un bref aperçu :

⇒ **dsu** : permet de tracer le code jusqu'à un symbole ou une adresse ;

- ⇒ **dsui** : permet de tracer le code jusqu'à une instruction ;
- ⇒ **dsf** : permet de tracer le code jusqu'à une détection de fin de *stack frame*.

1.3.6.4 Le système de trace

Par défaut, r2 n'enregistre pas les traces d'exécution (*stepping*). Afin de permettre à r2 d'enregistrer une trace, il suffit d'activer l'option **dbg.trace** :

Terminal

```
[0x00400566]> e dbg.trace = true
[0x00400566]> dsui jmp rax
0x00400567 3 mov rbp, rsp
0x0040056a 4 sub rsp, 0x10
0x0040056e 3 mov dword [rbp - 4], edi
0x00400571 4 mov qword [rbp - 0x10], rsi
0x00400575 4 cmp dword [rbp - 4], 0x3b
0x00400579 2 ja 0x4005c2
0x0040057b 3 mov eax, dword [rbp - 4]
0x0040057e 8 mov rax, qword [rax*8 + 0x400698]
0x00400586 2 jmp rax
Stop.
```

Il est possible ensuite de récupérer la trace via la commande **dt** :

Terminal

```
[0x00400566]> dt
0x00400567 size=3 count=1 times=1 tag=0
0x0040056a size=3 count=2 times=1 tag=0
0x0040056e size=4 count=3 times=1 tag=0
0x00400571 size=4 count=4 times=1 tag=0
0x00400575 size=2 count=5 times=1 tag=0
0x00400579 size=3 count=6 times=1 tag=0
0x0040057b size=8 count=7 times=1 tag=0
0x0040057e size=2 count=8 times=1 tag=0
```

La valeur **count** donne le numéro de l'instruction, la valeur **times** nous donne le nombre de fois où une instruction a été atteinte.

1.3.7 Analyse de code

Tout comme IDA, r2 est capable de procéder à l'analyse automatique d'un binaire. Cela étant dit, le workflow r2 est différent de celui proposé par IDA, en effet les développeurs de radare2 déconseillent de procéder à l'analyse complète de l'ensemble d'un binaire à son ouverture.

À la place, r2 propose un éventail de commandes permettant, au fur et à mesure, de récupérer de nombreuses informations concernant les instructions, *basic blocks*, fonctions, graphe de flot de contrôle (CFG), etc.

La commande **af** par exemple, permettra d'analyser les basic blocks d'une fonction, d'en extraire son CFG, et de renommer automatiquement les variables locales.

Terminal

```
[0x004004d6]> af
[0x004004d6]> pdf
;-- main:
/ (fcn) main 31
|   main (int arg_1h);
|       ; var int local_10h @ rbp-0x10
```



```

|      ; var int local_4h @ rbp-0x4
|      ; arg int arg_1h @ rbp+0x1
|      0x004004d6      55          push rbp
|      0x004004d7      4889e5      mov rbp, rsp
|      0x004004da      897dfc      mov dword [local_4h], edi
|      0x004004dd      488975f0     mov qword [local_10h], rsi
|      0x004004e1      837dfc01     cmp dword [local_4h], 1      ; [0x1:4]=-1 ; 1
|      ,=< 0x004004e5      7f07          jg 0x4004ee
|      | 0x004004e7      b8ffffff      mov eax, 0xffffffff      ; -1
|      ,==< 0x004004ec      eb05          jmp 0x4004f3
|      |'-> 0x004004ee      b800000000     mov eax, 0
|      |      ; JMP XREF from 0x004004ec (main)
|      |'-> 0x004004f3      5d          pop rbp
|      \ 0x004004f4      c3          ret

```

Vous remarquerez qu'afin d'afficher le code de notre fonction, nous avons utilisé la commande **pdf**, cette commande utilise les informations de l'analyse faite par r2 pour n'afficher que le code de la fonction analysée.

La commande **afb** va nous permettre à partir de cette analyse de récupérer la liste des basic blocks de cette fonction :

Terminal

```

[0x004004d6]> afb
0x004004d6 0x004004e7 00:0000 17 j 0x004004ee f 0x004004e7
0x004004e7 0x004004ee 00:0000 7 j 0x004004f3
0x004004ee 0x004004f3 00:0000 5 j 0x004004f3
0x004004f3 0x004004f5 00:0000 2

```

La première colonne correspond à l'adresse du basic block, la seconde à sa fin, la troisième aux informations de trace (**count**, **times**), la quatrième au nombre d'instructions, la cinquième à l'adresse du saut en cas de validation de condition, et enfin la dernière à l'adresse du saut si la condition n'est pas validée.

Il est possible de modifier ces informations de basic block via les commandes **afb+** et **afbe**.

Cela peut être utile lorsque l'analyse automatique de r2 ne suffit pas, on pourra ainsi reconstruire par exemple les différentes branches d'une *switch table*.

Radare2 nous permet aussi de récupérer de précieuses informations concernant les instructions, en effet, les commandes **ao** vont nous permettre aussi bien d'obtenir des informations basiques (**ao**), telles que le type d'une instruction, son mnémonique, les différents registres utilisés, etc.

Mais aussi la traduction de l'instruction en un langage intermédiaire (REIL), facilement analysable (**aor**).

Terminal

```

[0x004004d6]> pd 1
0x004004d6      55          push rbp

[0x004004d6]> ao
address: 0x4004d6
opcode: push rbp
mnemonic: push
prefix: 0
id: 588
bytes: 55

```

```

refptr: 0
size: 1
type: upush
esil: rbp,8,rsp, -=,rsp,[8]
stack: inc
stackop: -320062632
stackptr: 8

[0x004004d6]> aor
0x4004d6
0000.00: SUB R_rsp:64 , 8:64 , V_00:64
0000.01: STR R_rsp:64 , , V_01:64
0000.02: STR V_00:64 , , R_rsp:64
0001.00: LDM R_rsp:64 , , V_02:64
0001.01: STM R_rbp:64 , , R_rsp:64
[0x004004d6]>

```

NOTE

En interne, radare2 utilise un langage intermédiaire appelé ESIL, ce langage ayant été choisi historiquement pour de l'émulation (sujet que nous ne couvrirons pas dans cet article), il trouve rapidement ses limites en termes d'analyse.

Le langage REIL permet de découper chaque instruction analysée en groupe de micro-instructions facilement analysables. Chacune de ces instructions est au format **mnemonique src1, src2, dest**. **src1** et **src2** étant soit une référence, soit un registre soit une valeur immédiate, **dest** sera soit une référence, soit un registre.

REIL ne propose que deux instructions pour l'accès mémoire : **STM** et **LDM**. Ces instructions ne prennent que deux opérandes, une source et une destination.

Les branchements (conditionnels ou non) sont tous gérés par l'instruction **JCC** qui est au format **mnemonique condition, target**, où **condition** et **target** peuvent être une valeur immédiate, un registre ou une référence.

1.4 Le développement de plugins/scripts sous r2

Le développement de plugins pour r2 peut se faire de différentes façons en utilisant différents langages.

En effet, r2 propose plusieurs manières de développer des scripts et plugins :

- ⇒ r2pipe pour le *simple scripting* ;
- ⇒ les bindings lang-{python, rust, js, etc.} couplés avec r2pipe pour le développement de plugins ;
- ⇒ C en liant avec les bibliothèques r2.

Dans cette section, nous allons nous intéresser à la deuxième solution : lang-python et r2pipe.

r2pipe est un module minimaliste (ici en Python, mais il est disponible dans un grand nombre de langages) : il permet de contrôler une session r2 en offrant à l'utilisateur la possibilité d'appeler des commandes r2 via la méthode **cmd**.

Terminal

```
>>> import r2pipe
>>> r2 = r2pipe.open("/bin/ls")
>>> r2.cmd("p8 10")
'31ed4989d15e4889e248'
```

Il est possible d'installer r2pipe depuis pip, mais étant donné que radare2 est un projet en constante évolution, il est recommandé, comme évoqué dans l'introduction, de toujours récupérer la dernière version depuis GitHub.

On peut également appeler un script r2pipe directement depuis r2 :

Terminal

```
[0x004049a0]> #!pipe python mon_script.py
```

Lorsque r2pipe est invoqué depuis r2, il est permis d'omettre le nom du fichier :

Terminal

```
[0x004049a0]> #!pipe python
>>> import r2pipe
>>> r2 = r2pipe.open()
>>> r2.cmd("p8 10")
'31ed4989d15e4889e248'
```

r2 va créer un canal de communication bidirectionnel entre le script et r2 à l'aide de deux pipes.

En plus de la méthode `cmd`, r2pipe propose une autre façon très pratique de récupérer des informations. En effet, un grand nombre de commandes r2 accepte le suffixe `j`. Ces commandes vont alors générer des données au format json.

Terminal

```
[0x004049a0]> iej~{
[
  {
    "vaddr": 4213152,
    "paddr": 18848,
    "baddr": 4194304,
    "laddr": 0,
    "haddr": 24,
    "type": "program"
  }
]
```

NOTE

`~{}` est la fonction pretty print du grep interne de r2.

r2pipe est ensuite capable de les convertir en objets python grâce à `cmdj` :

Terminal

```
>>> import r2pipe
>>> r2 = r2pipe.open("/bin/ls")
>>> entrypoint_data = r2.cmdj("iej")
>>> print entrypoint_data[0]["type"]
program
```

Cette méthode trouve rapidement ses limites lorsqu'on veut développer un plugin relativement complexe nécessitant la présence d'un contexte.

En effet, la commande `#!pipe python` va lancer un nouveau shell python à chaque appel, tandis que `lang-python` va embarquer directement le moteur python dans radare2, annulant ainsi la limitation de `r2pipe`.

Afin d'installer `lang-python`, il faut d'abord installer et configurer `r2pm` (rien de bien méchant, il suffit juste d'exporter une variable d'environnement) :

Terminal

```
$ git clone https://github.com/radare2/radare2-pm
$ cd radare2-pm
$ export R2PM_DBDIR="$PWD/db"
$ r2pm -i lang-python2
```

NOTE

`r2pm` est un gestionnaire de paquets permettant de lister et installer tous les plugins connus de radare2.

Et voilà une interface offrant de plus grandes perspectives :

Terminal

```
[0x00005520]> #!python
In [1]: a = 10

In [2]:
Do you really want to exit ([y]/n)? y

[0x00005520]> #!python -e print a
10
```

1.4.1 L'interface de plugin

Une fois `lang-python2` installé, `r2` met à notre disposition le module `r2lang`, permettant l'enregistrement de plugins auprès de radare2. Nous allons pouvoir enregistrer 5 types de plugins :

- ⇒ `asm` : ce type de plugin va nous permettre d'assembler/désassembler du code pour de nouvelles architectures.
- ⇒ `io` : les plugins d'io ont pour but de fournir les données à analyser à radare2. Un fichier binaire est géré par un type d'io, le contenu de la mémoire d'un process par un autre.
- ⇒ `anal` : ce type de plugin permet de fournir à `r2` des informations détaillées au sujet d'instructions d'une nouvelle architecture. Radare2 utilisera ces informations lors de l'analyse du code, permettant ainsi entre autres la détection de basic blocks et de fonctions.
- ⇒ `bin` : les plugins bin sont utilisés afin d'ajouter le support de nouveaux formats de fichier, ils permettent de définir un nombre important d'informations, à savoir la liste des segments, des sections, des imports, etc.
- ⇒ `core` : permet simplement d'enregistrer des commandes auprès de radare2.

Afin de s'enregistrer au système de plugins, il suffit d'appeler la méthode `r2lang.plugin` :

Terminal

```
>>> r2lang.plugin("core", mon_plugin.plugin)
```

Le premier paramètre de cette méthode permet de sélectionner le type de plugin que nous voulons implémenter. Le second paramètre est une fonction retournant un dictionnaire de description et de callbacks.

Fichier

```
import r2lang
def plugin(a):
    def _call(s):
        if s.startswith("coucou"):
            print("coucou depuis python !")
            return 1
        return 0
    return {
        "name": "mon_plugin",
        "licence": "GPLv3",
        "desc": "mon premier plugin",
        "call": _call,
    }
r2lang.plugin("core", plugin)
```

Ici comme nous enregistrons un plugin core, seule la callback **call** est nécessaire. C'est elle qui va permettre d'intercepter les commandes utilisateur et de les gérer ou non. Lorsqu'une commande est gérée par le plugin, il doit retourner 1. Autrement, il devra retourner 0 afin de laisser r2 le faire à sa place.

Il suffit ensuite de lancer r2 de la manière suivante pour enregistrer le plugin :

Terminal

```
$ r2 /bin/ls -i plugin.py
[0x00005520]> coucou
coucou depuis python !
```

1.5 Demo time

Afin d'illustrer ce que nous avons vu dans les précédentes sections, nous allons écrire ensemble un moteur d'analyse par teinte basé sur r2 et REIL. Il s'agit ici d'une approche naïve avec un certain nombre de limitations afin de nous introduire aux bases de l'analyse de code.

Nous allons donc ajouter trois commandes à r2 :

- ⇒ **t.tr** : applique une teinte à un registre de notre choix ;
- ⇒ **t.tm** : applique une teinte à une adresse de notre choix ;
- ⇒ **t.ds** : en mode debug, cette commande permettra d'analyser l'instruction courante, appliquera des commentaires, puis procèdera à un step.

Pour ce faire, nous allons d'abord mettre en place une interface avec r2. Pour cet exemple, nous avons, a priori, besoin de 7 commandes pour arriver à nos fins :

- ⇒ **CC** qui nous permettra de commenter le code automatiquement ;
- ⇒ **s** afin de nous déplacer dans le fichier ;
- ⇒ **ds** pour procéder à l'exécution pas à pas du code ;
- ⇒ **dmj** pour la détection du mode debug ;
- ⇒ **?** qui nous permettra d'évaluer des expressions mathématiques simples ;
- ⇒ **aoj** afin de récupérer certaines informations concernant l'instruction ;
- ⇒ **aor** pour les informations sur la sémantique des instructions.

Fichier

```

class R2(object):
    CMD_HANDLED = 1
    CMD_NOT_HANDLED = 0
    def __init__(self):
        self.r2 = r2pipe.open()
        self.commands = {}

    def isdebugging(self):
        if self.get_maps():
            return True
        return False

    def get_maps(self):
        return self.r2.cmdj("dmj")

    def seek(self, addr=None):
        if addr:
            self.r2.cmd("s {}".format(addr))
            return int(self.r2.cmd("s"), 16)

    def set_comment(self, comment, address=None):
        if address:
            address = str(address)
            self.r2.cmd("CC- @ {}".format(address))
            self.r2.cmd("CC {} @ {}".format(comment, address))
        else:
            self.r2.cmd("CC-".format(comment))
            self.r2.cmd("CC {}".format(comment))

    def analyse_inst(self):
        return self.r2.cmd("aor")

    def step(self):
        self.r2.cmd("ds")

```

Ensuite, nous aurons besoin d'un embryon de classe qui représentera notre plugin, que nous appellerons R2TE :

Fichier

```

class R2TE(object):
    def __init__(self):
        self.r2 = R2()
        self.handlers = dict()
        self.register("ds", self.cmd_step)
        self.register("tr", self.cmd_taint_register)
        self.register("tm", self.cmd_taint_memory)

    def register(self, name, func):
        self.handlers[name] = func

    def handle(self, command, args):
        self.handlers[command](args)

    def cmd_analyse(self, a):
        pass

    def cmd_step(self, a):
        pass

    def cmd_taint_register(self, a):
        pass

    def cmd_taint_memory(self, a):
        pass

```



```
def plugin(self, a):
    def _call(s):
        args = s.split()
        if not args[0].startswith("t."):
            return R2.CMD_NOT_HANDLED
        try:
            module, command = args[0].split(".")
            self.handle(command, args[1:])
        except Exception as e:
            print e
            return R2.CMD_NOT_HANDLED
    return {
        "name": "R2TE",
        "licence": "GPLv3",
        "desc": "Radare2 taint engine",
        "call": _call,
    }
r2te = R2TE()
r2lang.plugin("core", r2te.plugin)
```

Afin de pouvoir procéder à l'analyse par teinte, il nous faudra d'abord parser les informations de la commande **aor** (il n'existe pas de commande **aorj**). Nous créons donc une classe *operand* et nous mettons à jour la méthode **analyse_inst** de la classe R2 afin qu'elle retourne une liste d'instructions REIL (une instruction étant implémentée comme un tuple contenant le mnémonique et une liste d'opérandes).

Fichier

```
class operand(object):
    def __init__(self, optype, opval, opsize):
        self.optype = optype
        self.opval = opval
        if self.optype == "reg":
            self.opval = opval
            if opval in unalias:
                self.opval = unalias[opval]
        self.opsiz = opsize
```

Le dictionnaire **unalias** est utilisé ici uniquement dans le but de ne pas avoir à différencier les variantes de taille d'un registre donné et de toujours retourner la version 64 bits du registre.

Cela dit, nous gardons l'information de taille *opsiz* fournie par REIL, ce qui nous permettra de déterminer la taille de la mémoire teinte lorsque la teinte passera d'un registre à la mémoire.

Fichier

```
def analyse_inst(self):
    def _ssplit(s, d):
        return [ e.strip() for e in s.split(d, 1) ]

    def _parse_op(op):
        if ":" not in op:
            return operand("none", None, None)
        val, size = _ssplit(op, ":")
        if val.startswith("R_"):
            return operand("reg", val[2:], int(size))
        if val.startswith("V_"):
            return operand("ref", val[2:], int(size))
        if val.startswith("0x"):
```

```

        return operand("imm", int(val, 16), int(size))
    return operand("imm", int(val, 10), int(size))

def _parse_ops(ops):
    operands = list()
    for op in ops:
        op = _parse_op(op)
        if op.optype != "none":
            operands.append(op)
    return operands

def _parse_ril(ril):
    ril = ril.strip()
    try: cmd, ops = _ssplit(ril, " ")
    except: return None
    ops = ops.strip().split(",")
    return cmd, _parse_ops(ops)

instructions = list()
reil_code = self.r2.cmd("aor").split("\n")[1:]
for line in reil_code:
    if "," not in line:
        continue
    _, inst = _ssplit(line, ":")
    instructions.append(_parse_ril(inst))
return instructions

```

Ensuite, il nous faudra trouver un moyen de suivre la teinte qui transite entre les registres.

Les instructions REIL étant au format **MNEMONIQUE OPERAND1, OPERAND2, DESTINATION**, nous allons partir de l'hypothèse que pour tous les mnémoniques autres que STM, LDM et JCC, si **OPERAND1** et/ou **OPERAND2** sont teintés, alors **destination** le sera aussi. Dans le cas où la destination se trouve être teintée, nous la placerons dans un pool des références/registres teintés.

Nous nous intéresserons plus tard à la teinte de la mémoire (STM, LDM) et de RIP en cas de branchement conditionnel (JCC).

Nous aurons donc besoin d'ajouter un pool des références teintées à la classe R2TE :

Fichier

```

class R2TE(object):
    def __init__(self):
        self.r2 = R2()
        self.handlers = dict()
        self.register("ai", self.cmd_analyse)
        self.register("ds", self.cmd_step)
        self.register("tr", self.cmd_taint)
        self.refs = dict()

```

Nous avons besoin de deux méthodes : une pour mettre à jour le pool de références, une pour vérifier si un registre ou un référence est teinté.

Fichier

```

def update_pool(self, op, taint):
    self.refs[op.opval] = taint

def is_tainted(self, op):
    if op.opval in self.refs:
        return self.refs[op.opval]
    return False

```


Maintenant que nous avons toutes les billes en main pour détecter le passage de teinte d'un registre à un autre, nous pouvons ajouter les méthodes `process cmd_taint_register` et `cmd_step` à R2T2 :

Fichier

```
def _process_minst_v1(self, mn, ops):
    ops, dst = ops[:-1], ops[-1]
    taint = any([self.is_tainted(op) for op in ops])
    self.update_pool(dst, taint)

def process(self, code):
    for mn, ops in code:
        self._process_minst(mn, ops)

def cmd_taint_register(self, a):
    self.refs[a[0]] = True

def cmd_step(self, a):
    if not self.r2.isdebugging():
        print "You need to switch to debug mode, see 'ood?'"
    self.r2.seek("rip")
    reil_code = self.r2.analyse_inst()
    self.process(reil_code)
    tainted_regs = [ reg for reg in self.get_tainted_regs() ]
    if len(tainted_regs):
        self.r2.set_comment("tainted: {}".format(", ".join(tainted_regs)))
    self.r2.step()
```

Enfin, nous pouvons enregistrer le plugin auprès de radare2 :

Fichier

```
r2te = R2TE()
success = r2lang.plugin("core", r2te.plugin)
```

Observons le résultat depuis radare2 :

Terminal

```
[0x7f36f4a04ed0]> #!python misc-simple.py
[0x7f36f4a04ed0]> "wa mov rax, rsp; add rbx, rax; mov rax, rsi"
Written 9 byte(s) (mov rax, rsp; add rbx, rax; mov rax, rsi) = wx 4889e04801c34889f0
[0x7f36f4a04ed0]> t.tr rsp
[0x7f36f4a04ed0]> t.ds
[0x7f36f4a04ed0]> t.ds
[0x7f36f4a04ed3]> t.ds
[0x7f36f4a04ed6]> pd 3 @ 0x7f36f4a04ed0
0x7f36f4a04ed0 4889e0 mov rax, rsp ; tainted: rsp, rax
0x7f36f4a04ed3 4801c3 add rbx, rax ; '#' ; tainted: zf,
cf, rax, pf, rsp, rbx, of, sf
0x7f36f4a04ed6 4889f0 mov rax, rsi ; tainted: zf, cf,
pf, rsp, rbx, of, sf
[0x7f36f4a04ed6]>
```

Nous voyons clairement que `rax` et `rsp` sont teints après l'exécution de l'instruction `mov rax, rsp`. Ensuite, la teinte est bien propagée dans les `eflags` et `rbx` après l'exécution de l'instruction `mov rbx, rax`. Enfin, nous observons qu'à la suite de l'exécution de l'instruction `mov rax, rsi`, le registre `rax` n'est plus teinté.

Maintenant que nous sommes en mesure de propager la teinte de registre à registre, tâchons de faire de même pour la propagation registre-mémoire et inversement.

La première problématique à résoudre afin de procéder à la propagation de teinte d'un registre vers une adresse mémoire, est d'être capable de retrouver l'adresse mémoire en question. En effet, pour les architectures x86, un offset peut avoir plusieurs formes (registre * constante + disp, registre * constante + registre, registre, etc.).

Étant donné que radare2 est capable d'évaluer ce genre d'opération via la commande `?`, nous allons extraire l'expression entre crochets et utiliser `r2` pour l'évaluer. Évidemment, cela ne fonctionnera qu'en mode debug.

De plus, certaines instructions telles que `push`, `pop`, `call` et `ret` vont modifier la stack.

Nous allons donc ajouter à la classe `R2` la méthode `resolve_access` qui nous permettra de résoudre l'accès mémoire de l'instruction courante :

Fichier

```
def resolve_access(self):
    self.seek("rip")
    inst = self.r2.cmdj("aoj")[0]

    if inst["mnemonic"] == "push":
        address = self.r2.cmd("? rsp - 8").split(" ")[0]
    elif inst["mnemonic"] == "call":
        address = self.r2.cmd("? rsp - 8").split(" ")[0]
    elif inst["mnemonic"] == "pop":
        address = self.r2.cmd("? rsp").split(" ")[0]
    elif inst["mnemonic"] == "ret":
        address = self.r2.cmd("? rsp").split(" ")[0]
    else:
        access = inst["opcode"].split("(")[1].split(")")
        address = self.r2.cmd("{} {}".format(access)).split(" ")[0]
    return int(address, 10)
```

STM et LDM étant respectivement de la forme : `MNEMONIQUE OPERAND, , MEMOIRE` et `MNEMONIQUE MEMOIRE, , OPERAND`. On pourra facilement propager la teinte d'un registre vers une adresse mémoire et inversement.

De plus, JCC étant de la forme : `JCC OPERAND, , MEMOIRE`, il sera possible de déterminer la teinte de RIP en fonction de la teinte de `OPERAND`.

Nous mettons donc à jour la méthode `_process_minst` de la classe `R2TE` :

Fichier

```
def _process_minst(self, mn, ops):
    dst = ops[-1]
    ops = ops[:-1]
    if mn == "STM":
        access = self.r2.resolve_access()
        for offset in xrange(ops[0].opsize / 8):
            self.mems[access + offset] = self.is_tainted(ops[0])
    elif mn == "LDM":
        access = self.r2.resolve_access()
        if access in self.mems:
            self.update_pool(dst, self.mems[access])
        else:
            self.update_pool(dst, False)
    elif mn == "JCC":
        dst = operand("reg", "rip", 64)
        taint = self.is_tainted(ops[0])
        self.update_pool(dst, taint)
    else:
        taint = any([self.is_tainted(op) for op in ops])
        self.update_pool(dst, taint)
```

Observons le résultat depuis radare2 :

Terminal

```
[0x7fca73c14ed0]> dcu main
Continue until 0x5635b6904c10 using 1 bpsize
hit breakpoint at: 5635b6904c10
```



```
[0x5635b6904c10]> #!python misc-simple.py
[0x5635b6904c10]> ? rsi
140725972353176 0x7ffd5197c098 03777652145740230 131061.3G d5197000:0098
140725972353176 "\x98\xc0\x97Q\xfd\x7f" 0b011111111111111101010001
1001011111100000010011000 140725972353176.0 140725970796544.000000f
140725972353176.000000 0t200110021021022222022011221122
[0x5635b6904c10]> t.tm 140725972353176
[0x5635b6904c10]> t.tm 140725972353177
[0x5635b6904c10]> t.tm 140725972353178
[0x5635b6904c10]> t.tm 140725972353179
[0x5635b6904c10]> t.tm 140725972353180
[0x5635b6904c10]> t.tm 140725972353181
[0x5635b6904c10]> t.tm 140725972353182
[0x5635b6904c10]> t.tm 140725972353183
[0x5635b6904c10]> t.ds
[0x5635b6904c10]> t.ds
[0x5635b6904c12]> t.ds
[0x5635b6904c16]> t.ds
[0x5635b6904c18]> t.ds
[0x5635b6904c19]> t.ds
[0x5635b6904c1a]> t.ds
[0x5635b6904c1c]> t.ds
[0x5635b6904c1f]> t.ds
[0x5635b6904c23]> t.ds
[0x5635b6904c26]> t.ds
[0x5635b6904c2f]> t.ds
[0x5635b6904c34]> t.ds
[0x5635b6904c36]> pd 13 @ main
;-- main:
;-- section_end..plt.got:
;-- section..text:
0x5654e7c6dc10      4157      push r15      ; [14] --r-x
section size 72792 named .text
0x5654e7c6dc12      4156      push r14
0x5654e7c6dc14      4155      push r13
0x5654e7c6dc16      4154      push r12
0x5654e7c6dc18      55      push rbp
0x5654e7c6dc19      53      push rbx
0x5654e7c6dc1a      89fd      mov ebp, edi
0x5654e7c6dc1c      4889f3      mov rbx, rsi
0x5654e7c6dc1f      4883ec58      sub rsp, 0x58      ; 'X'
0x5654e7c6dc23      488b3e      mov rdi, qword [rsi] ; tainted: rdi
0x5654e7c6dc26      64488b042528.  mov rax, qword fs:[0x28] ;
[0x28:8]=-1 ; '(' ; 40 ; tainted: rdi
0x5654e7c6dc2f      4889442448      mov qword [rsp + 0x48], rax ;
tainted: rdi
0x5654e7c6dc34      31c0      xor eax, eax      ; tainted: rdi
```

Nous voilà en mesure de récupérer la propagation de teinte depuis la mémoire vers un registre. Cette implémentation n'est évidemment pas complète, et pleine de limitations (si **rax** est teinté, un **xor rax, rax** marquera **rax** comme étant teinté, ou encore si **al** est tenté, un **shr rax, 8** marquera rax comme étant teinté).

CONCLUSION

Nous avons pu voir dans cet article qu'il était possible de rapidement étendre les possibilités d'analyse de code avec r2pipe et ce, en utilisant uniquement quelques commandes de base de r2.

On pourra par la suite coupler r2 avec d'autres outils/frameworks tels que Triton ou Angr afin de fournir des informations plus détaillées sur le comportement d'un binaire. ■

ACTUELLEMENT DISPONIBLE

MISC N°96 !



BLOCKCHAIN : UN RÉEL PROGRÈS POUR LA SÉCURITÉ ?

NE LE MANQUEZ PAS
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :
<https://www.ed-diamond.com>



AFL / FUZZING / CODE-COVERAGE / LLVM / VULNÉRABILITÉS

USAGES AVANCÉS D'AFL

gapz

La recherche de vulnérabilités en passant par le fuzzing nécessite souvent une bonne préparation en amont afin de ne pas perdre un temps fou pour en définitive ne rien trouver. Cet article va présenter quelques techniques particulières à American Fuzzy Lop, célèbre pour avoir trouvé une quantité impressionnante de vulnérabilités dans des programmes tels que GNU bash, Perl, le kernel d'OpenBSD et bien d'autres.

INTRODUCTION

American Fuzzy Lop, activement développé par lcamtuf (Michal Zalewski), est un célèbre outil de fuzzing qui a notamment réussi à trouver des bugs dans une grande quantité de programmes. En témoigne le joli tableau maintenu sur la page du projet [1], dont une partie provient notamment du « fuzzing project » [2]. On retrouve la dernière version de celui-ci via une archive élégamment nommée [afl-latest.gz](https://lcamtuf.coredump.cx/afl-latest.gz) et toutes les versions antérieures avec un numéro de version maintenu manuellement.

Pour celui qui chercherait une introduction générale à afl ainsi qu'à ces concepts, techniques d'instrumentation de code, stratégies de mutation, et tous les petits détails qui font d'afl un outil incontournable, Kevin Denis a publié un article sur ce sujet dans le hors-série *MISC* n°11 consacré aux outils de sécurité. Aussi, et d'une manière générale pour ce qui va être traité dans cet article, la documentation présente dans les sources du programme sous le répertoire [docs/](#) est très bien faite et il ne faut pas hésiter à en abuser.

La question principale qui va être traitée ici est l'amélioration globale du processus de fuzzing avec afl en utilisant les outils supplémentaires existant dans son écosystème. Il va principalement s'agir de raffiner quelques éléments clefs : couverture de code, optimisation du corpus de fichiers d'entrées, rapidité d'exécution, faciliter les crashes pour certains types de bugs (qui n'auraient pas fait crasher le programme). L'angle d'attaque va être de prendre un exemple simple, mais non trivial, un démon réseau, avec une approche qui part de zéro pour ensuite raffiner le processus de fuzzing afin d'aboutir à la découverte d'une vulnérabilité, la CVE-2016-6254. La seconde partie de l'article traitera d'un usage un peu différent d'afl, à savoir non pas la recherche d'un bug classique (*buffer overflow*, *use-after-free*, etc.), mais de vérifier la cohérence ou l'exactitude d'opérations mathématiques utilisées en cryptographie. On se focalisera uniquement sur les outils et leurs usages à des fins de recherche de vulnérabilités, sans discuter de l'exploitabilité de ce qu'ils pourraient trouver.

Pour celui qui serait avide d'actualité dans la recherche de vulnérabilités, cet article ne traitera hélas pas de la possibilité d'une combinaison avec de l'exécution symbolique (très utilisée avec afl lors du DARPA Grand Challenge 2016). Ce sujet nécessiterait à lui seul un article à part entière, et certainement un autre auteur !

Cet article est essentiellement destiné à ceux qui n'ont que peu utilisé afl. Pour ceux ayant déjà une bonne expérience de l'outil, peut-être glanerez-vous des informations utiles ici et là dans la première partie et apprendrez à utiliser afl pour faire du « differential fuzzing » dans la seconde.

1. RAPPEL SUR AFL EN QUELQUES COMMANDES

Voici quelques rappels de base qui seront utiles pour la compréhension de l'article. Tout d'abord, une utilisation très simple et classique d'afl est de compiler le programme en utilisant [afl-gcc](#), puis de produire ou récupérer un ou plusieurs « samples » (par exemple un fichier jpeg si l'on teste un programme qui va traiter du jpeg), puis de lancer afl et d'attendre simplement un crash. Tout d'abord la compilation, en substance :

```
$ CC=afl-gcc make
```

Terminal

Avant de lancer afl, il faut tout d'abord créer un répertoire qui va contenir l'ensemble des « samples » que l'on va tester dans un premier temps, et un second répertoire qui va servir

à afl pour stocker les données qu'il va produire (et quelques informations supplémentaires relatives à l'exécution en cours) :

Terminal

```
└─ in
  └─ test1
  └─ test2
  └─ [...]
```

L'architecture des fichiers produits par afl est organisée de la manière suivante :

Terminal

```
└─ crashes
└─ fuzz_bitmap
└─ fuzzer_stats
└─ hangs
└─ plot_data
└─ queue
  └─ id:000000,orig:1
  └─ id:000001,src:000000,op:int16,pos:2,val:+0,+cov
  └─ id:000002,src:000000,op:int16,pos:2,val:+1024,+cov
  └─ id:000003,src:000000,op:havoc,rep:16,+cov
  └─ id:000004,src:000000,op:havoc,rep:4
  └─ id:000005,src:000002,op:int16,pos:0,val:+0,+cov
  └─ [...]
```

On retrouve notamment dans **crashes** les données qui ont produit un crash, c'est-à-dire la fin du programme avec un signal particulier (typiquement SIGSEV ou SIGABRT), dans **hangs** celles qui ont produit un time-out (par défaut le time-out est fixé à une seconde, ce qui est modifiable via l'option **-t**) et dans **queue** l'ensemble du corpus de fichiers qu'afl a produit. Tous les fichiers que l'on retrouve sous ces répertoires correspondent à un chemin d'exécution unique dans le code du programme ciblé (à l'exception des « samples » de base fournis par l'utilisateur). Ils utilisent une convention de nommage qui correspond à la manière dont afl a généré les données contenues par ces fichiers.

En termes de configuration, afl passe essentiellement par des variables d'environnements qui sont entièrement détaillées dans [afl-latest/docs/env_variables.txt](#).

Pour rappel, afl n'a à la base aucune connaissance du format des données à soumettre au programme qu'il va fuzzer. La seule possibilité existante, sans entrer dans le développement d'un « mutator » spécifique à un format, va être d'ajouter un fichier contenant un dictionnaire de mots clefs en argument au lancement d'**afl-fuzz**. Il existe d'ailleurs pour cela quelques dictionnaires prêts à l'emploi livrés avec afl (on retrouve notamment du jpeg, png, js ou encore du json, cf. [afl-latest/dictionaries/](#)).

Bien entendu, ceci n'est qu'un rappel extrêmement minimaliste, et même s'il est possible trouver des vulnérabilités, il va être nécessaire la plupart du temps d'aller un peu plus loin. Là sera tout l'enjeu du reste de l'article.

2. FUZZING D'UN DÉMON RÉSEAU : DE LA COMPILATION AU CRASH

L'exemple qui sera présenté tout au long de cette première partie va être un démon réseau, en l'occurrence collectd. Ce dernier a pour but, comme son nom l'indique, de collecter des données sur le système sur lequel il est lancé puis, dans le cas où le plugin « network » est

utilisé, de les envoyer vers un autre démon collectd qui lui se chargera de stocker les informations dans des fichiers (typiquement du rrd). Ce qui sera intéressant ici va être de fuzzer la partie réseau de ce démon, et ce, pour plusieurs raisons. La première, spécifique à collectd, est que très souvent (pour ne pas dire systématiquement) ce démon est lancé avec les droits d'utilisateur root. La seconde raison, d'ordre général, est qu'il n'est pas évident de fuzzer un programme réseau avec afl et que collectd est suffisamment simple pour faire un bon exemple pour cet article. Tout au long de cette partie, notre approche va être de progresser dans le fuzzing de collectd comme si nous étions en train d'effectuer cette analyse pour la première fois, cela dans le but d'avoir une idée générale d'une approche potentielle que l'on pourrait avoir dans un cas réel.

2.1 Collectd

Notre cible va être plus particulièrement la fonction qui s'occupe de parser les données reçues via le réseau. Cette fonction fait partie du plugin « network ». L'idée derrière ce plugin est relative à l'architecture de collectd, qui récolte des données sur des serveurs pour les faire remonter, via le réseau en utilisant le plugin « network », à un démon centralisé. La version vulnérable qui va être utilisée ici est la 5.5.1 de collectd. La compilation est très simple, il y a quelques dépendances pas trop ennuyeuses à satisfaire. Ensuite, on utilisera le fichier de configuration minimaliste suivant :

Fichier

```
Hostname "localhost"
FQDNLookup false
BaseDir "/opt/collectd/var/lib/collectd"
PluginDir "/opt/collectd/lib/collectd"
TypesDB "/opt/collectd/share/collectd/types.db"

ReadThreads 1
WriteThreads 1

LoadPlugin network

<Plugin network>
    Listen "localhost"
</Plugin>
```

Pour lancer collectd, on utilisera l'option **-f** qui permet d'éviter qu'il fork, se mette en background et n'accepte qu'une seule connexion, et l'on spécifiera le fichier de configuration précédent :

Terminal

```
collectd -C ./collectd.conf -f
```

Dans un premier temps, il n'est pas nécessaire de comprendre le fonctionnement de collectd en détail pour le fuzzer avec afl, notamment le format des données utilisées par le plugin « network ». Celles-ci sont bien entendu documentées sur le site officiel, mais l'approche ici sera plus expérimentale, à savoir lancer afl et regarder comment il arrive à progresser dans le code. Car même en n'ayant aucune connaissance de base sur le format des données à soumettre à collectd, il va réussir très rapidement à structurer une entrée valide. Bien entendu, dans le cas de collectd cela n'est pas très impressionnant, mais dans le cas par exemple d'une bibliothèque jpeg, afl réussit à partir d'un fichier contenant « hello » de produire une image valide après seulement quelques itérations (comprendre : produire un en-tête jpeg valide) [3].

2.2 Approche réseau

Une des raisons pour lesquelles afl n'a pas de possibilités de fuzzer directement via le réseau est que c'est une tâche relativement complexe pour prendre en compte les différentes implémentations possibles des aspects réseau. Aussi la philosophie d'afl est-elle de rester simple avec un minimum de configurations à effectuer.

Dans le cas d'un démon réseau, nous avons plusieurs approches possibles afin de fuzzer notre programme. Il faut savoir qu'afl soumet à notre programme des données directement via l'entrée standard ou simplement en passant le nom de fichier en argument. Dans le cas de collectd, ce dernier attend des données sur le port 25826 en UDP lorsque l'on utilise le plugin « network ». Il va donc falloir trouver un moyen afin de fournir les données au démon, soit en passant directement par le port UDP, soit en usant de quelques astuces.

2.2.1 Fork d'afl avec le support du réseau

La première approche que l'on peut adopter va être d'utiliser le patch de Doug Birdwell [4] qui permet d'utiliser directement le réseau. Il est à noter que la version sur laquelle le patch a été appliqué est une très ancienne version et qu'il n'est donc pas vraiment recommandé de l'utiliser tel quel. On peut tout de même noter que cette approche fonctionne, la vulnérabilité trouvée dans collectd qui va être présentée ici a été découverte en premier avec cette méthode. Pour utiliser cette version modifiée d'afl, il va tout de même falloir effectuer une modification du programme ciblé afin qu'il s'arrête après le traitement des données reçues. Moyennant un peu de compréhension de la logique interne du programme, cette modification peut être plus ou moins triviale : un « exit() » correctement placé suffit la plupart du temps. Ensuite, il faut fournir en argument à afl-fuzz les informations relatives au port et protocole de transport à utiliser, par exemple :

Terminal

```
afl-fuzz -i in -o out -m none -t 20+ -D 7 -Nudp://localhost:25826 /opt/collectd/sbin/collectd -C ./collectd.conf -f
```

On peut noter la présence d'un nouveau « timer » (-D), qui va spécifier le délai à attendre avant de soumettre les données au programme. Sinon on risque d'envoyer les données avant qu'il ne soit capable de les traiter, typiquement parce que la partie réseau ne serait pas encore disponible. Il convient de tester expérimentalement la bonne calibration des timers (pour -t également, qui correspond au time-out) afin d'obtenir un résultat cohérent, car il est hautement probable que sans de bonnes valeurs afl-fuzz n'arrive simplement pas à se lancer, ou alors ait un comportement étrange (toutes les exécutions qui time-out, le programme qui ne reçoit jamais les données, etc.). Les limites rencontrées avec cette approche sont notamment sa modularité (certains programmes ne vont simplement pas fonctionner), mais aussi les performances relativement médiocres obtenues en

american fuzzy lop 1.95b (collectd)			
process timing		overall results	
run time	0 days, 4 hrs, 19 min, 33 sec	cycles done	0
last new path	0 days, 0 hrs, 2 min, 21 sec	total paths	473
last uniq crash	0 days, 0 hrs, 3 min, 20 sec	uniq crashes	4
last uniq hang	0 days, 0 hrs, 2 min, 33 sec	uniq hangs	12
cycle progress		map coverage	
now processing	88 (18.68%)	map density	2195 (3.35%)
paths timed out	0 (0.00%)	count coverage	1.22 bits/tuple
stage progress		findings in depth	
now trying	interest 32/8	favorable paths	97 (20.51%)
stage execs	1153/4083 (28.24%)	new edges on	415 (87.74%)
total execs	473k	total crashes	16 (4 unique)
exec speed	30.47/sec (slow!)	total hangs	17 (12 unique)
fuzzing strategy yields		path geometry	
bit flips	103/15.5k, 45/15.5k, 38/15.5k	levels	3
byte flips	1/1940, 2/1918, 2/1874	pending	452
arithmetic	110/108k, 40/60.9k, 23/38.1k	pend fav	80
known ints	6/7681, 34/37.2k, 32/66.2k	own finds	472
dictionary	0/0, 0/0, 0/38.0k	imported	n/a
havoc	31/53.0k, 0/0	variable	n/a
trim	0.92%/722, 0.00%		
[cpu: 203%]			

Fig. 1 : Crash trouvé sur collectd par afl avec un patch permettant de fuzzer directement via UDP (en local).

termes de temps d'exécution par seconde à cause du temps de lancement du démon. La figure présentée ici est la première capture d'écran du crash trouvé sur collectd avec afl en utilisant le patch de Doug Birdwell (et ASan, qui sera présenté un peu plus loin). Comme on peut le constater, malgré des performances médiocres, le crash a été obtenu après seulement un peu moins d'un demi-million d'exécutions, soit environ quatre heures dans notre cas. On peut donc imaginer le trouver bien plus rapidement en usant de quelques optimisations.

2.2.2 « Désoscker » un programme en utilisant PREENY

L'approche en utilisant l'outil PREENY [5] va être d'utiliser `LD_PRELOAD` afin charger une bibliothèque qui va modifier le comportement de certaines fonctions, tel que `accept()` typiquement, en les faisant utiliser les entrées et sorties standards. On va de cette manière « désoscker » l'application. Comme toujours, il est rare qu'en appliquant simplement cette approche cela fonctionne sans autres modifications. Il va de nouveau falloir placer correctement un `exit()` afin que le programme s'arrête après avoir traité les données reçues. Pour collectd par exemple, le lancement d'afl en utilisant PREENY ressemble simplement à cela :

Terminal

```
LD_PRELOAD="/home/gapz/src/preeny/x86_64-linux-gnu/desock.so" afl-fuzz -i in
-o out -m none /opt/collectd/sbin/collectd -C ./collectd.conf
```

2.2.3 Adapter le code du programme

Quand on parle d'adapter le code, autant dire qu'il n'y a pas de recette générale pour cela et que tout dépend énormément de notre cible. Dans le cas d'un serveur, il y a cependant une logique similaire, quelle que soit la cible, étant donné que le serveur récupère les données depuis le réseau puis appelle une fonction pour les traiter dans la grande majorité des cas.

NOTE

On utilise ici de manière légèrement abusive les termes « depuis le réseau » pour simplement décrire l'usage d'une interface spécifique dans le programme, typiquement les *socket*.

L'idée ici va être de prendre des données via l'entrée standard puis de les passer directement à la fonction qui les traite. Dans l'idée, cela donne un peu près cela :

Fichier

```
data = get_data_from_network()
process(data)
```

Qui devient :

Fichier

```
data = get_data_from_stdin()
process(data)
```

Une variante serait de soumettre les données via le réseau à l'intérieur du programme lui-même, car il peut ne pas être trivial d'appeler directement la fonction qui traite les données ou de modifier la partie réseau du programme (des mécanismes de *polling*

présents avant l'appel de cette partie du code pourraient en effet ajouter de la complexité).

Ce qui donnerait :

Fichier

```
data = get_data_from_stdin()
put_to_network(data)
```

Puis, à un autre endroit du code, déclenché par l'envoi de données :

Fichier

```
data = get_data_from_network()
process(data)
```

Le cas de `collectd` est relativement simple, car il y a une fonction spécifique qui est appelée pour parser les « paquets ». Dans le fichier `src/network.c` :

Fichier

```
static int parse_packet (socket_t *se, /* {{{ */
    void *buffer, size_t buffer_size, int flags,
    const char *username)
```

Comme notre objectif est de fuzzer les parties du code qui vont parser ce qui provient du réseau, l'idée, à partir du moment où l'on a identifié la fonction qui s'occupe de cela, va être très simple : ajouter dans `main` une lecture de l'entrée standard qui va copier les données dans un buffer puis effectuer un appel direct à la fonction `parse_packet`. Aussi, il est bien entendu possible, plutôt que d'aller modifier le programme lui-même, d'écrire un programme-test spécifique à cette fin. Voici ce que cela donne sur le démon `collectd`, à savoir `src/daemon/collectd.c` :

Fichier

```
--- collectd-5.5.1-orig/src/daemon/collectd.c 2018-01-10 14:07:35.190904424 +0000
+++ collectd-5.5.1/src/daemon/collectd.c 2018-02-15 10:23:47.809013237 +0000
@@ -25,6 +25,8 @@
 *   Alvaro Barcellos <alvaro.barcellos at gmail.com>
 **/

#include "network.h"
+
#include "collectd.h"
#include "common.h"

@@ -696,6 +698,19 @@
        return (1);
    }

+    #define MAXBUF 1024
+    size_t len;
+    char buf[MAXBUF];
+
+    len = read(STDIN_FILENO, buf, MAXBUF);
+
+    if (len <= 0)
+        return -1;
+
+    parse_packet(NULL, buf, len, 0, NULL);
+
+    if (0) {
+        /*
```

```

        * run the actual loops
        */
@@ -714,6 +735,7 @@

        /* close syslog */
        INFO ("Exiting normally.");
+
    }

    do_shutdown ();

```

Pour pouvoir appeler directement `parse_packet`, il va falloir cependant effectuer une légère modification sur `daemon/Makefile` ainsi que sur `network.h` afin de permettre l'utilisation de cette fonction.

Fichier

```

--- collectd-5.5.1-orig/src/daemon/Makefile.am 2018-01-10
14:07:35.190904424 +0000
+++ collectd-5.5.1/src/daemon/Makefile.am      2018-01-11
14:37:23.575981286 +0000
@@ -67,6 +67,7 @@

        utils_tail.c utils_tail.h \
        utils_time.c utils_time.h \
        types_list.c types_list.h \
+
        ../network.c ../network.h \
        utils_threshold.c utils_threshold.h

```

Ensuite, il suffit de déclarer la fonction `parse_packet` dans `network.h` et de supprimer le mot clef `static` dans `network.c`. À partir de là, il est donc possible, une fois compilé, de lancer `collectd` en lui fournissant directement des données via l'entrée standard. Un simple test via la ligne de commandes peut nous le confirmer :

Terminal

```

$ /opt/collectd/sbin/collectd -C ./collectd.conf -f <<(echo bla)
network plugin: parse_packet: Received truncated packet, try increasing
'MaxPacketSize

```

Cette approche a le grand avantage d'aller cibler très précisément ce que l'on cherche à tester dans le programme. Dans le cas de `collectd`, cette partie est relativement simple et il se peut que dans de nombreux cas cela ne soit pas si évident. Cependant, même si la compréhension de la logique interne du programme et l'adaptation de ce dernier peuvent être une barrière importante, on peut imaginer qu'il soit possible d'appeler des parties du code spécifique qui ne le seraient qu'après, par exemple, une authentification ou simplement l'obtention d'un état précis du programme. Autrement, cela nécessiterait tout une dynamique à mettre en place avant de soumettre nos données avec `afl`, ce qui nécessite l'écriture d'un programme spécifique à cette tâche, avec le désavantage de perdre en performance (on ajoute des étapes qui prennent du temps avant d'arriver à la partie spécifique que l'on veut tester), mais avec l'avantage de ne pas forcément aller dans les détails internes du programme (si l'on teste par exemple une partie correspondant à un protocole normalisé ou si le fonctionnement est simplement documenté).

2.2.4 Adapter le comportement du programme via des options

On a parlé dans la section précédente de la modification du code du programme ciblé ou, de la même manière, de l'écriture d'un programme de test spécifique. On pourrait imaginer qu'il existe une option permettant à `collectd` d'aller lire les données via `stdin`. Quelques

astuces peuvent faire gagner un temps précieux, comme le cas où OpenSSH a été fuzzé avec afl en utilisant son mode « inetd », ce qui a comme finalité la lecture de stdin par le démon d'OpenSSH. Il convient de bien lire la documentation, même si cela paraît évident, pour voir si l'on ne pourrait pas utiliser une astuce pour arriver à nos fins.

2.2.5 Remarques d'ordre général

Il est très courant d'avoir à désactiver les calculs de checksums (pour collectd il n'y en a pas), sinon il est tout bonnement impossible d'escompter d'aller en profondeur dans le code avec afl, car il n'arrivera pas, en modifiant aléatoirement des bits, à générer un checksum valide ! Étant donné que le contrôle de ce dernier arrive généralement au début du processus de traitement des données, on ne pourra pas aller plus loin. Pour cela, deux méthodes existent : désactiver le contrôle directement dans le code (c'est la solution recommandée) ou alors recalculer le checksum après la mutation du côté d'afl (on pourra développer une bibliothèque post-mutation, cf. [afl-latest/experimental/post_library](#)).

2.3 Les premières tentatives

C'est le moment de découvrir les joies d'[afl-fuzz](#) et d'[afl-gcc](#). Au-delà de la compilation et des quelques options qui vont avec ([AFL_HARDEN](#) notamment), on va découvrir, durant cette phase d'essai, que dans presque tous les cas où cela ne fonctionne pas, ce n'est pas afl le fautif, mais l'usage qui en est fait. Un grand nombre d'erreurs est possible : programme mal compilé donc pas d'instrumentation, programme trop lent, mauvaise lecture de stdin et bien d'autres. Un des éléments de base les plus importants à vérifier lors de ces premières tentatives va être la découverte des chemins par afl-fuzz (*total paths* dans l'interface).

2.4 Optimisation des performances

Une fois que l'on a vérifié que l'on pouvait correctement lancer [afl-fuzz](#), il est temps de mettre à contribution quelques techniques afin d'améliorer les performances du fuzzing. Dans notre approche, cette étape se situe relativement tôt dans le processus général du fuzzing, et cela pour la raison qu'il ne sert à rien de gaspiller des ressources des heures durant alors qu'il existe quelques astuces simples à mettre en œuvre afin d'augmenter la vitesse d'exécution.

Outre le fait essentiel de minimiser la configuration du programme cible (de la même manière, si l'on a écrit un programme pour fuzzer une partie spécifique, il faut être le plus minimaliste possible) et de n'utiliser que de petits « samples » (sinon le fuzzer passe un temps énorme à modifier des bits qui ne produisent aucun résultat intéressant, en plus du fait que le programme est simplement plus lent avec des données en entrée plus grandes en taille), vont être présentées ici des techniques qui se basent sur le mode *llvm* d'afl. À savoir les mécanismes d'instrumentation différée et du mode persistant.

2.4.1 Instrumentation différée

Le principe de l'instrumentation différée est très simple : le temps de chargement du programme peut être diminué en « forkant » après un ensemble de tâches effectuées à l'intérieur du `main` dont l'exécution n'est pas nécessaire à chaque lancement du programme pour le fuzzing. Par exemple, parser les options de la ligne de commandes ou encore le chargement des fichiers de configurations. Pour pouvoir utiliser ce mécanisme, il faudra compiler le programme cible avec [afl-clang-fast](#) (qu'il faudra préalablement compiler). N'oubliez pas qu'il vous est nécessaire, pour cela, d'avoir *clang* sur votre système. Aussi, il sera parfois nécessaire de spécifier la variable d'environnement `LLVM_CONFIG` pour que tout cela fonctionne (les erreurs produites sont généralement très compréhensibles) :

Terminal

```
[~/src/afl-2.52b/llvm_mode]$ make

[*] Checking for working 'llvm-config'...
[-] Oops, can't find 'llvm-config'. Install clang or set $LLVM_CONFIG or $PATH
beforehand.
(Sometimes, the binary will be named llvm-config-3.5 or something like that.)
Makefile:67: recipe for target 'test_deps' failed
make: *** [test_deps] Error 1
LLVM_CONFIG=/usr/bin/llvm-config-4.0 make

[*] Checking for working 'llvm-config'...
[*] Checking for working 'clang'...
[*] Checking for '../afl-showmap'...
[+] All set and ready to build.
[...]
```

Une petite astuce au niveau de la compilation du programme cible est d'ajouter un `ifdef` lorsque l'on compile avec `afl-clang-fast`, de façon à éviter de commenter/décommenter du code chaque fois que l'on fait des essais avec le mode `llvm` ou que l'on veut recompiler le programme normalement. Il suffira maintenant de placer correctement notre point de départ pour forker (pour différer l'instrumentation), de la manière suivante :

Fichier

```
// chargement des fichiers de configuration
// parsing de la ligne de commandes
// [...], tout ce qui n'est pas nécessaire d'être relancé à chaque fois

#ifdef AFL_LLVM_MODE
__AFL_INIT() ;
#endif
```

2.4.2 Mode persistant

L'idée du mode persistant est simplement d'exécuter un nombre de fois défini une partie spécifique du code via l'usage de `__AFL_LOOP`. L'idée est extrêmement simple et le gain de performance est tout à fait impressionnant. Cependant, il va falloir faire attention au fait que l'on boucle dans un programme lancé, et qu'il faudra, si cela est nécessaire, mettre le programme dans un état similaire en début de chaque itération. Cela signifie notamment de faire attention à l'état des descripteurs de fichiers, à la remise à zéro de certaines variables ou encore à éviter les fuites mémoire (au risque d'en accumuler très rapidement). Cette fois-ci, voyons un exemple concret. Dans le `main` de `src/daemon/collectd.c`, aux alentours de la ligne 700 (après tout ce qui concerne le chargement des fichiers de configurations et le traitement des options de la ligne de commandes), on peut placer :

Fichier

```
#define MAXBUF 1024
size_t len;
char buf[MAXBUF];

while (__AFL_LOOP(1000)) {

    // réinitialisation
    len = 0 ;
    memset (buf, 0, MAXBUF) ;
```



```
// lecture de stdin
len = read(STDIN_FILENO, buf, MAXBUF);
if (len <= 0)
    return 0;

// fuzzing !
parse_packet(NULL, buf, len, 0, NULL);
}
```

Dans l'extrait de code présenté ici, on va effectuer une lecture de stdin puis un appel à `parse_packet` avec les données ainsi récupérées. La valeur indiquée pour `__AFL_LOOP` indique simplement le nombre maximal d'itérations que va effectuer afl avant de relancer complètement le programme. Cette dernière valeur va notamment éviter que, s'il y a des fuites de mémoires ou une mauvaise réinitialisation de l'état du programme, l'on se retrouve très rapidement à avoir un fonctionnement erratique.

On peut ensuite lancer afl comme on en a l'habitude, et bien veiller à ce qu'il détecte le mode persistant au lancement :

Terminal

```
$ afl-fuzz -i in/ -o out/ -m none /opt/collectd/sbin/collectd -C ./collectd.conf -f
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 2 CPU cores and 1 runnable tasks (utilization: 50%).
[+] Try parallel jobs - see /usr/local/share/doc/afl/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'in/'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[+] Persistent mode binary detected.
[*] Attempting dry run with 'id:000000,orig:1'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 6, map size = 30, exec speed = 375 us
[!] WARNING: Instrumentation output varies across runs.
[+] All test cases processed.

[+] Here are some useful stats:

Test case count : 1 favored, 1 variable, 1 total
  Bitmap range : 30 to 30 bits (average: 30.00 bits)
  Exec timing : 375 to 375 us (average: 375 us)

[*] No -t option specified, so I'll use exec timeout of 20 ms.
[+] All set and ready to roll!
```

Les deux figures présentées ici montrent la vitesse d'exécution pour le cas normal (aucune optimisation) et pour le mode persistant. Comme on peut le constater, on a un gain de performance de l'ordre d'un facteur de 60 ! On peut remarquer également que ces valeurs restent plutôt stables (pour le cas du mode persistant, il y a quelques

légères variations, mais il arrive souvent que cela ne soit pas si stable, comme on pourra le voir plus loin avec une autre figure). Concernant le crash trouvé précédemment, on arrive à le retrouver ici en seulement quatre minutes (contre un peu plus d'une heure dans le cas normal).

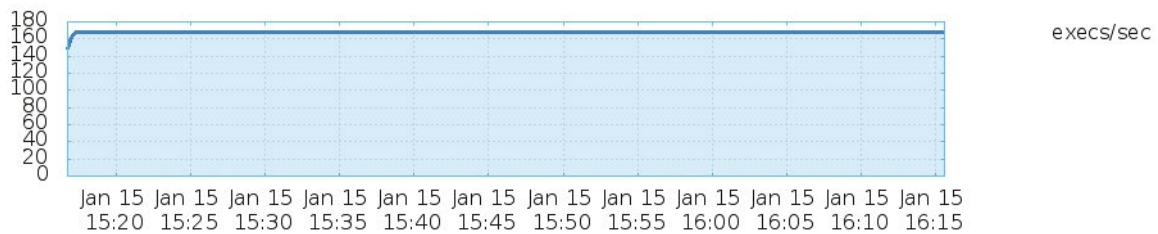


Fig. 2 : Temps d'exécution sur une heure de fuzzing dans le cas non optimisé.



Fig. 3 : Temps d'exécution sur une heure de fuzzing dans le cas optimisé (mode persistant).

2.5 Un peu de « sanitizing »

2.5.1 Address Sanitizer : quelques précautions d'usage

Lorsque l'on va fuzzer notre programme, il est capital qu'il crash afin que l'on puisse détecter l'erreur. Or, dans la plupart des cas de lectures ou écritures invalides en mémoire par exemple, il ne va simplement rien se passer. C'est là qu'entrent en jeu les « sanitizers », et notamment le célèbre *Address Sanitizer* (ou plus simplement ASan), qui va s'attacher à détecter quelques grands classiques tels que les dépassements de tampon (en lecture et en écriture) en tout genre (heap/stack) ou encore les *use-after-free* et faire ainsi crasher le programme lorsqu'il y aura un usage invalide (attention, ASan ne détecte pas les fuites de mémoires, pour cela rien ne vaut Valgrind, mais ASan intègre sa solution, *LeakSanitizer*). Il existe bon nombre d'autres « sanitizers » qui ne seront pas couverts par cette section, mais dont les retours d'expérience sont facilement trouvables ici et là sur la toile. Une des raisons principales pour laquelle ASan a été choisi ici est tout simplement parce qu'il s'interface extrêmement bien avec afl. Autant le dire dès à présent, l'usage d'un tel outil n'est vraiment pas optionnel lorsque l'on va fuzzer, car il est extrêmement efficace pour détecter des bugs en apparence triviaux, mais difficiles à déceler (comme heartbleed). ASan a déjà été l'objet d'un article à part entière dans le numéro 96 de *MISC* « Détection de failles avec Address Sanitizer » [6].

Tout d'abord, il convient de prendre conscience de ce que signifie fuzzer avec ASan activé. Il va en effet se charger de surveiller les accès mémoire afin de vérifier qu'ils soient tous corrects en maintenant notamment une liste d'adresses potentiellement valides. Pour ce faire, il lui est nécessaire de demander l'allocation d'une quantité très importante de mémoire. Cela a pour conséquence directe que si le programme fuzzé se met à consommer réellement toute cette mémoire allouée, le système hôte pourrait devenir instable (on s'en rend généralement compte après un OOM qui va engendrer la

mort de quelques processus afin de libérer de la mémoire). La documentation officielle d'afl nous met bien en garde, et propose de mettre en place un contrôle rigoureux de la mémoire via plusieurs scénarios :

- ⇒ essayer au maximum de fuzzer sur une architecture 32 bits (si l'on est sur 64 bits, mais que le système est « multi-arch », on peut toujours compiler le programme cible en 32 bits, via **-m32**) et de fixer une limite mémoire au lancement d'**afl-fuzz** via l'option **-m**, (la documentation officielle propose de commencer avec 800M) ;
- ⇒ si l'on ne peut pas se passer d'une architecture 64 bits, alors il convient de contrôler la mémoire encore plus strictement en mesurant ce dont le programme cible va avoir besoin (en utilisant par exemple **recidivm**, un outil développé par un des contributeurs réguliers d'afl, Jakub Wilk, <http://jwilk.net/software/recidivm>) ;
- ⇒ passer par l'usage des *cgroups* afin de limiter la mémoire (cf. afl-latest/experimental/asan_cgroups/limit_memory.sh).

Dans les faits, et pour avoir personnellement un peu expérimenté sur une architecture 64 bits, il arrive plutôt rarement que le système devienne instable, et cela en passant **-m none** au lancement d'**afl-fuzz**. Hanno Böck, du « fuzzing project », a fait également le même constat (sachant qu'il a testé un très grand nombre de programmes et même utilisé une distribution GNU/Linux avec un userland entièrement compilé avec le support d'ASan, et a découvert une petite quantité de bugs de cette manière).

2.5.2 ASan par la pratique

Pour pouvoir utiliser ASan, il va falloir compiler notre programme cible en ajoutant son support dans les options de compilation. Comme le monde est bien fait, afl offre la possibilité d'utiliser une variable d'environnement pour cela :

Terminal

```
export AFL_USE_ASAN=1
```

Après il suffit de compiler comme on en a l'habitude avec **afl-gcc** (ou un des autres wrappers d'afl). Une des conséquences de l'utilisation d'ASan, en plus de trouver des bugs bien entendu, est une baisse de performance, légère, mais tout de même notable (de l'ordre de 30 % sur collectd). Il convient de souligner également qu'en condition normale, ASan ne produit pas de faux positifs (ça vous évitera de vous perdre en conjectures sur un dysfonctionnement de ce dernier). On peut également noter l'existence de **libdislocator**, une bibliothèque pour remplacer l'allocateur de la libc afin de faciliter les crashes relatifs au tas, qui aura un impact moindre au niveau des performances par rapport à ASan (et en couvrant un spectre moindre de bug). Pour pouvoir utiliser cette dernière, il faudra utiliser la variable d'environnement **AFL_PRELOAD** (en indiquant le chemin vers libdislocator.so) qui va se charger d'effectuer un **LD_PRELOAD** pour le programme cible sans perturber afl lui-même (ce qui se passerait si l'on utilisait directement **LD_PRELOAD**).

2.6 Optimisation du corpus de « samples »

Lorsqu'on a laissé tourner suffisamment de temps afl pour avoir un bon nombre de chemins découverts, il va être nécessaire d'optimiser notre corpus de fichiers contenant les données à tester afin d'améliorer notre efficacité (meilleure couver-

ture de code, moins de bits superflus). En substance, on va utiliser **afl-cmin** et **afl-tmin** qui vont minimiser notre corpus : l'un opérant sur le corpus afin d'élaborer un sous-ensemble de ce dernier avec une couverture de code équivalente, l'autre opérant sur les fichiers eux-mêmes afin de les réduire à une taille optimale (c'est-à-dire le plus petit et simple possible sans modifier le chemin d'exécution). Dans un premier temps, il faudra donc lancer **afl-cmin** sur le répertoire de travail d'afl (le répertoire que l'on spécifie après l'option **-o** au lancement d'afl-fuzz) pour en extraire un sous-ensemble. Ensuite, on pourra boucler sur l'ensemble des fichiers de ce nouveau sous-ensemble afin de les minimiser en utilisant **afl-tmin**. Retrouvez quelques astuces à ce propos dans l'article de Brandon Perry [11].

2.7 Visualisation de la couverture de code

En plus d'utiliser les différentes métriques mises à disposition via l'interface d'afl, (comme le nombre de chemins découverts, la *map density*, ou encore le *level*, cf. **afl-latest/docs/status_screen.txt**), il manque tout de même un élément essentiel : les parties exactes du code que l'on atteint et celles que l'on n'atteint pas. Une solution va être d'utiliser **afl-cov** [12], qui va permettre de générer un rapport html de la couverture de code en utilisant gcov :

Terminal

```
/home/gapz/src/afl-cov/afl-cov -d /out/collectd/out-inprocess/ --coverage-cmd
"/opt/collectd/sbin/collectd -C /home/gapz/fuzzing/collectd/hello-inprocess/
collectd.conf -f < AFL_FILE" --code-dir /home/gapz/src/collectd-5.5.1-cov/
```

Une fois l'ensemble des fichiers du corpus traité, un rapport html sera produit et l'on pourra naviguer via ce dernier dans la source du programme avec les informations relatives à la couverture de code (pourcentage des fonctions et de lignes couvertes, quelles parties touchées avec un code couleur).

2.8 Pour boucler la boucle

Il manque, pour finir correctement ce processus, deux éléments. Le premier est de savoir quand est-ce que l'on arrête afl, et le second est de comment traiter les crashes s'il y en a. Concernant le premier point, une idée générale va être de constater qu'afl n'arrive plus à trouver de chemins et qu'il a effectué plusieurs cycles. Aussi, il est important d'optimiser le corpus de fichiers que l'on soumet à afl si celui-ci a découvert une quantité importante de chemins et de relancer le processus. Pour ce qui est des crashes, il existe bon nombre d'outils permettant d'effectuer un tri (**afl-crash-analyzer**, **crashwalk**, **afl-utils**), et il est important de se rappeler que lorsqu'on essaye de reproduire un crash il faut mettre en place la même limite mémoire que l'on a utilisée avec **afl-fuzz**. La documentation (**afl-latest/docs/README**) propose ainsi d'utiliser par exemple :

Terminal

```
$ LIMIT_MB=50
$ ( ulimit -Sv $[LIMIT_MB << 10]; /path/to/tested_binary ... )
```

avec **LIMIT_MB**, la valeur passée par l'option **-m** à **afl-fuzz**.

3. FUZZING « DIFFÉRENTIEL » AVEC AFL

3.1 Les bibliothèques de calcul sur les « grands nombres »

Derrière le terme de « differential fuzzing » se cache une idée très simple à saisir. Pour trouver des erreurs dans un programme, on pourrait comparer les résultats produits par celui-ci avec un autre censé effectuer la même tâche. Ce principe, lorsque l'on pense en termes d'opérations mathématiques, devient encore plus évident. Une application directe va être de tester des implémentations d'opérations cryptographiques (ou mathématiques), en utilisant comme donnée d'entrée des valeurs générées par afl et de comparer les résultats produits par une implémentation cible et une implémentation de référence. Les algorithmes de la cryptographie asymétrique nécessitent l'usage d'une bibliothèque de calculs sur les grands nombres, car les valeurs manipulées vont dépasser la taille maximale de ce que l'on peut stocker dans un seul mot machine (typiquement 64 bits). Ces bibliothèques vont offrir un ensemble d'opérations usuelles, telles que l'addition ou la soustraction, mais aussi des opérations plus spécialisées comme le calcul du PGCD. Elles sont donc une cible de choix pour le « differential fuzzing ». Le cas classique en cryptographie est d'utiliser des vecteurs de tests fournis par le standard pour les intégrer aux tests unitaires, ce qui a une portée limitée. L'approche du fuzzing différentiel est bien entendu applicable aux opérations haut niveau (une vérification de signature par exemple).

3.2 Exemple simple avec BN_sqr() d'OpenSSL : CVE 2014-3570

L'exemple présenté ici fait référence à une vulnérabilité trouvée dans OpenSSL sur la fonction `BN_sqr()` permettant d'effectuer une mise au carré (x^2). L'idée est très simple : il va s'agir de tester la cohérence de cette opération (`BN_sqr()`), en comparant les résultats par cette dernière avec ceux de `BN_mul()` (où il suffira de passer deux fois le même nombre pour effectuer une opération similaire à `BN_sqr()`). Ralf-Philipp Weinmann, lors de la Blackhat 2015 [7], présentait ainsi le code suivant permettant de retrouver cette vulnérabilité :

Fichier

```
len = read(STDIN_FILENO, buf, 256);
if (len <= 0) exit(1);

a = BN_bin2bn(buf, len, NULL); b = BN_bin2bn(buf, len, NULL);
r1 = BN_new(); r2 = BN_new();

if (a == NULL || b == NULL) exit(1);

BN_sqr(r1, a, ctx);
/* BN_mul(r2, a, a, ctx) calls BN_sqr() !!! */
BN_mul(r2, a, b, ctx);

/* raise SIGFPE if results differ */
if (BN_cmp(r1, r2) != 0) return 0/0;
```

Il suffit ensuite de compiler ce programme comme on en a l'habitude avec **afl-gcc** par exemple, puis de lancer **afl-fuzz**. Comme on peut le constater, en quelques lignes de C, il est possible de tester très facilement la cohérence d'opérations mathématiques utilisées en cryptographie.

3.3 Exemple avec la libecc de l'ANSSI

3.3.1 libecc

La **libecc** de l'ANSSI [8] est une bibliothèque écrite en C qui implémente notamment des algorithmes de signatures basés sur les courbes elliptiques. Elle a entre autres comme objectif de n'avoir aucune dépendance (pas même envers la libc !) et a donc implémenté les fameuses opérations sur les grands nombres. La **libecc** est une cible intéressante, car le code est relativement récent et est de très bonne qualité (ce qui n'empêche pas de faire des erreurs, mais il est très agréable pour celui qui doit le lire !). Pour l'exemple proposé ici, aucune vulnérabilité n'a été découverte. L'idée va être de comparer l'implémentation du calcul de l'inverse modulaire pour l'implémentation de **libecc** avec celle d'OpenSSL. Schématiquement, cela donne la figure présentée ici.

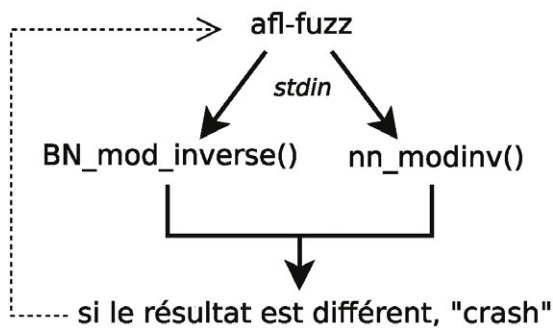


Fig. 4 : Fuzzing « différentiel » pour l'implémentation de l'inverse modulaire, entre libecc et OpenSSL.

3.3.2 Écriture du programme de test

Afin d'implémenter la proposition précédente, il faut donc écrire un programme spécifique à cette tâche. Tout d'abord, il va falloir déclarer et initialiser les variables qui seront nécessaires à l'utilisation des fonctions de calcul de l'inverse modulaire pour **libecc** et OpenSSL :

```
// libecc
nn ae, ne, re;
nn_init(&ae, 0);
nn_init(&ne, 0);
nn_init(&re, 0);

// OpenSSL
BN_CTX *ctx = BN_CTX_new();
```

Fichier


```
BIGNUM *ao = BN_new();
BIGNUM *no = BN_new();
BIGNUM *ro = BN_new();
```

Ensuite, il va falloir aller lire stdin afin d'affecter des valeurs aux différentes variables. Pour rappel, va être effectué ici le calcul de l'inverse « a » modulo « n », et le résultat sera placé dans « r ». Les valeurs de « a » et « n » seront fournies par afl via stdin, et seront affectées ensuite à « a » et « n » (pour OpenSSL les variables sont nommées **ao** et **no** et pour **libecc** **ae** et **ne**). afl ne va fournir qu'une suite de bits via stdin, il va donc falloir la séparer afin d'affecter des valeurs à nos deux variables. Pour ce faire, une astuce d'un exemple fourni par Hanno Böck [9] sera utilisée (**MAXBUF** et **in** représentent respectivement la taille maximale du buffer utilisé pour récupérer les données via stdin et le buffer lui-même) :

Fichier

```
size_t len, l1, l2, l3;
unsigned int divi1, divi2;
unsigned char in[MAXBUF];
unsigned char *a, *b;

len = read(STDIN_FILENO, in, MAXBUF);

if (len < 5)
    return -1;

divi1 = in[0];
divi2 = in[1];
divi1++;
divi2++;
l1 = (len - 2) * divi1 / 256;
l2 = (len - 2 - l1) * divi2 / 256;
l3 = (len - 2 - l1 - l2);
assert(l1 + l2 + l3 == len - 2);

a = in + 2;
b = in + 2 + l1;
if ((l1 == 0) || (l2 == 0) || (l3 == 0))
    return -1;
```

On obtient alors deux valeurs dans **a** et **b** qu'il va falloir assigner aux variables déclarées un peu plus haut :

Fichier

```
// libecc
nn_init_from_buf(&ae, a, l1);
nn_init_from_buf(&ne, b, l2);

// OpenSSL
BN_bin2bn(a, l1, ao);
BN_bin2bn(b, l2, no);
```

Il faut cependant faire attention à la taille des données que l'on soumet avec **nn_init_from_buf**, car la **libecc** ne manipule que des nombres d'une taille maximale de 2^{16} octets. Une petite vérification sur **l1** et **l2** sera alors nécessaire

avant d'effectuer l'appel à la fonction `nn_init_from_buf`. Ensuite vient l'appel des fonctions qui effectuent le calcul de l'inverse modulaire :

Fichier

```
// libecc
if (!nn_modinv(&re, &ae, &ne))
    return -1;
// OpenSSL
if (BN_mod_inverse(ro, ao, no, ctx) == NULL)
    return -1;
```

Maintenant, il va falloir comparer les résultats ainsi obtenus. Seulement, `re` et `ro` n'ont absolument pas le même format. Une des manières de faire la comparaison va être de les transformer en chaînes de caractères avec un encodage en hexadécimal (ou décimal, peu importe) puis de les comparer. Pour cela, OpenSSL nous facilite la vie et propose une fonction qui effectue cette tâche :

Fichier

```
char *resopenssl = BN_bn2hex(ro);
```

Pour ce qui va être de `libecc`, il va falloir effectuer cette conversion à la main, ce qui n'est pas trop compliqué après l'étude de la structure interne utilisée :

Fichier

```
char *reslibecc, *reslibecc_ptr;
reslibecc = reslibecc_ptr = malloc(re.wlen * WORD_BITS);
for (w = re.wlen - 1; w >= 0; w--) {
    reslibecc_ptr += sprintf(reslibecc_ptr, "%016lX", re.val[w]);
}
```

Du fait de cette dernière conversion, il n'est pas possible de comparer directement les deux chaînes de caractères dans leur intégralité via `strcmp`, car il est fort probable que des « 0 » soient au début de `reslibecc`. Plutôt que de perdre du temps à les enlever, on peut directement passer par un appel à `strstr` (ce n'est pas parfait, mais fonctionne expérimentalement). De plus, afin qu'afl puisse détecter une différence de résultats obtenus, il est nécessaire d'arrêter le programme (ce qui sera fait via `assert`) :

Fichier

```
assert(strstr(reslibecc, resopenssl) != NULL);
```

Et enfin, il ne faut pas oublier de libérer la mémoire :

Fichier

```
BN_free(ao);
BN_free(no);
BN_free(ro);
BN_CTX_free(ctx);

free(resopenssl);
free(reslibecc);
```

3.3.3 Fuzzing

Pour lancer le fuzzing, rien de plus simple. Tout d'abord, compilez le programme développé précédemment (il suffit de rajouter quelques `include` pour qu'il soit complet).

Créez par exemple un répertoire **fuzzing/** sous **libecc/src/**, mettez-y votre programme (**nn_modinv-BN_mod_inverse.c**), et utilisez ce Makefile pour compiler tout cela :

Fichier

```
ROOT_DIR=../..
ROOT_DIR=../..
BUILD_DIR=$(ROOT_DIR)/build
include $(ROOT_DIR)/common.mk

# fuzzing
CC=afl-gcc

CFLAGS += -I$(ROOT_DIR)/src/ -I$(ROOT_DIR)/src/external_deps -I/usr/
include

all: nn_modinv-BN_mod_inverse

nn_modinv-BN_mod_inverse: nn_modinv-BN_mod_inverse.c
$(CC) $(BIN_CFLAGS) nn_modinv-BN_mod_inverse.c $(LIBARITH) $(BIN_LDFLAGS)
-ssl -crypto -o nn_modinv-BN_mod_inverse

clean:
@rm -f nn_modinv-BN_mod_inverse

.PHONY: all clean
```

Il suffit maintenant de lancer afl comme on a l'habitude de le faire et d'attendre simplement les crashes ! Comme première valeur d'entrée (le « sample » dans **in/**), on pourra utiliser n'importe quelle valeur (du moment qu'elle soit d'une taille suffisante pour être utilisée) :

Terminal

```
afl-fuzz -i in -o out/ -m none ~/src/libecc/src/fuzzing/nn_modinv-BN_mod_
inverse
```

Il est important de comprendre ce que veut dire attendre dans une telle situation. On va chercher en effet à mettre en évidence des bugs qui auront des probabilités extrêmement faibles d'apparaître. Ce qui signifie certainement avoir un grand nombre d'exécutions totales, à moins d'être très chanceux. Pour donner un ordre d'idée, le problème sur **BN_sqr()** (CVE 2014-3570) n'arrive qu'avec une probabilité de l'ordre de 2^{-128} sur une plateforme 64 bits. afl, grâce notamment à son mécanisme de couverture de code, a réussi à trouver ce bug après un peu moins de 18 millions d'exécutions ! [7]

3.4 Conclusions

Il est intéressant de voir que, même dans le cas de bibliothèques à usage cryptographiques, il est possible d'avoir des implémentations présentant des erreurs de calcul, et celles-ci sont aisément détectables via l'usage du « fuzzing différentiel ». Cependant, l'approche présentée dans cette partie nécessite l'étude détaillée de l'API de l'implémentation visée (et bien entendu de l'implémentation de référence utilisée).

Il existe des approches plus haut niveau, comme celle de CDF (*Crypto Differential Fuzzing*, un framework de fuzzing différentiel), qui est présenté dans un article du présent numéro : « Testons votre crypto ! ». Une autre remarque concerne l'exploitabilité de ces bugs. En effet, quand bien même l'on a une CVE entre les mains, quand en est-il de son exploitabilité réelle ? Il est fort probable que la plupart des erreurs de calcul ne surviennent qu'avec une probabilité extrêmement faible. Et quand bien même on arriverait à atteindre la partie du code problématique, que faire d'une erreur de retenue ? À la conférence 34c3, Filippo Valsorda a fait une démonstration de l'exploitation d'un bug présent dans l'implémentation Go de la courbe NIST P-256, « Squeezing a key through a carry bit », où il réussit, sur la base d'une erreur de retenue, à recalculer l'intégralité de la clef privée [10].

CONCLUSION

Cet article n'aura fait qu'un petit tour des possibilités d'afl, bien d'autres sujets pouvant être abordés (fuzzing distribué, développement d'un « mutator » pour un format spécifique, surcouche d'exécution symbolique, fuzzing en mode blackbox). Ce que l'on a pu voir ici, est qu'avec un peu de code et quelques astuces d'afl, il est relativement simple de mettre en place efficacement du fuzzing, et ce même sur un démon réseau ou dans des cas bien plus spécifiques, tels la vérification d'opérations mathématiques.

REMERCIEMENTS

Merci à lcamtuf pour afl ainsi qu'aux nombreux contributeurs. Merci également à Ryad Benadjila pour les échanges sur la bibliothèque **libecc** ainsi qu'à Laurent Butti et Aurélien pour les relectures. ■

RÉFÉRENCES

- [1] <http://lcamtuf.coredump.cx/afl/>
- [2] <https://fuzzing-project.org/>
- [3] <https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html>
- [4] <https://github.com/jdbirdwell/afl>
- [5] <https://github.com/zardus/preeny>
- [6] <https://connect.ed-diamond.com/MISC/MISC-067/Detection-de-failles-avec-Address-Sanitizer>
- [7] <https://comsecuris.com/slides/slides-bignum-bhus2015.pdf>
- [8] <https://github.com/ANSSI-FR/libecc>
- [9] <https://github.com/hannob/bignum-fuzz/>
- [10] https://media.ccc.de/v/34c3-9021-squeezing_a_key_through_a_carry_bit
- [11] <https://foxglovesecurity.com/2016/03/15/fuzzing-workflows-a-fuzz-job-from-start-to-finish/>
- [12] <https://github.com/mrash/afl-cov>



2

CRYPTOGRAPHIE

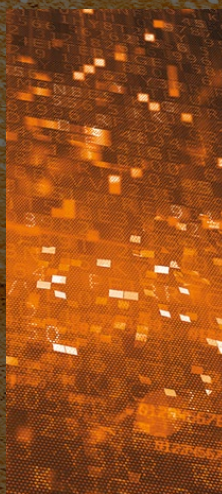
À découvrir dans cette partie...



CRYPTO / TESTS / ATTAQUES PAR CANAUX AUXILIAIRES

Testons votre crypto !

Les vulnérabilités en cryptographie sont une classe bien à part et méritent à elles seules une analyse détaillée. Découvrez Wycheproof et CDF, deux outils vous permettant de vous assister dans cette tâche, et ce de manière automatisée ! p.66



CRYPTOGRAPHIE / DÉTECTION D'ALGORITHMES / GRAP

Algorithmes et implémentations cryptographiques vulnérables : détection avec grap

Vous disposez de binaires et vous voulez savoir s'ils utilisent tel algorithme cryptographique considéré aujourd'hui obsolète ? C'est l'une des nombreuses possibilités offertes par grap, que vous pouvez découvrir dans cet article au travers d'un exemple concret : MD5. p.76

CRYPTO / TESTS / ATTAQUES PAR CANAUX AUXILIAIRES

TESTONS VOTRE CRYPTO !

Yolan ROMAILLER

Où l'on ne parle pas de crypto-monnaie, mais bien de cryptographie. Tester des primitives cryptographiques non pas pour vérifier leur conformité à une spécification donnée, mais bien pour s'assurer de leur sécurité est une tâche ardue. Voyons comment aborder ce problème.

Pour tester un programme, le plus souvent, de nombreuses solutions s'offrent à nous. Le fuzzing, ainsi qu'expliqué dans ce hors-série dans l'article « Usages avancés d'AFL », page 44, fait partie des solutions les plus efficaces et les plus utilisées pour causer des crashes ou des comportements inattendus dans des programmes. Toutefois, lorsque l'on s'intéresse à des domaines spécifiques tels que la cryptographie, on réalise rapidement qu'un programme qui ne plante pas n'est pas pour autant sûr.

Et si Heartbleed eût pu être découvert grâce au fuzzing [BOCK], un problème provenant réellement des primitives cryptographiques utilisées, tel qu'un « padding oracle », ou une « attaque par courbe invalide », ne saurait être détecté par un fuzzer dont le but est de faire *simplement* planter le programme testé.

Les bogues cryptographiques sont généralement plus subtils. Par exemple, lorsque l'on signe un message en utilisant l'algorithme de signature digitale sur courbe elliptique « ECDSA », pour chaque signature, une nouvelle valeur aléatoire éphémère (le plus souvent de 256 bits) est générée. Il suffit d'un biais statistique extrêmement faible (1 bit parmi les 256) [ECDSA] lors de la génération de cette valeur pour permettre à n'importe qui d'extraire des signatures la clef privée utilisée pour les créer. Pour détecter ce genre de problème, des outils spécifiques ont été développés au cours des années. On peut citer parmi ceux-ci le projet Wycheproof [WYCH], ainsi que le développement de l'ACVP (*Automated Cryptographic Validation Protocol*) par le NIST, dont le but est de mettre en place un protocole permettant de valider de manière automatisée la conformité d'algorithmes cryptographiques donnés à leurs spécifications) ou encore CDF (*Crypto Differential Fuzzing*, [CDF]). Tous ont pour but de faciliter le travail du cryptographe en permettant de détecter dès que possible les erreurs classiques faites lors d'une implémentation cryptographique.

Rappelez-vous enfin qu'il est toujours plus difficile de prouver l'absence d'un problème que sa présence, et les systèmes de tests cryptographiques n'échappent pas à cette règle.

1. DÉTECTER DES ERREURS CRYPTOGRAPHIQUES, DE MANIÈRE AUTOMATISÉE

Quelles sont les erreurs cryptographiques que l'on peut espérer détecter automatiquement ? Il s'agit là d'une question dont les réponses actuelles sont encore en plein développement. Cela va notamment dépendre des algorithmes que l'on considère. Commençons par noter que certains problèmes peuvent être qualifiés de problèmes *cryptographiques*, par opposition aux problèmes *fonctionnels* classiques.

Ces erreurs, si elles ne remettent pas en cause la bonne exécution d'un protocole ou d'un programme, peuvent le rendre vulnérable de manière parfois peu évidente. Notre but va être bien entendu de les repérer aussi tôt que possible dans le cycle de vie d'un programme, afin de les corriger.

Commençons donc par passer en revue quelques-uns des problèmes *classiques* rencontrés par un cryptographe lorsqu'il vérifie des implémentations cryptographiques.

Les problèmes dits de canaux auxiliaires (plus connus sous le nom de « side-channel » dans la langue de Shakespeare) : un programme dont la durée d'exécution dépend de valeurs secrètes permet généralement à un adversaire de déduire les valeurs secrètes en chronométrant ses temps d'exécution face à différentes valeurs d'entrée. La présence d'un tel canal auxiliaire permet dès lors d'utiliser le temps d'exécution comme un *oracle* (voir encadré) qui répondra généralement à une question que l'attaquant se pose quant aux valeurs secrètes et qui peut permettre de déduire ces valeurs après avoir fait appel à cet oracle suffisamment de fois. Nous voici face à une attaque par « timing oracle » (oracle de temps) latente.

Les attaques par canaux auxiliaires sont à la mode ces derniers temps avec notamment les vulnérabilités sur processeurs Spectre et Meltdown [SPCTR] qui sont justement basées sur un canal auxiliaire en lien avec le cache du processeur (bien qu'en théorie le problème sous-jacent puisse aussi conduire à d'autres types de

canaux auxiliaires) pour obtenir des informations sur des éléments en mémoire protégée. Ces attaques ne concernent ainsi pas uniquement des éléments qui sont à proprement parler cryptographiques. Quoi qu'il en soit, lors du développement de fonctions cryptographiques il est très important d'être conscient que même le plus petit bit d'information qui fuite peut parfois être exploité par un attaquant.

NOTE

LES ORACLES EN CRYPTOGRAPHIE

En cryptographie, on appelle « oracle » une boîte noire qui prend une valeur d'entrée et donne la réponse à une question précise concernant la valeur d'entrée, sans être restreint en termes de connaissance. Un oracle peut être utilisé indifféremment par n'importe quel utilisateur, y compris un adversaire. On a déjà parlé d'oracle de bourrage (*padding oracle*) dans MISC n° 87 [MISC], qui vont révéler si un texte chiffré, une fois déchiffré, possède une structure de bourrage conforme ou non à sa spécification, et ce, même si la personne qui soumet le texte chiffré à l'oracle n'en possède pas la clef de déchiffrement. De tels oracles ont encore récemment été exploités pour déchiffrer le trafic TLS d'un serveur sans en posséder la clef privée [ROBOT].

Ainsi, ces *timing oracles* sont un exemple type de pièges dans lesquels un développeur qui n'est pas rompu aux arcanes cryptographiques peut tomber. L'illustration la plus simple d'un *timing oracle* consiste en un programme qui va vérifier un mot de passe caractère par caractère et retourner une erreur dès que l'un des caractères est faux :

Fichier

```
func isEqual(a, b string) bool {
    if len(a) != len(b) {
        return false
    }
    for i := 0; i < len(a); i++ {
        if a[i] != b[i] {
            return false
        }
    }
    return true
}
```

Mais avec une telle implémentation, on fournit à l'attaquant un *timing oracle*, car si le premier caractère est correct, mais le second ne l'est pas, le temps d'exécution sera plus long que si le premier caractère est incorrect ! Et en pratique, cette fonction était effectivement présente dans **java.security.MessageDigest** jusqu'en 2009 [HALE]. Le même type de problème de timing peut se retrouver dans des implémentations de RSA lorsque celles-ci vérifient que le bourrage d'un texte déchiffré est correct et peut résulter en une attaque par *padding oracle*, où l'oracle est en réalité le temps d'exécution du programme, avant le rejet d'un texte chiffré au bourrage invalide.

Un autre piège dans lequel il est facile de tomber consiste à ne considérer que des valeurs fonctionnelles et à ne pas s'intéresser au comportement de notre implémentation lorsqu'elle se voit fournir des valeurs d'entrées invalides. Ainsi, certaines implémentations vont valider des signatures DSA invalides avec des clefs invalides, car elles ne vérifient pas que les valeurs utilisées dans ces clefs sont des valeurs plausibles.

Il est d'autant plus important d'également vérifier qu'une fonction ne va pas accepter et effectuer de travail dépendant de valeurs secrètes sur des valeurs d'entrées invalides, car par exemple les algorithmes cryptographiques sur courbe elliptique sont très sensibles à ce genre de problème. Au point que s'il l'on s'amuse à interagir avec un point qui ne se trouve pas sur

la courbe que l'on pensait, on risque de révéler notre clef privée [INVC], ce qui est généralement le pire (ou le meilleur, selon le point de vue) résultat possible d'une attaque contre un algorithme cryptographique.

Nous le mentionnions dans l'introduction, mais les aléas d'un cryptographe sont tels que parfois un biais extrêmement faible lors de la génération d'un nombre dit *pseudo-aléatoire* peut suffire à révéler notre clef privée. On se souvient tous de la PlayStation 3 de Sony dont le microcode était signé cryptographiquement et dont la clef privée avait été révélée lors de la Chaos Communication Conference 27C3, car ils avaient réutilisé plusieurs fois la même valeur éphémère, qui ne l'était dès lors plus, en lieu et place d'en générer une nouvelle durant le processus de signature ECDSA. Le même problème a également permis le vol de plus de 59 bitcoins en 2013, car une application Android n'utilisait pas des valeurs suffisamment aléatoires pour signer des transactions bitcoins.

Maintenant, la question qui nous intéresse est bien de détecter ces problèmes de manière automatique. Quelles sont nos options ? On va bien sûr devoir distinguer quelques cas. Toutefois, plusieurs outils existent et permettent aux développeurs de tester leur code de manière plus ou moins aisée.

L'un des cas théoriquement faciles à détecter, mais trop rarement testé et pas nécessairement facile à corriger consiste à vérifier qu'une implémentation s'exécute en un temps constant. Pour ce faire, plusieurs outils existent tels que Ctgrind [CTGR], un plugin pour Valgrind malheureusement peu utilisé, ou Duedect [DUDE], un programme en C qui permet de tester une fonction contre deux ensembles de valeurs d'entrées et va déterminer à l'aide de tests statistiques si la fonction semble s'exécuter en un temps équivalent dans les deux cas ou non. Cette dernière méthode a l'avantage de permettre le test en *boîte noire* d'une implémentation, ainsi que d'être facilement ajoutée à une suite de tests unitaires.

Vient ensuite le projet d'une équipe de Google, nommé Wycheproof en référence au Mont Wycheproof, la plus petite montagne au monde, afin d'illustrer son but, car plus petite est la montagne, plus elle est facile à franchir. Et son but est bien de rendre aussi facile que possible le test d'implémentation cryptographique. Avec plus de 80 tests, ce ne sont pas moins de 40 problèmes sécuritaires qui furent découverts dans plus de 3 bibliothèques cryptographiques pour Java. Nous allons nous y intéresser plus en détail dans la section suivante.

Un autre projet dont le but est de faciliter le travail des développeurs qui veulent implémenter des primitives cryptographiques s'appelle CDF, un « fuzzeur différentiel » développé chez Kudelski Security que l'on présentera également plus en détail ci-après.

NOTE

LES VECTEURS DE TEST

On appelle « vecteurs de test » les paires constituées des données d'entrée, par exemple une clef et un texte clair, et de leurs données de sortie escomptées, par exemple le texte chiffré correspondant à ce texte clair pour la clef en question. La plupart des algorithmes validés par le NIST disposent de telles valeurs de test dans le cadre de leur programme de validation d'algorithme cryptographique [CAVP]. De plus, la plupart des documents de standardisation tels que les RFCs (*Requests For Comments*) de l'IETF (*Internet Engineering Task Force*) fournissent des vecteurs de test, généralement dans leurs annexes.

Wycheproof a une approche qui consiste à trouver des cas limites, des valeurs invalides, et à produire des vecteurs de tests afin de vérifier qu'une implémentation testée fonctionne ainsi qu'elle le devrait. Ils peuvent également tester de la sorte la vulnérabilité face à certaines attaques, notamment celles de D. Bleichenbacher, qui est l'un des auteurs du projet. CDF quant à lui, va comparer deux implémentations afin de détecter des différences de comportements entre les deux, mais va également tenter de s'appuyer sur des vecteurs de tests afin de tenter de pousser chaque implémentation à la faute, ou à un comportement inattendu. Ainsi, les deux approches sont complémentaires et il est possible en théorie d'utiliser les vecteurs de tests très complets développés par l'équipe de Wycheproof avec CDF en profitant de la notion d'interface

plus générique de ce dernier. Cela n'est toutefois pas encore le cas en pratique et CDF se contente de quelques vecteurs de tests. À l'avenir, les deux projets devraient s'étoffer, l'équipe de Wycheproof ayant annoncé leur intention de publier leurs vecteurs de test dans un format aisément réutilisable et CDF ayant mentionné la possibilité de les ajouter à sa batterie de tests, nous permettant ainsi de tester toujours plus loin nos implémentations cryptographiques.

1.1 Utiliser Wycheproof pour tester votre bibliothèque crypto

Avant de se lancer dans les tests, il est nécessaire d'installer le tout. À la date de publication de cet article, il est relativement facile d'installer Wycheproof [WYCH], mais cela demande vite un peu de temps.

Pour commencer, il est nécessaire d'installer Bazel [BZL], un outil d'automatisation de « build » pour différents langages de programmation, y compris Java. La documentation est bien faite, bien qu'anglophone, mais sur Ubuntu 16.04 le tout se fait en un tournemain.

Il nous faut tout d'abord installer le « Java Development Kit » ainsi que Bazel. Pour ce faire, il suffit d'ajouter le PPA de Bazel et la clef GPG correspondante, et de l'installer depuis les dépôts :

Terminal

```
$ sudo apt-get install openjdk-8-jdk
$ echo "deb [arch=amd64] http://storage.googleapis.com/bazel-apt stable
jdk1.8" | sudo tee /etc/apt/sources.list.d/bazel.list
$ curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
$ sudo apt-get update && sudo apt-get install bazel
```

Attention à se méfier de Bazel, car la rétrocompatibilité n'est pas la priorité du projet et d'une version à l'autre, il se peut que Wycheproof ne soit plus aussi facile à utiliser. Les tests décrits ici sont faits avec Bazel en version 9.

Tout n'est pas fini, car il nous faut encore nous assurer d'utiliser l'extension cryptographique Java qui permet de gérer les *grandes* tailles de clefs lors des tests faits par Wycheproof. Pour des raisons historiques, car les États-Unis limitaient l'export d'algorithme cryptographique *trop fort* en le classant au même niveau que l'export de munitions, et ce jusqu'en 1996, Oracle se limite volontairement à n'exporter que des algorithmes de chiffrement qui ne dépassent pas les 128 bits de sécurité. Afin de profiter de plus de sécurité, il est nécessaire d'activer manuellement l'extension cryptographique Java autorisant des tailles de clefs plus grandes.

Il est possible de faire tourner le tout sans cela, mais vous risquez de rencontrer beaucoup d'exceptions du type « illegal key size » [JSSO]. Il fut un temps où il était nécessaire de télécharger la « JCE unlimited strength policy » indépendamment du reste, mais depuis la version JDK 8u151 (qui se trouve justement être la version actuellement dans les dépôts Ubuntu), la politique de sécurité peut être configurée directement via le paramètre **crypto.policy** dans **jre/lib/security/java.security** et doit se voir assigner la valeur **unlimited** pour effectuer les tests en question sans problèmes.

Une fois cela fait, il faut télécharger ou compiler la bibliothèque à tester. Notez que Wycheproof ne gère (actuellement) que les bibliothèques qui implémentent l'interface cryptographique commune de Java, via la notion de *fournisseur* (« provider ») de Java. On compte notamment parmi celles-ci BouncyCastle, SpongyCastle et les fournisseurs du projet OpenJDK.

Prenons comme exemple SpongyCastle 1.54. Il n'y a pas besoin de l'installer nous-mêmes, car Bazel va s'en charger grâce à Maven. On peut donc simplement le tester en lançant :

Terminal

```
$ bazel test SpongyCastleTest_1_54
[...]
Executed 1 out of 1 test: 1 fails locally.
```

Et l'on constate que le test échoue. Ensuite, il s'agit d'aller voir le journal du test pour constater que l'erreur en question provient de dix-huit des tests effectués par Wycheproof qui ont échoué avec SpongyCastle 1.54. Certains sans que cela porte réellement à conséquence, tandis que d'autres peuvent représenter un réel problème, comme celui-ci :

Fichier

```
18) testExceptionsPKCS1(com.google.security.wycheproof.RsaEncryptionTest)
   java.lang.AssertionError: Exceptions leak information about the padding for RSA/ECB/
   PKCS1PADDING
   at org.junit.Assert.fail(Assert.java:88)
   at com.google.security.wycheproof.RsaEncryptionTest.testExceptions(RsaEncryptionTest.
   java:138)
```

qui nous informe d'un possible oracle de bourrage dans le déchiffrement RSA PKCS #1 causé par des exceptions Java qui diffèrent selon le texte chiffré fourni.

On peut également vouloir tester sa propre implémentation. Heureusement pour nous, le système de fournisseurs de Java nous facilite grandement la tâche : pour tester notre propre bibliothèque, il nous suffit de créer un fichier **wycheproof/java/com/google/security/wycheproof/NotreBiblioTest.java** qui s'occupe simplement de définir notre propre fournisseur comme étant le fournisseur à utiliser, et voilà ! D'où tout l'intérêt d'avoir cette notion d'interface cryptographique commune. Enfin, en pratique si l'on crée notre propre fournisseur de sécurité Java, il faut le signer avec un certificat qui a été validé par la « JCA Code Signing Certification Authority », et pour l'utiliser ensuite avec Wycheproof, il faudra également aller modifier le fichier **wycheproof/BUILD** de sorte à pouvoir utiliser Bazel avec notre nouveau fournisseur.

Wycheproof a de beaux jours devant lui et prévoit de supporter bientôt bien plus de bibliothèques que celles de Java en portant leurs vecteurs de test dans un format plus facilement réutilisable.

1.2 Utiliser CDF pour tester votre implémentation crypto

CDF est l'acronyme de « Crypto Differential Fuzzer » [CDF], en cela qu'il va permettre de tester une implémentation en comparant ses valeurs de sortie avec celle d'une autre implémentation. Ce type d'approche est courant lorsque l'on implémente une nouvelle bibliothèque. Toutefois, la plupart du temps, chaque bibliothèque va implémenter sa propre suite de tests différentiels à l'aide de ses tests unitaires. Certaines ne le font pas et se contentent alors des vecteurs de test fournis généralement en annexe des spécifications des algorithmes implémentés.

CDF tente dès lors une approche plus généraliste et propose la notion d'interface afin de permettre l'interopérabilité entre deux bibliothèques distinctes tant par leur implémentation que par leurs interfaces. Cela nous permet ainsi de comparer leurs comportements, dans l'espoir de détecter des comportements étranges ou déviants d'une implémentation à

l'autre. D'autant plus qu'il n'est jamais facile de vérifier la conformité d'un programme à une spécification sur la base d'uniquement quelques vecteurs de test !

CDF utilise deux approches complémentaires pour tester un programme. La première consiste à tester l'interopérabilité d'une implémentation avec une autre sur des valeurs d'entrées aléatoires, tandis que la deuxième consiste à vérifier à la fois la fonctionnalité et l'interopérabilité face à des cas limites particuliers dont le but est de pousser au vice les implémentations. Différentes manières de comparer l'interopérabilité de primitives existent, notamment en fonction du style d'opération supportée par cette primitive. Parmi celles-ci, on peut distinguer par exemple : le chiffrement qui doit être compatible avec le déchiffrement ainsi qu'illustré sur la Figure 1, la signature d'un message qui doit être compatible avec la vérification de sa validité, ou enfin simplement le fait que les mêmes valeurs de sortie devraient être produites lorsque les mêmes valeurs d'entrée sont utilisées avec un algorithme déterministe (qui n'a pas besoin de valeur aléatoire).

Afin de tester une implémentation avec CDF, il est nécessaire d'implémenter l'une des interfaces de CDF. Prenons par exemple l'interface prévue pour tester la primitive connue sous le nom d'ECDSA. ECDSA est un algorithme de signature digitale à clef publique sur courbe elliptique qui supporte deux opérations : la signature d'un message et la vérification d'une signature. Pour ce faire, l'interface de CDF va tester l'interopérabilité ainsi qu'indiqué sur la Figure 2.

Pour signer un message, il est nécessaire de posséder une clef privée, tandis que pour vérifier une signature, il faut la clef publique correspondant à la clef privée utilisée lors de la signature.

L'interface de CDF pour ECDSA requiert un exécutable pouvant prendre comme entrée une clef privée ainsi qu'un message, et qui doit fournir en sortie la signature de ce message par cette clef privée. Elle requiert également, pour la vérification, un exécutable prenant en entrée un message, une clef publique et une signature, et retournant **true** ou **false**, suivant si la signature est vérifiée correctement ou non.

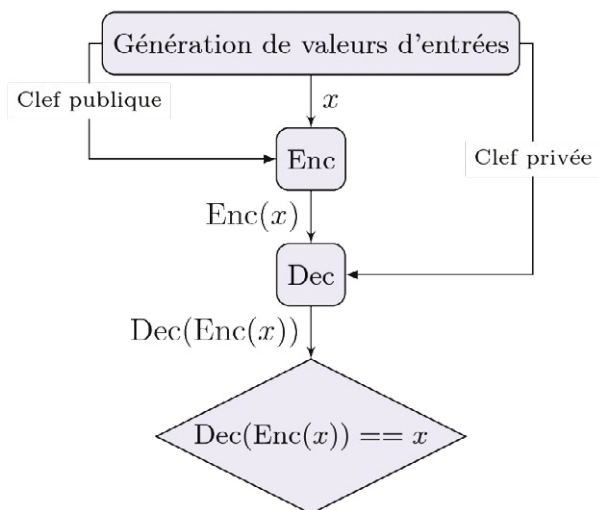


Fig. 1 : Exemple d'interface pour tester un algorithme de chiffrement, tant symétrique (auquel cas la clef publique est la même que la clef privée) qu'asymétrique. On commence par chiffrer (opération « Enc ») une valeur aléatoire x , puis l'on tente de déchiffrer (« Dec ») et l'on vérifie que le résultat obtenu correspond à celui escompté.

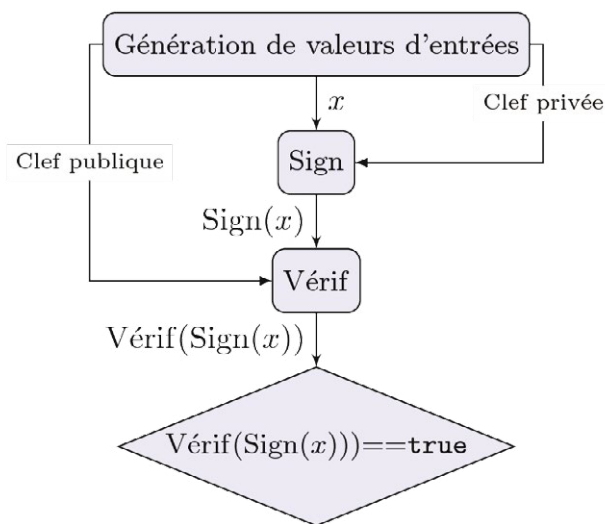


Fig. 2 : Exemple d'interface utilisée pour tester l'interopérabilité d'algorithmes de signature digitale. On commence par signer une valeur aléatoire x et l'on tente ensuite de vérifier la signature en question. Celle-ci devrait se vérifier correctement si et seulement si la clef publique correspond à la clef privée utilisée.

Cette notion d'interface basée sur les entrées-sorties permet à CDF de fonctionner en *boîte noire*, sans avoir besoin d'accéder à des API bas niveau, facilitant ainsi grandement son utilisation avec une implémentation particulière, rendant même le processus de test indépendant du langage de programmation utilisé, du moment que l'on a un exécutable qui implémente cette interface.

CDF intègre également une réimplémentation en Go de Ductect à sa suite de tests, permettant ainsi de comparer les timings des deux implémentations testées, pour voir si elles s'exécutent en un temps constant face à différentes classes de valeurs d'entrées ou non. Ductect étant basé sur une méthode probabiliste, un résultat négatif ne signifie pas pour autant qu'une implémentation fonctionne en temps constant, mais qu'elle l'est *probablement*, et vice-versa.

Pour utiliser CDF, la première étape consiste à sélectionner l'interface correspondant à la primitive testée. Disons que l'on s'intéresse à DSA et que l'on veut tester une implémentation de DSA en Python, par exemple celle de PyCrypto. La première étape consiste à regarder en quoi consiste l'interface de CDF pour DSA. Elle a la spécification suivante :

Opération	Valeurs d'entrée	Valeurs de sortie
Signature	p q g y x m	r s
Vérification	p q g y r s m	true/false

Où les valeurs **p,q** et **g** sont les paramètres DSA, **y** est la clef publique, **x** est la clef privée, **m** est le message et enfin la paire **r,s** représente les deux entiers qui forment une signature DSA. Toutes ces valeurs doivent de plus être au format hexadécimal.

On peut donc implémenter un petit programme en Python qui va satisfaire cette spécification de sorte à prendre les valeurs escomptées en entrée et donner les valeurs correspondantes en sortie :

Fichier

```
#!/usr/bin/env python3
import binascii, sys
from Crypto.Random import random
from Crypto.PublicKey import DSA
from Crypto.Hash import SHA256

# On vérifie que l'on a reçu suffisamment d'arguments en entrée :
if len(sys.argv) == 8:
    signing = False
elif len(sys.argv) == 7:
    signing = True
else:
    print("Veuillez fournir en entrée P, Q, G, Y, X, Msg ou P, Q, G, Y, R, S, Msg")
    sys.exit(1)
# On récupère les arguments en hexadécimal
q = int(sys.argv[2], 16)
p = int(sys.argv[1], 16)
g = int(sys.argv[3], 16)
pub = int(sys.argv[4], 16)
message = binascii.unhexlify(sys.argv.pop())

if signing:
    priv = int(sys.argv[5], 16)
    params = (pub, g, p, q, priv)
else:
```

```

params = (pub, g, p, q)
r = int(sys.argv[5], 16)
s = int(sys.argv[6], 16)
sig = (r, s)

key = DSA.construct(params)

hash = SHA256.new(message).digest()
# Spécialité PyCrypto : s'assurer de ne pas fournir une valeur plus grande
que la taille du groupe.
grp_len = int((q.bit_length() + 7) / 8)
k = random.StrongRandom().randint(1, key.q-1)

if signing:
    # on tronque la valeur d'entrée selon la spécification de DSA puisque
    PyCrypto ne le fait pas.
    sig = key.sign(hash[:grp_len], k)
    print(format(sig[0], 'x').zfill(40))
    print(format(sig[1], 'x').zfill(40))
else:
    if key.verify(hash[:grp_len], sig):
        print("true")
    else:
        print("false")

```

Une fois cela fait, il est aisé de tester notre implémentation avec CDF, il suffit d'avoir installé Go et de faire :

Terminal

```

go get github.com/kudelskisecurity/cdf
cd $GOPATH/src/github.com/kudelskisecurity/cdf
make examples-go
cdf dsa ./votre-executable examples/dsa_sha256_go.go

```

Et la suite de tests de CDF va alors s'occuper de comparer l'implémentation de DSA testée avec celle de la bibliothèque Go standard. Les erreurs s'affichent directement dans la console et sont normalement relativement explicites. En cas de besoin, si par exemple une erreur n'est pas claire, il est possible de poser des questions directement sur la page GitHub de CDF.

CDF devrait également s'étoffer au cours des années, avec plus d'interfaces pour supporter plus d'algorithmes et plus de tests pour chacune de ses interfaces.

CONCLUSION

De nombreuses personnes sont convaincues que la cryptographie est « trop dure » pour être abordée aisément, mais au final la plupart des problèmes cryptographiques dont nous avons tous entendu parler au cours de ces dernières années proviennent de problèmes d'implémentation de fonctions cryptographiques. Ainsi, développer du code cryptographique sûr s'est révélé au fil des années être une tâche ardue. Plusieurs projets, tels que ceux dont nous avons parlé ici, ont pour but de faciliter la tâche tant au développeur qu'à l'utilisateur en leur permettant de tester une implémentation cryptographique et de vérifier la robustesse de celle-ci face à de nombreuses attaques, sans pour autant devoir lire l'intégralité des publications académiques sur ce sujet, et sans avoir besoin de devenir un cryptographe pour ce faire. Toutefois, mieux vaut garder à l'esprit que seul un résultat positif « est vulnérable à X »

est significatif, et qu'un résultat négatif ne veut pas pour autant dire qu'une implémentation donnée est absolument sûre. Tout est relatif et une nouvelle attaque, une variante d'une ancienne attaque ou un détail spécifique à une implémentation donnée peut parfois suffire à compromettre un système. La prudence reste donc de mise.

REMERCIEMENTS

Je tiens à remercier mon entreprise, Kudelski Security, qui m'a permis de rédiger cet article et de développer CDF dans le cadre de mon travail de recherche en cryptographie, ainsi que Jean-Philippe Aumasson pour avoir conçu la notion de fuzzing différentiel et participé au développement de CDF. ■

RÉFÉRENCES

- [BOCK] H. Böck, « How Heartbleed could've been found » :
<https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html>
- [MISC] J. Le Floch, « Les attaques sur RSA : de la théorie à la pratique », *MISC n°87*, septembre-octobre 2016
- [HALE] C. Hale, « A Lesson In Timing Attacks (or, Don't use MessageDigest.isEquals) » :
<https://codahale.com/a-lesson-in-timing-attacks/>
- [INVC] T. Jager, J. Schwenk, J. Somorovsky, « Practical Invalid Curve Attacks on TLS-ECDH » :
https://doi.org/10.1007/978-3-319-24174-6_21
- [CTGR] A. Langley, « Ctgrind » : <https://github.com/agl/ctgrind>
- [DUDE] O. Reparaz, J. Balasch, I. Verbauwhede, « Dude, is my code constant time ? » :
<https://eprint.iacr.org/2016/1123>
- [BZL] Bazel - a fast, scalable, multi-language and extensible build system :
<https://bazel.build/>
- [JSSO] J. Black, « Java Security : Illegal key size or default parameters ? » :
<https://stackoverflow.com/a/6481658/2757014>
- [WYCH] D. Bleichenbacher, T. Duong, E. Kasper, Q. Nguyen, « Project Wycheproof » :
<https://github.com/google/wycheproof>
- [CDF] JP. Aumasson, Y. Romailier, « Crypto Differential Fuzzer » :
<https://github.com/kudelskisecurity/cdf>
- [ROBOT] « Return Of Bleichenbacher's Oracle Threat » :
<https://robotattack.org/>
- [CAVP] NIST Cryptographic Algorithm Validation Program :
<https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program>
- [ECDSA] D. Aranha, P.-A. Fouque, B. Gérard, J.-G. Kammerer, M. Tibouchi, et al., « GLV/GLS Decomposition, Power Analysis, and Attacks on ECDSA Signatures with Single-Bit Nonce Bias », *Advances in Cryptology - ASIACRYPT 2014*, 8873, pp.262-281, 2014,
<https://hal.inria.fr/hal-01094002>
- [PYDSA] R. Rottmann, « DSA Signature Algorithm - A simple implementation in Python » :
<https://github.com/rrothmann/pydsa>
- [SPCTR] « Meltdown and Spectre : Vulnerabilities in modern computers leak passwords and sensitive data » : <https://meltdownattack.com>

CRYPTOGRAPHIE / DÉTECTION D'ALGORITHMES / GRAP / REVERSE

ALGORITHMES ET IMPLÉMENTATIONS CRYPTOGRAPHIQUES VULNÉRABLES : DÉTECTION AVEC GRAP

Léonard BENEDETTI, Aurélien THIERRY et Julien FRANCO

Il arrive régulièrement que des algorithmes cryptographiques deviennent obsolètes ou des implémentations vulnérables. Dans cet article, nous reviendrons sur les raisons qui peuvent pousser à déprécier ces algorithmes ou implémentations et nous présenterons grap, un outil permettant de détecter des motifs à l'intérieur de binaires exécutables et pouvant être utilisé pour rechercher du code cryptographique. Nous illustrerons enfin notre technique avec un cas concret : la détection de MD5, une fonction de hachage cryptographique obsolète encore répandue aujourd'hui.

INTRODUCTION

Lors de l'analyse d'un logiciel ou d'un système, les algorithmes cryptographiques apparaissent comme des remparts importants en matière de sécurité. En effet, pour peu que les algorithmes utilisés soient reconnus robustes et leurs implémentations appropriées, ils constituent une fondation solide sur laquelle s'appuyer. Il est rare par exemple qu'un attaquant analysant un service ayant recours à AES pour des opérations de chiffrement réalise une attaque cryptanalytique sur cet algorithme, qui est standard et considéré aujourd'hui comme sécurisé [AESa].

Cela étant dit, ces remparts ne sont pas infranchissables, tant s'en faut. Il arrive que des recherches aboutissent à des cryptanalyses mettant en évidence des faiblesses majeures dans des algorithmes ou des protocoles réputés sûrs, les rendant de fait obsolètes et impropres pour certains usages. D'autre part — et sans que les algorithmes en eux-mêmes soient remis en cause — des vulnérabilités dans les implémentations de ces derniers sont régulièrement découvertes, ce qui peut avoir pour effet de remettre en cause la sécurité attachée à l'opération cryptographique.

C'est pour ces raisons, et dans la mesure où la sécurité d'un système qui met en œuvre des routines cryptographiques repose souvent de façon critique sur la solidité de ces dernières, qu'il est important d'utiliser des algorithmes cryptographiques considérés comme sûrs par la communauté scientifique, et des implémentations éprouvées conçues pour résister aux différents types d'attaques envisageables (notamment les attaques par canaux auxiliaires). Malheureusement, il est encore fréquent de tomber face à des systèmes qui utilisent des algorithmes obsolètes (parfois depuis très longtemps) ou des implémentations vulnérables. Il est donc intéressant de pouvoir les détecter de façon automatique afin d'accélérer le travail de l'analyste et l'assister dans sa recherche.

Dans la première partie, nous reviendrons sur des algorithmes cryptographiques qui sont aujourd'hui considérés comme n'étant plus sécurisés pour tenter d'illustrer les différentes raisons qui peuvent conduire à cette obsolescence. Dans la deuxième partie, nous présenterons d'une part l'outil grap qui permet de détecter des motifs de graphes au sein de programmes exécutables, et nous détaillerons d'autre part une technique basée sur cet outil qui permet de procéder à la reconnaissance de routines cryptographiques (et donc notamment celles qui sont dépréciées). Il convient de noter que la détection d'algorithmes et d'opérations cryptographiques en général peut se faire à plusieurs niveaux d'abstraction, avec différents outils ; nous nous attacherons donc également à résumer brièvement chacune de ces approches, qui apparaissent comme complémentaires. Enfin, dans une troisième partie, nous présenterons un exemple en montrant comment il est possible de recourir à notre technique avec grap pour détecter des implémentations répandues (celles d'OpenSSL/LibreSSL) de MD5, une fonction de hachage obsolète. Nous détaillerons alors le processus de la construction des motifs pour y parvenir dans ce cas précis.

1. ALGORITHMES CRYPTOGRAPHIQUES OBSOLÈTES

Les découvertes de vulnérabilités dans les implémentations d'algorithmes cryptographiques, ou a fortiori dans leurs utilisations, sont relativement courantes. Elles nécessitent alors des correctifs, mais elles ne remettent généralement pas en question l'algorithme lui-même ou ses fondements mathématiques. Il peut s'agir par exemple de corriger une erreur de programmation qui donnait lieu à une fuite de données ; de rendre constant le temps d'exécution de certaines suites d'opérations qui ne l'étaient pas, et éviter ainsi des attaques par analyse du temps de calcul (susceptibles de déduire des informations secrètes à partir d'une mesure du temps d'exécution de l'algorithme) ; ou encore de rectifier une utilisation impropre d'un algorithme (par exemple, la réutilisation d'un couple clé et *nonce* lors d'une opération de chiffrement).

NOTE

ALGORITHMES ET CANAUX AUXILIAIRES

Même si la résistance des algorithmes cryptographiques aux attaques par canaux auxiliaires est souvent une préoccupation qui relève de leurs implémentations, cette caractéristique n'est pas déconnectée de leur conception. Ainsi, certains algorithmes accordent une grande importance à ce type d'attaques dans leur fonctionnement.

Citons l'exemple d'Argon2 [ARGON2], une fonction de hachage cryptographique publiée en 2014 et destinée à permettre la dérivation de clés et le hachage de mots de passe. Elle a remporté en 2015 la compétition *Password Hashing Competition* [PHC] qui visait à développer de nouvelles approches pour le stockage des mots de passe et à recommander celle considérée comme la meilleure. Cette fonction se décline en deux variantes : Argon2d, particulièrement résistante au crackage par GPU ou ASIC, mais vulnérable aux attaques par canaux auxiliaires, ce qui est adapté pour une utilisation dans les cryptomonnaies (comme preuve de travail) ; et Argon2i, destinée au hachage ou à la dérivation de mots de passe, qui accède à la mémoire indépendamment des données en entrée pour assurer une résistance face aux attaques par canaux auxiliaires. Il est également possible d'utiliser une variante hybride : Argon2id, qui tente d'associer les avantages des deux déclinaisons.

Néanmoins, il peut également arriver que des algorithmes cryptographiques très largement utilisés deviennent obsolètes. Cela donne lieu à chaque fois à des problèmes de migrations pour un certain nombre de systèmes qui continuent d'utiliser — pendant parfois de nombreuses années — des algorithmes cryptographiquement faibles, voire cassés. Revenons ici sur plusieurs de ces cas en détaillant les raisons qui ont mené à ces dépréciations.

DES est un algorithme de chiffrement symétrique par blocs, il a été publié par le *Federal Information Processing Standard* (FIPS) en 1977. Il est dérivé d'un algorithme mis au point par IBM qui a été modifié à la demande de la *National Security Agency* (NSA), très probablement pour le rendre plus résistant face à la cryptanalyse différentielle, une technique qui ne deviendra publique qu'à la fin des années 1980. C'est aux alentours de cette période que sont apparues plusieurs attaques analytiques (cryptanalyses différentielle et linéaire) exhibant ainsi les premières faiblesses. Finalement, l'algorithme tomba définitivement en 1997 : la taille de la clé (56 bits) n'était plus suffisante pour résister aux attaques par force brute, il était alors possible de parcourir l'ensemble de l'espace de clés et de les essayer toutes. Des projets de calcul distribué (DESCHALL et distributed.net) parvinrent à décrypter des messages chiffrés en quelques mois, puis l'*Electronic Frontier Foundation* (EFF) mit au point Deep Crack, une machine basée sur des ASIC (*Application-Specific Integrated Circuits*) dédiés à bruteforcer DES, capable de venir un bout d'un message chiffré en quelques dizaines d'heures.

On préconisa alors de le remplacer par 3DES, un algorithme dans lequel on utilise trois fois DES pour chiffrer un message en ayant recours à une opération de chiffrement, une autre de déchiffrement, puis une dernière de chiffrement ; avec potentiellement à chaque fois une clé différente (portant ainsi la taille de la clé pour 3DES à 168 bits au maximum). Mais de nouvelles faiblesses ont été identifiées menant à une diminution de la complexité des attaques, ce qui a conséquemment conduit à considérer également qu'il s'agissait d'un algorithme faible. Finalement, en 2000, à l'issue du concours éponyme organisé par le *National Institute of Standards and Technology* (NIST), l'algorithme de chiffrement AES fut publié afin de remplacer DES et 3DES. AES est aujourd'hui extrêmement répandu, tandis que l'usage de 3DES reste associé à des applications spécifiques (notamment bancaires [EMV]).

SHA-1 est une fonction de hachage cryptographique publiée en 1995 par le FIPS. Elle prend un message en entrée et produit une empreinte de 160 bits. Ce type de fonctions doit assurer une résistance aux préimages (il doit être difficile de trouver un message qui produit une empreinte choisie ou identique à celle d'un autre message) et aux collisions (il doit être difficile de trouver deux messages qui produisent des empreintes identiques). En 2005, plusieurs attaques sont publiées et mettent en évidence qu'il devient envisageable de

générer des collisions avec une puissance de calcul suffisante [SHA2005] [SHA-Sch]. C'est finalement après plusieurs améliorations de ces attaques théoriques et des démonstrations sur des versions amoindries que se produit, en 2017, « the SHattered attack » [SHattered], qui exhibe une attaque de collisions avec préfixes identiques (c'est-à-dire qu'en utilisant un préfixe imposé suivi de blocs calculés par l'attaque, deux messages différents produiront une empreinte identique). Cette découverte termine de démontrer la faiblesse de SHA-1 et l'importance de migrer vers d'autres fonctions.

Nous terminerons enfin avec Dual_EC_DRBG, un algorithme moins répandu que ceux que nous avons abordés jusqu'alors, mais qui constitue un cas singulier [DECDRBG1]. Il est présenté en 2004 comme un générateur de nombres pseudo-aléatoires cryptographiquement sûr. Malgré les critiques de plusieurs cryptographes, il suit son processus de standardisation dans la norme NIST. En 2007, deux chercheurs de Microsoft avancent l'idée que l'algorithme contienne une porte dérobée. En effet, si certaines constantes utilisées par le générateur sont judicieusement choisies par un acteur, il est possible pour cet acteur de prédire les nombres qui seront générés à partir de ceux qui l'ont été avant [DECDRBG2]. Les enquêtes successives montreront alors que ces constantes ont été choisies sans justification particulière par la NSA (contrairement au principe répandu « *nothing up my sleeve number* », qui préconise de choisir des nombres remarquables — décimales de pi, etc. — qui seraient moins suspicieux que des nombres a priori arbitraires), et qu'elle a également activement participé auprès d'acteurs publics (notamment des organismes de normalisation et de standardisation) et privés (notamment RSA Security) pour pousser à l'utilisation de ce générateur. Finalement, en avril 2014, le NIST a formellement recommandé l'abandon de cet algorithme... On considère aujourd'hui qu'il s'agit là d'une tentative délibérée de la NSA d'introduire et d'encourager l'utilisation d'un algorithme obsolète par construction.

L'utilisation de ces quelques algorithmes dans un cadre cryptographique (mais aussi, plus généralement de tout algorithme cryptographique considéré comme obsolète) fait très souvent courir un risque de sécurité (qui peut être critique lorsque les attaques sont faciles à effectuer), alors pourtant qu'il existe des alternatives reconnues. Mais les migrations peinent parfois à se mettre en place, et l'utilisation imprudente de ces algorithmes peut subsister dans certains systèmes. Il est ainsi intéressant de disposer de techniques permettant la détection de ce type d'algorithmes cryptographiques afin d'identifier de tels systèmes.

2. UTILISATION DE GRAP POUR LA DÉTECTION D'ALGORITHMES CRYPTOGRAPHIQUES

2.1 Présentation de l'outil

grap est un outil open source (licence MIT) [GRAP-git] qui permet de détecter des motifs de graphes, définis par l'analyste, au sein de programmes exécutables x86 (32 ou 64 bits) [GRAP-REcon, GRAP-Sthack].

Le programme prend en entrée des motifs et des fichiers binaires exécutables, utilise un désassembleur basé sur le framework Capstone pour extraire les graphes de flot de contrôle (abrégé CFG pour *Control Flow Graph*) des binaires, puis détecte les motifs dans ces graphes. Ces motifs sont des graphes définis par l'utilisateur à partir de conditions sur les instructions (par exemple : **"opcode is xor and arg1 is rax"**) et de répétitions (par exemple : trois instructions identiques doivent se suivre).

Les motifs prennent la forme de fichiers textes (extension **.grapp**) écrits dans un sous-langage du langage DOT. Cette syntaxe permet de décrire de façon textuelle, mais assez intuitive, des motifs en utilisant des conditions sur la sémantique de l'assembleur (c'est-à-dire sur les instructions : les *opcodes* et leurs opérandes), mais aussi sur sa structure en utilisant les arcs pour former des liaisons entre les instructions (ou en utilisant des conditions telles que le nombre d'enfants ou de parents) et ainsi décrire la topologie à rechercher.

Voici, à titre d'exemple, un motif grap visant à détecter une routine utilisée par le malware Backspace :

Fichier

```
digraph decrypt_sample_4ee00c46da143ba70f7e6270960823be_loop {
  A [label="?? (x3)" cond=true repeat=3 getid="backspace_decrypt"]
  B [label="xor ??, 0x11" cond="opcode is xor and arg2 is 0x11"]
  C [label="sub ??, 0x25" cond="opcode is sub and arg2 is 0x25"]
  D [label="?? (x3)" cond=true repeat=3]
  E [label="j*" cond="opcode beginswith j and nchildren == 2"]

  A -> B
  B -> C
  C -> D
  D -> E
  E -> A [childnumber=2]
}
```

Ce motif vise à détecter une boucle que l'on retrouve dans la fonction de déchiffrement utilisée par Backspace. Il repère les zones où l'on retrouve trois instructions quelconques (c'est-à-dire pour lesquelles la condition de match est toujours vraie) qui mènent à un **xor 0x11**, suivi par un **sub 0x25**, suivi à nouveau de trois instructions quelconques puis d'un saut (une instruction qui commence par **j**) vers les trois instructions quelconques détectées au début (il s'agit donc d'une boucle).

NOTE

UN PEU DE THÉORIE SUR LE DÉSASSEMBLAGE...

Les microprocesseurs x86 ont une architecture de Harvard modifiée, permettant ainsi la présence de code automodifiable, et utilisent un jeu d'instructions de type CISC (*Complex Instruction Set Computing*) où les instructions ont une taille variable.

Dans ces conditions, il n'est pas possible de réaliser un désassemblage parfait dans le cas général. En effet, ce problème — qui consiste en fait à différencier dans le binaire ce qui constitue des données et ce qui constitue du code — est au moins aussi difficile que le problème de l'accessibilité (« est-il possible, lors de l'exécution du programme, d'atteindre une zone donnée ? »), un problème qui est lui-même au moins aussi difficile que le problème de l'arrêt qui a été prouvé indécidable par Alan Turing. Ainsi, puisque le problème indécidable de l'arrêt se réduit au désassemblage de programmes x86 dans le cas général, on conclut subséquemment que ce dernier problème est aussi indécidable.

Quoi qu'il en soit, et malgré cet état de fait dans le cas général, les désassembleurs sont ordinairement capables de produire un désassemblage tout à fait satisfaisant, même si certaines situations (telles que le code automodifiable, ou encore les sauts indirects, du type **jmp rax**, quand ils ne sont pas triviaux) ne seront pas toujours gérés correctement. On pourra alors se retrouver dans des situations où une zone de code n'a pas été correctement désassemblée, ou alors qu'elle l'a été, mais sans que le désassembleur puisse déterminer un chemin pour y accéder alors même qu'il existe.

En somme, dans le cas de l'analyse statique et de grap en particulier, ce n'est pas parce qu'on détecte un motif dans un programme qu'il est nécessairement utilisé. On peut toutefois généralement le vérifier en cherchant un chemin qui mène au code détecté (avec le CFG produit par grap, ou avec un désassembleur), mais ne pas trouver un tel chemin ne permet pas de conclure quant à savoir si ce code est réellement utilisé ou pas (ici, *l'absence de la preuve n'est pas la preuve de l'absence*).

Lorsque grap détecte un motif dans un programme, il renvoie l'adresse des instructions matchées qui sont marquées avec l'option **getid** dans le motif. Cette adresse permet de

retrouver immédiatement le sommet correspondant dans le CFG généré par grap (les sommets sont nommés avec l'adresse des instructions) ; il est alors possible de chercher dans ce graphe les différents chemins qui permettent d'arriver à ce sommet. On peut ainsi vérifier si la zone détectée est réellement utilisée par le programme et dans quelles conditions. La même action peut être réalisée directement dans un désassembleur (ou en utilisant le plugin IDA présenté ci-après) en se rendant à l'adresse de la zone détectée et en recherchant les *cross-references* qui permettent d'y accéder. Il faut cependant souligner que dans les deux cas, on repose sur la qualité du désassemblage effectué pour conclure à l'utilisation du code détecté (voir détails dans l'encadré).

2.2 Utilisation

grap peut être utilisé directement en ligne de commandes ou à l'intérieur de scripts en utilisant les *bindings* Python disponibles. De plus, afin de rendre la détection et la rédaction de motifs plus agréables, nous avons également développé un plugin pour le désassembleur IDA.

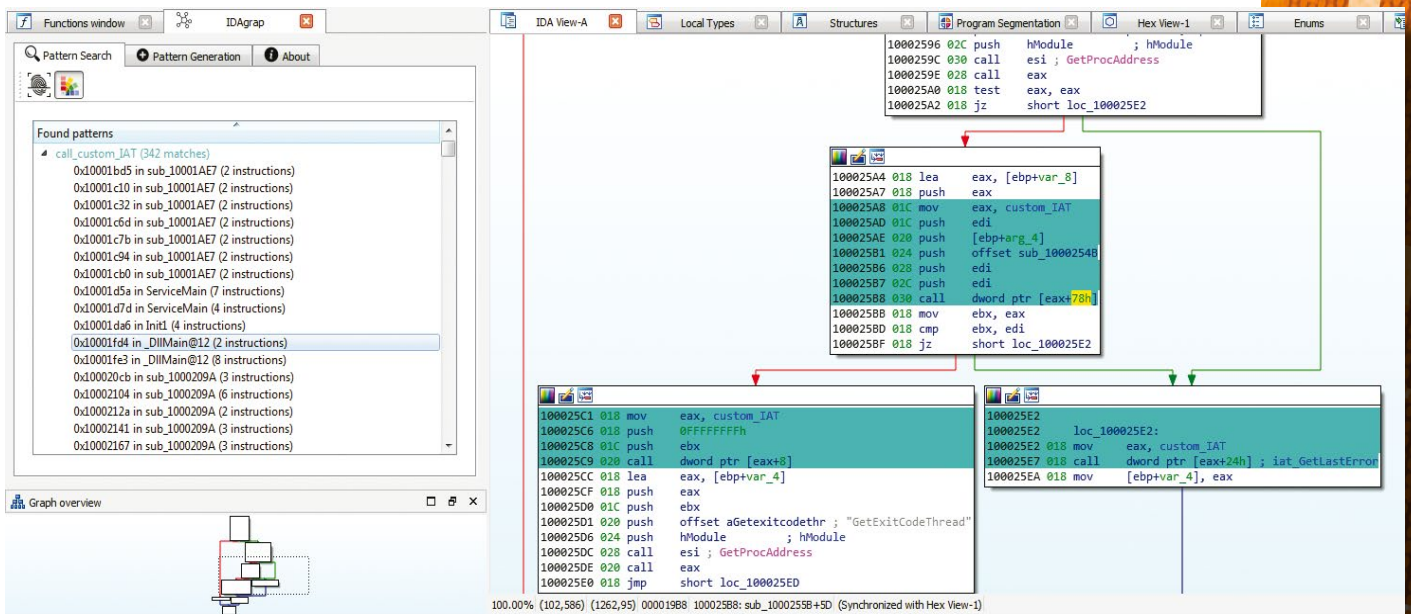


Fig. 1 : Interface graphique du plugin IDAgap permettant de voir directement dans IDA les détections de motifs grap.

La figure 1 présente l'interface du plugin permettant de détecter les motifs au sein d'IDA. Les zones matchées seront colorées et l'analyste pourra se rendre directement aux endroits concernés. Le plugin permet également de créer de façon interactive de nouveaux motifs de détection. En effet, l'écriture manuelle de ces motifs peut se révéler fastidieuse : dans certaines situations, l'analyste veut juste produire un motif rapidement (pour vérifier par exemple si une zone de code ne peut pas se retrouver dans d'autres binaires). À cette fin, le plugin propose un outil de génération de motifs interactifs visible à la figure 2.

Pour l'utiliser, on sélectionne directement dans le CFG l'instruction de départ et celles d'arrivée pour obtenir immédiatement un motif qui détectera le code sélectionné. L'outil permet également d'appliquer au motif généré plusieurs transformations qui visent à rendre ce dernier plus générique. En effet, le motif produit avec les options par défaut ne pourra détecter que les zones qui sont identiques au code sélectionné, alors qu'il est souvent souhaitable de détecter également celles qui sont très similaires sans être pour autant tout à fait identiques. Les transformations disponibles permettront alors de rendre le motif plus tolérant (par exemple, en remplaçant le match de l'instruction de saut **jl** en particulier, par une détection de n'importe quelle instruction de saut).

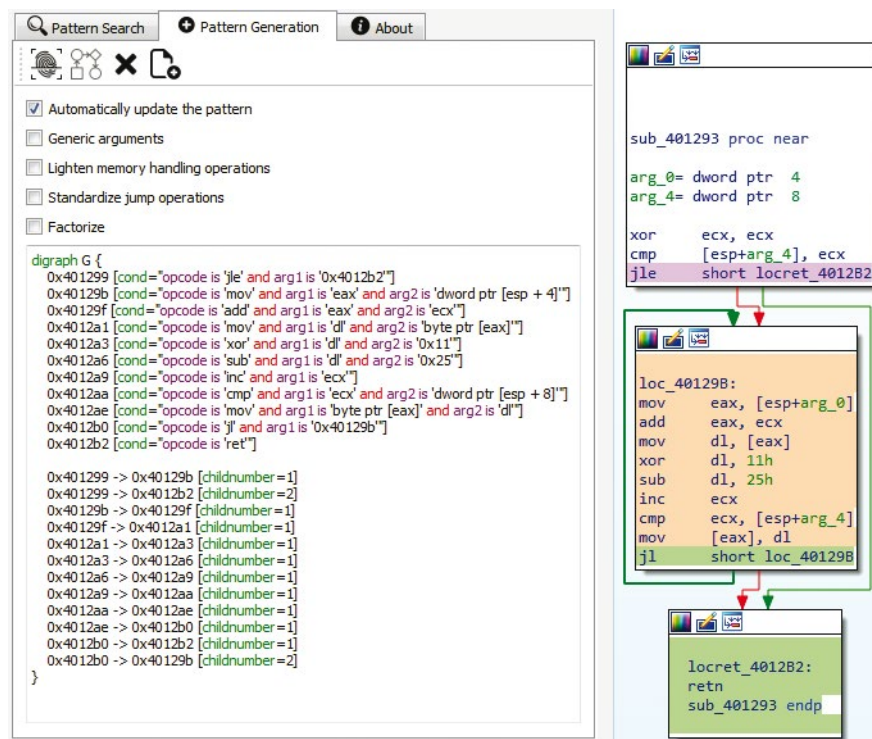


Fig. 2 : Interface graphique du plugin IDAgraph permettant de générer de façon interactive de nouveaux motifs grap directement dans IDA.

Cette possibilité de générer rapidement des motifs relativement génériques est pratique, néanmoins elle ne vient pas se substituer totalement à la rédaction manuelle. Car même si les transformations permettent d'améliorer la généricité du motif produit, et pourront gérer directement des cas simples (recherche de codes identiques, identification de routines en particulier, etc.), lorsque l'on cherche à détecter une implémentation relativement complexe de façon assez précise, il est toujours nécessaire de procéder à un travail d'analyse pour rédiger un motif qui sera à même de gérer les variations non triviales qui apparaissent lors de la compilation. Il n'empêche que cet outil est suffisant pour créer rapidement des motifs pour des tâches simples, et permet de gagner du temps en l'utilisant comme base que l'on viendra modifier pour construire des motifs plus complexes.

Ainsi, l'utilisation de grap permet de faciliter le travail de l'analyste dans plusieurs situations telles que la reconnaissance automatique de routines connues ou l'identification de familles de malware. Nous avons également montré dans de précédents travaux qu'il est possible d'utiliser grap pour détecter des implémentations d'algorithmes cryptographiques [GRAP-Grehack]. Nous proposons de revenir sur la démarche à suivre pour la conception de motifs à cet effet.

2.3 Création de motifs pour la détection d'implémentations cryptographiques

Afin de créer un motif grap permettant de détecter efficacement une implémentation d'un algorithme cryptographique, il faut d'abord étudier attentivement les spécifications de l'algorithme en question. Cette première étape permet d'identifier les opérations effectuées par l'algorithme et sa structure générale. Il faut toutefois garder à l'esprit que, bien souvent, les spécifications et le pseudo-code de référence ont plus vocation à décrire ce que doit faire l'implémentation, et non pas comment cette dernière doit fonctionner.

En effet, un certain nombre de raisons peuvent pousser les développeurs d'une implémentation à modifier, parfois de façon importante, la manière dont l'implémentation va réaliser l'opération cryptographique (même si bien entendu, il faut qu'en fin de compte le résultat soit le même) :

- ⇒ la performance : les références et les spécifications cherchent la plupart du temps à être claires avant d'être performantes, lorsqu'on rédige une implémentation pratique on cherche souvent à garantir une certaine rapidité (ou une certaine taille, notamment dans le secteur de l'embarqué) et l'on procède donc à des optimisations lorsque cela est possible ;
- ⇒ la sécurité : même si un algorithme cryptographique repose sur une base mathématique solide, il est possible d'essayer d'attaquer une implémentation en ayant recours à des méthodes spécifiques : attaques par canaux auxiliaires ou d'autres failles d'implémentation (comme un dépassement de tampon) ;
- ⇒ la modularité : certains algorithmes très proches peuvent présenter des similitudes importantes (par exemple SHA-256 et SHA-512), les développeurs peuvent alors éviter de faire de la réplication de code en n'implémentant qu'une seule fois les opérations en question, les différentes implémentations utiliseront alors des mêmes morceaux de code.

Il en découle que plusieurs implémentations d'un même algorithme peuvent ainsi être très différentes. Il est donc nécessaire pour la détection avec grap de choisir une implémentation précise sur laquelle portera l'étude. La conception du ou des motifs aura alors pour objet la détection de cette implémentation en particulier (et éventuellement, si cela est possible, de celles qui seraient suffisamment similaires à celle-ci).

Ensuite, l'étude de l'implémentation choisie se déroule d'une part sur le code source, afin de constater comment l'algorithme est mis en œuvre, mais aussi et surtout sur les exécutable binaires produits par le compilateur. Il est alors primordial d'étudier plusieurs versions de cette implémentation compilées différemment, c'est-à-dire avec différentes options d'optimisation en utilisant plusieurs compilateurs différents. En effet, lors de la compilation un nombre important de transformations peuvent également être appliquées à des fins d'optimisations (telles que des déroulages de boucle, de l'extension *inline*, des optimisations arithmétiques, et bien d'autres encore). De plus, chaque compilateur a sa logique interne différente de celle des autres qui fait qu'il générera souvent un code assembleur lui aussi différent.

L'analyse se poursuit donc sur ces différents binaires : après les avoir désassemblés, on recherche des invariants dans l'assembleur. Il faut donc comparer les différentes versions en identifiant la sémantique générique (c'est-à-dire les instructions que l'on retrouve d'une version à l'autre) et le comportement de la structure (c'est-à-dire de la topologie du graphe d'une version à l'autre) afin de produire un motif efficace et le plus générique possible. Lors de la rédaction du motif, on aura alors recours aux conditions de répétitions et d'optionnalité sur les instructions permises par grap pour essayer de gérer le plus de cas possible.

2.4 Approches différentes et complémentaires

Au-delà de l'approche que nous proposons, il existe déjà plusieurs techniques qui permettent de détecter des algorithmes (cryptographiques) au sein d'exécutables. On peut les regrouper dans deux catégories en fonction du niveau d'abstraction auquel la technique se place : celles qui vont intervenir directement au niveau du binaire ou des données de l'exécutable (*byte level* ou *data level*) et celles qui vont intervenir au niveau de l'algorithme (*algorithm level*).

La première catégorie est la plus fréquente et peut-être la plus simple à mettre en place. Cela consiste à rechercher des constantes (ou signatures) utilisées par l'algorithme, soit directement au niveau des octets du binaire (par exemple avec des règles [YARA]), soit dans les données de ce dernier après désassemblage (par exemple avec [Signsrch], [FindCrypt2] ou encore [IDAScope]). Cette technique permet une détection très rapide, mais elle repose sur le fait que les constantes (si elles existent) soient directement lisibles et n'aient pas été modifiées.

Les techniques de la seconde catégorie consistent à identifier les algorithmes à partir de leurs relations d'entrées/sorties. On part du principe que pour un jeu d'entrées particulier, l'algorithme produit toujours une même sortie. L'idée de cette approche (mise en œuvre par exemple par [Sibyl] et [Aligot]) est donc de détecter les différentes fonctions au sein de l'exécutable et de les appeler avec des entrées choisies pour vérifier si la sortie correspond à un algorithme connu. Cette technique permet d'identifier correctement des

algorithmes, indépendamment de leur implémentation, même si le binaire a été obfusqué ; de plus, l'ajout de nouveaux algorithmes est aisé (il faut ajouter les sorties de l'algorithme pour des entrées données dans la base de référence). Cependant, cette approche n'est pas forcément applicable sur des binaires de grande taille et la moindre modification de l'algorithme qui viendrait modifier la relation d'entrées/sorties rend la détection impossible.

Notre approche avec grap se place à un niveau d'abstraction situé entre ces deux catégories (*implementation level*). Il est possible de détecter des implémentations indépendamment des constantes (ainsi l'identification est possible même si ces dernières ont été modifiées), la détection se fait sans exécuter le programme et est relativement rapide. En contrepartie, la création de motifs efficaces n'est pas toujours facile ou possible et peut parfois prendre du temps ; d'autre part, cette technique n'est pas du tout robuste face à un attaquant qui cherche activement à éviter la détection.

Par suite, l'approche que nous proposons ne vise pas à remplacer les autres déjà existantes, au contraire : elles ont toutes leurs intérêts et leurs inconvénients et sont complémentaires entre elles.

3. DÉTECTION D'UNE IMPLÉMENTATION DE MD5

3.1 Présentation

MD5 est une fonction de hachage cryptographique publiée en 1992 par Ronald Rivest. Elle visait à remplacer MD4, une autre fonction du même auteur publiée deux ans auparavant et qui semblait présenter des faiblesses en matière de sécurité (ce qui sera confirmé plus tard). Dès 1996, des cryptographes mettent en évidence des faiblesses importantes dans le fonctionnement interne de MD5, et en 2004, une collision complète est exhibée [MD2004]. Dès lors, cet algorithme ne devrait plus être utilisé à des fins cryptographiques, d'autant que de nouvelles attaques seront publiées par la suite.

Malheureusement, la migration vers des algorithmes plus résistants ne s'est pas faite aussi rapidement que l'on aurait pu l'espérer pour l'ensemble des systèmes concernés. Même si MD5 est encore utilisé pour certaines applications bien spécifiques (à ce jour, aucune attaque de préimage envisageable en pratique n'a été publiée pour MD5), son utilisation dans un cadre cryptographique demeure globalement déconseillée. Il est donc intéressant d'être en mesure de détecter de façon automatique des implémentations de cet algorithme.

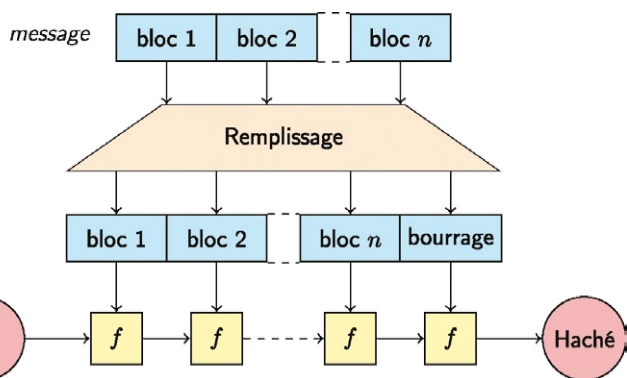


Fig. 3 : Fonctionnement de la construction de Merkle-Damgård (crédit : F. Mouhartem, licence CC0).

3.2 Analyse

3.2.1 Analyse de l'algorithme

MD5 prend un message en entrée et produit une empreinte de 128 bits. Cette fonction repose sur la construction de Merkle-Damgård qui consiste à découper le message à hacher en blocs de taille fixe qui seront donnés en entrée d'une fonction de compression f avec le bloc traité précédemment (avec un vecteur d'initialisation pour le premier bloc). La figure 3 résume ce fonctionnement.

L'algorithme travaille sur un état de 128 bits, chaque bloc est traité par la fonction de com-

pression elle-même composée de quatre tours. Chaque tour consiste à appliquer à seize reprises sur l'état, que l'on divise en quatre mots binaires de 32 bits notés **a**, **b**, **c** et **d** : une fonction non linéaire qui dépend du tour, des additions modulaires et une rotation vers la gauche. Les quatre fonctions non linéaires sont :

$$\Rightarrow F(b, c, d) = (b \wedge c) \vee (\neg b \wedge d);$$

$$\Rightarrow G(b, c, d) = (b \wedge d) \vee (c \wedge \neg d);$$

$$\Rightarrow H(b, c, d) = b \oplus c \oplus d;$$

$$\Rightarrow I(b, c, d) = c \oplus (b \vee \neg d).$$

Ainsi, dans le premier tour **F** est appliquée seize fois, dans le deuxième tour **G** est appliquée seize fois, et ainsi de suite.

3.2.2 Analyse de l'implémentation

Pour cette étude, nous avons choisi d'utiliser l'implémentation de [LibreSSL], un fork d'OpenSSL (dans le cas de MD5, les implémentations sont de toute façon identiques), car c'est une des plus répandues. Nous avons donc analysé le code source de l'implémentation en C ainsi que les binaires produits par différents compilateurs (GCC et Clang) avec différents niveaux d'optimisations (de **-O0** à **-O3**).

On pourra retrouver le code source de l'implémentation étudiée dans le répertoire des sources de LibreSSL **src/lib/libcrypto/md5/**. On peut alors constater que cette dernière est constituée de plusieurs fonctions (**MD5_Init**, **MD5_Update** et **MD5_Final**) qui permettent de préparer le contexte, le mettre à jour, et finaliser le calcul de l'empreinte. La fonction de compression se situe dans la fonction **md5_block_data_order**. Voici un extrait du code source (provenant des fichiers **md5_loc1.h** et **md5_dgst.c**) qui présente différents éléments pertinents :

Fichier

```
#define F(b,c,d) (((c) ^ (d)) & (b)) ^ (d)
#define G(b,c,d) (((b) ^ (c)) & (d)) ^ (c)
#define H(b,c,d) ((b) ^ (c) ^ (d))
#define I(b,c,d) ((~(d)) | (b)) ^ (c)

/* La macro ROTATE(a,s) permet d'appliquer à l'entier a une rotation de s bits
vers la gauche */
#define R0(a,b,c,d,k,s,t) { a+=((k)+(t)+F((b),(c),(d))); a=ROTATE(a,s); a+=b; };
#define R1(a,b,c,d,k,s,t) { a+=((k)+(t)+G((b),(c),(d))); a=ROTATE(a,s); a+=b; };
#define R2(a,b,c,d,k,s,t) { a+=((k)+(t)+H((b),(c),(d))); a=ROTATE(a,s); a+=b; };
#define R3(a,b,c,d,k,s,t) { a+=((k)+(t)+I((b),(c),(d))); a=ROTATE(a,s); a+=b; };

/* La définition de cette macro n'est valable que lorsque le type "unsigned int"
fait 32 bits */
#define HOST_c2l(c,l) ((l)*=((const unsigned int *) (c)), (c)+=4)

void md5_block_data_order (MD5_CTX *c, const void *data_, size_t num)
{
    const unsigned char *data=data_;
    unsigned MD32_REG_T A,B,C,D,l;

    /* [...] Préparation du contexte */

    for (;num--;)
    {
        HOST_c2l(data,l); X( 0)=1; HOST_c2l(data,l); X( 1)=1;

        /* Round 0 */
        R0(A,B,C,D,X( 0), 7,0xd76aa478L); HOST_c2l(data,l); X( 2)=1;
        R0(D,A,B,C,X( 1),12,0xe8c7b756L); HOST_c2l(data,l); X( 3)=1;
        /* [...] 14 autres appels de R0 et 12 de HOST_c2l */
    }
}
```

```

/* Round 1 */
R1(A,B,C,D,X( 1), 5,0xf61e2562L);
R1(D,A,B,C,X( 6), 9,0xc040b340L);
/* [...] 14 autres appels de R1 */

/* Round 2 */
R2(A,B,C,D,X( 5), 4,0xfffa3942L);
R2(D,A,B,C,X( 8),11,0x8771f681L);
/* [...] 14 autres appels de R2 */

/* Round 3 */
R3(A,B,C,D,X( 0), 6,0xf4292244L);
R3(D,A,B,C,X( 7),10,0x432aff97L);
/* [...] 14 autres appels de R3 */

/* Mise à jour de l'état interne */
A = c->A += A;
B = c->B += B;
C = c->C += C;
D = c->D += D;
}
}

```

On peut observer que les expressions des fonctions **F** et **G** ont été simplifiées pour n'utiliser plus que trois opérations au lieu de quatre (ces expressions booléennes simplifiées sont équivalentes aux originales). D'autre part, on remarque que les tours ont été complètement déroulés : les seize appels à la fonction linéaire du tour sont tous écrits explicitement.

Enfin, nous analysons les différents binaires générés avec des compilateurs et des niveaux d'optimisations différents. On peut alors constater plusieurs différences au niveau du code assembleur produit. Les figures 4 et 5 l'illustrent en montrant l'assembleur correspondant au début de la première ronde (la première routine **R0**) avec des niveaux d'optimisations différents : on peut par exemple remarquer dans la version optimisée que l'appel à la procédure **ROTATE** a été remplacé par une rotation vers la gauche (**rol**).

```

xor    eax, dword [rbp+var_50]
and    eax, dword [rbp+var_58]
xor    eax, dword [rbp+var_50]
mov    edx, eax
mov    eax, dword [rbp+var_48]
add    eax, edx
add    dword [rbp+var_5C], eax
mov    eax, dword [rbp+var_5C]
mov    esi, 0x3 ; argument #2 for method ROTATE_13e993
mov    edi, eax ; argument #1 for method ROTATE_13e993
call   ROTATE_13e993 ; ROTATE
mov    dword [rbp+var_5C], eax

```

Fig. 4 : Assembleur du début du premier tour de MD5, généré par GCC au niveau d'optimisations minimum (-O0).

```

xor     edi, esi
mov     r9d, dword [r10+0x18]
mov     r12d, dword [r10+0x1c]
and     edi, edx
xor     edi, esi
lea     esi, dword [rbx+rsi-0x173848aa]
mov     dword [rsp+0x20+var_60], ebx
lea     edi, dword [r13+rdi-0x28955b88]
mov     ebx, dword [r10+0x14]
mov     dword [rsp+0x20+var_50], r9d
add     eax, edi
mov     edi, dword [r10+8]
rol     eax, 0x7

```

Fig. 5 : Assembleur du début du premier tour de MD5, généré par GCC au niveau d'optimisations maximum (-O3).

Cette étape d'analyse va donc permettre d'identifier les différences que l'on peut observer dans différentes conditions de compilation. En plus des exemples déjà cités, nous avons également noté qu'en 64 bits avec les optimisations activées, le compilateur peut parfois *packer* automatiquement les mots binaires 32 bits sur des registres 64 bits. Il est également susceptible de procéder à des réarrangements des opérations de manipulation de la mémoire (**mov** et **lea**).

Enfin, et cela conclura cette phase d'analyse, il faut noter qu'en plus de l'implémentation de MD5 en C, LibreSSL dispose aussi d'une implémentation optimisée directement écrite en assembleur x86_64 (fichier **asm/md5-x86_64.pl**). Il serait souhaitable de pouvoir également détecter cette dernière avec nos motifs.

3.3 Création des motifs

3.3.1 Motif principal

Pour la détection de cette implémentation, nous nous concentrerons sur la fonction de compression et laisserons de côté les fonctions qui servent à manipuler le contexte (**MD5_Init**, etc.) car ces dernières sont principalement constituées d'opérations de manipulation de la mémoire et peuvent donc être fortement modifiées, voire éclatées, lors de la compilation. En conséquence, ce motif s'attachera à caractériser l'implémentation quasiment exclusivement à partir de la sémantique du CFG, et pour cause : la fonction de compression ne comporte pas de saut, la topologie de son graphe ne peut donc pas servir pour la détection.

Nous allons donc essayer de matcher successivement les opérations des fonctions non linéaires (**xor**, **and**, **xor** pour **F**, ou encore **not**, **or**, **xor** pour **I**), les additions et les rotations. On pensera également à ajouter entre ces éléments des nœuds pour les opérations de manipulation de la mémoire (**mov** et **lea**). Ainsi, on pourrait détecter la première étape du premier tour avec ce motif :

Fichier

```
digraph md5_crypto_round0_0 {
    Round0_0_0 [cond="opcode is 'xor'"]
    Round0_0_0a [cond="(opcode is 'mov') or (opcode is 'lea')" repeat=*
    lazyrepeat=true]
    Round0_0_1 [cond="opcode is 'and'"]
    Round0_0_1a [cond="(opcode is 'mov') or (opcode is 'lea')" repeat=*
    lazyrepeat=true]
    Round0_0_2 [cond="opcode is 'xor'"]
    Round0_0_e [cond="(opcode is 'mov') or (opcode is 'lea') or (opcode is
'add') or (opcode is 'sub')" repeat=* lazyrepeat=true]
    Round0_0_r [cond="(opcode is 'call') or (opcode is 'push') or (opcode is
'rol') or (opcode is 'ror')" repeat=* lazyrepeat=true]
    Round0_0_g [cond="(opcode is 'mov') or (opcode is 'lea') or (opcode is
'add') or (opcode is 'sub')" repeat=* lazyrepeat=true]

    Round0_0_0 -> Round0_0_0a
    Round0_0_0a -> Round0_0_1
    Round0_0_1 -> Round0_0_1a
    Round0_0_1a -> Round0_0_2
    Round0_0_2 -> Round0_0_e
    Round0_0_e -> Round0_0_r
    Round0_0_r -> Round0_0_g
}
```

Il suffit ensuite de répéter ce motif seize fois pour gérer l'intégralité du premier tour. Puis de répéter le motif ainsi obtenu quatre fois en modifiant les opérations de la fonction non linéaire pour l'adapter à celle de chaque tour pour obtenir un motif détectant l'implémentation étudiée. Cela étant dit, la rédaction manuelle d'un tel motif serait très fastidieuse, il est donc possible : soit d'utiliser le plugin IDA pour générer un motif de base pour le faire évoluer ensuite, soit d'écrire un petit script qui s'occupera de générer le motif que nous venons de décrire (ce script, ainsi que le motif complet, est d'ailleurs disponible dans le dépôt de grap dans le dossier **patterns/crypto**).

3.3.2 Heuristique

Le motif décrit à l'instant permet de matcher précisément l'implémentation que nous avons étudiée, c'est-à-dire qu'il cherche à détecter toute la structure et ses différentes étapes, qui doivent donc apparaître de façon explicite et dans l'ordre attendu. Un tel motif est donc peu incertain (peu de faux positifs).

Toutefois, il serait souhaitable de disposer également d'un motif plus « tolérant » qui serait à même de détecter ce qui s'approche suffisamment de la structure de l'implémentation. En particulier, à des niveaux importants d'optimisations, certaines modifications effectuées par les compilateurs peuvent venir bouleverser cette structure claire (par exemple lorsque les opérations sur les entiers 32 bits sont packées sur des registres 64 bits).

Il est donc possible de concevoir une variante « heuristique » de notre motif, plus générique et qui serait plus tolérante face aux modifications. Pour ce faire, on part de l'idée que le compilateur est obligé de respecter relativement l'ordre et la sémantique des opérations au début du premier tour (car il faut d'abord calculer les premiers résultats avant de pouvoir mutualiser les opérations). À partir de ce postulat, on modifie le motif pour matcher nécessairement ces éléments en particulier, alors que le match des autres tours devient lui optionnel.

Le motif ainsi produit (également disponible dans le dépôt du projet) est beaucoup plus tolérant et peut détecter des implémentations proches de celle étudiée (notamment lorsqu'elle subit des modifications importantes à la compilation), mentionnons d'ailleurs qu'il détecte correctement l'implémentation en assembleur x86_64 contenue dans LibreSSL que nous avons abordée plus haut. En contrepartie, il est très légèrement plus incertain : la détection traduit la présence d'un élément proche d'une implémentation de MD5, mais il y a une possibilité qu'il s'agisse d'un faux positif (nous avons observé quelques cas où cette heuristique réagit à la présence de MD4).

3.4 Synthèse

Nous avons été en mesure de concevoir des motifs efficaces pour la détection d'une implémentation choisie de l'algorithme MD5, sans reposer sur les constantes qu'il utilise. Cette détection *implementation-based* est par ailleurs relativement tolérante face à des variantes ou à des modifications du compilateur. Voici comment il est possible d'utiliser ces motifs en ligne de commandes sur les binaires qui se trouvent dans le répertoire courant :

Terminal

```
$ grap --show-all --quiet ../md5_crypto_v1.0.grapp *
shasum.grapcfg (4903)
sha256sum.grapcfg (6969)
sha512sum.grapcfg (8134)
md5sum.grapcfg (4145) - md5_crypto_heuristic (1)
x64_libcrypto.so.41.1.0_clang_03.grapcfg (172293) - md5_crypto_heuristic (1)
x64_libcrypto.so.37.0.0_03.grapcfg (191307) - md5_crypto_heuristic (1)
x64_libcrypto.so.37.0.0_00.grapcfg (318160) - md5_crypto_full (1), md5_crypto_
heuristic (18)
x86_libcrypto.so.37.0.0_00.grapcfg (346416) - md5_crypto_full (1), md5_crypto_
heuristic (18)
Steam.exe.grapcfg (475343) - md5_crypto_heuristic (17)
```

Sur cet exemple, on peut voir, après le nom du binaire et le nombre de nœuds dans son CFG, les motifs qui ont été détectés.

Nous proposons enfin d'achever cette étude de cas avec un exemple concret d'utilisation des motifs conçus : la plateforme de distribution de jeux vidéo Steam est connue pour utiliser OpenSSL, il est donc probable que l'on puisse retrouver dans ses binaires une implémentation de MD5 livrée avec cette bibliothèque (ce qui ne pose donc pas de problème de sécurité en soi). Sur Windows, le programme exécutable **Steam.exe** (version 4.28.51.7) est strippé, retrouver la procédure où cet algorithme est mis en œuvre pourrait donc prendre du temps sans recourir à une technique de détection. En l'espèce, notre motif détecte correctement l'implémentation, ce qui permet de localiser immédiatement la procédure en question.

CONCLUSION

Il est assez fréquent que des vulnérabilités soient découvertes au sein d'implémentations d'algorithmes cryptographiques. Il arrive également parfois que ces algorithmes eux-mêmes deviennent obsolètes. La détection automatique de tels algorithmes, dont l'utilisation serait susceptible de causer de graves problèmes de sécurité, apparaît donc comme quelque chose d'utile pour l'analyse de programmes exécutables.

Nous avons montré que l'utilisation de grap pour réaliser cette détection au niveau des implémentations est possible et présente des avantages intéressants qui rendent cette approche complémentaire aux autres techniques usuellement utilisées. En particulier, plutôt que de raisonner sur l'algorithme ou sur ses constantes, on se place au niveau de l'implémentation, qui est également potentiellement vulnérable. ■

RÉFÉRENCES

- [AESa] A. Bogdanov, D. Khovratovich et C. Rechberger, « Biclique Cryptanalysis of the Full AES », Lee, D.H. (ed.) ASIACRYPT 2011, LNCS, vol. 7073, Springer, 2011. <https://eprint.iacr.org/2011/449>
- [ARGON2] Argon2 : <https://github.com/p-h-c/phc-winner-argon2>
- [PHC] Password Hashing Competition : <https://password-hashing.net/>
- [EMV] S. Bouzeffrane, « La norme Europay Mastercard Visa », CEDRIC – CNAM : https://cedric.cnam.fr/~bouzeffra/cours/Cartes_Bouzeffrane_EMV_nov2009.pdf
- [SHA2005] X. Wang, Y. Lisa Yin et H. Yu, « Finding Collisions in the Full SHA-1 », Advances in Cryptology – CRYPTO 2005, Springer, 2005.
- [SHA-Sch] B. Schneier, « Cryptanalysis of SHA-1 », 2005, https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html.
- [SHattered] SHattered (2017) : <https://shattered.io/>
- [DECDRBG1] É. Gaspar (gapz), « Surveillance généralisée : Dual_EC_DRBG, 10 ans après », *MISC* n°84, mars 2016.
- [DECDRBG2] Dual_EC_DRBG backdoor : a proof of concept (Aris' Blog), 2013 : <https://blog.0xbadc0de.be/archives/155>
- [GRAP-git] A. Thierry, L. Benedetti et J. Thieuleux, grap : <https://github.com/AirbusCyber/grap>
- [GRAP-REcon] A. Thierry et J. Thieuleux, « grap : define and match graph patterns within binaries », REcon Brussels 2017, janvier 2017.
- [GRAP-Sthack] A. Thierry et J. Thieuleux, « Recherche de motifs de graphes dans un exécutable avec grap », Sthack 2017, avril 2017.
- [GRAP-Grehack] L. Benedetti, A. Thierry et J. Francq, « Detection of cryptographic algorithms with grap », GreHack 2017, novembre 2017.
- [YARA] YARA : <https://plusvic.github.io/yara/>
- [Signsrch] Signsrch : <http://aluigi.altervista.org/mytoolz.htm>
- [FindCrypt2] FindCrypt2 : <http://www.hexblog.com/?p=28>
- [IDAScope] IDAScope : https://bitbucket.org/daniel_plohmann/simplifire.idascope/
- [Sibyl] Sibyl : <https://github.com/cea-sec/Sibyl>
- [Aligot] Aligot : <https://code.google.com/archive/p/aligot/>
- [MD2004] X. Wang, D. Feng, X. Lai et H. Yu, « Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD », Cryptology ePrint Archive, Report 2004/199, 2004. <https://eprint.iacr.org/2004/199>
- [LibreSSL] LibreSSL : <https://www.libressl.org/>



3

IOT

À découvrir dans cette partie...

FIRMWARE / EMBEDDED / IOT / EXPLOIT



Toolkit 4.0 : revue des outils pour l'exploitation pour l'IoT

Il n'est plus nécessaire d'affirmer que le « S » de « IOT » signifie « security ». Il est cependant nécessaire d'utiliser une grande gamme d'outils pour faire de la recherche de vulnérabilités dans ce domaine, tant la diversité des cibles est grande. Que ce soit pour extraire les firmwares, les reverser ou pour les analyser, découvrez ici un ensemble d'outils dédiés à l'IOT. p.92



FIRMWARE / EMBEDDED / IOT / EXPLOIT

TOOLKIT 4.0 : REVUE DES OUTILS POUR L'EXPLOITATION POUR L'IOT

Hugo GENESSE

Les petits dispositifs embarqués font maintenant partie de notre vie quotidienne, mais quels sont les outils spécialisés nécessaires pour tester leur sécurité ?

Avec plus de 4000 entreprises présentes au Consumer Electronic Show 2018 [1] où il a, entre autres, été possible de voir une couche pour bébé connectée à Internet [2], il est facile de prévoir que nous serons bientôt inondés de bidules électroniques connectés dans les prochaines années. Avec un marché qui pourrait devenir de plus en plus compétitif comme celui-ci, on peut aussi se douter que la sécurité pourrait parfois être mise à l'arrière de la liste des priorités pour permettre un développement plus rapide et moins coûteux. En sachant cela, comment peut-on analyser cette pléiade d'objets connectés en tant que chercheur en sécurité informatique ?

Il existe un certain nombre de ressources sur la rétro-ingénierie et l'exploitation de ce type d'objets qui se focalisent sur la partie matérielle, par exemple en identifiant les branchements UART ou JTAG [3] et sur l'analyse de ces protocoles. Cependant, moins d'articles traitent des outils de recherche de vulnérabilités et de l'analyse de la partie firmware, c'est-à-dire le code en lecture seule sur les puces de ces appareils, présentes sur ce type de plateformes qui sont beaucoup moins communes que les architectures d'Intel ou même ARM. Cet article a donc pour but de faire une revue des outils existant pour la recherche de vulnérabilités sur les dispositifs de l'Internet des objets au niveau du firmware en traitant de l'émulation, du débogage, de l'extraction, de la rétro-ingénierie et des autres outils qui peuvent s'avérer très utiles, mais qui s'inscrivent moins facilement dans ces catégories.

Afin de rendre l'exercice plus concret, je vous invite à suivre avec le firmware qui sera utilisé tout au long de l'article : le firmware de la caméra de surveillance TP-LINK NC230 qui est toujours en vente au moment d'écrire ces lignes. Le firmware est disponible sur le site web du fabricant et sur un nouveau repo GitHub [4] qui tentera de faciliter l'accès aux différents firmwares qui seront introduits plus tard dans l'article. Sans plus attendre, partons à l'aventure !

1. RECONNAISSANCE

1.1 FCC ID

Tout d'abord, il est souvent judicieux d'utiliser les documents officiels pour avoir une idée des fonctionnalités présentes dans l'appareil. Un truc très intéressant est de faire une recherche du « FCC ID » trouvé sur l'appareil de façon obligatoire lorsque l'appareil émet des ondes et est vendu aux États-Unis. En le cherchant sur Internet, on peut ainsi trouver beaucoup d'informations comme des photos haute résolution qui peuvent servir à identifier un branchement par exemple. Pour notre caméra NC230, aucun numéro n'est affiché sur l'appareil ou sur la boîte, mais une recherche du modèle sur le site fccid.io [5] nous permet d'obtenir des photos du circuit imprimé de la caméra et ses composantes :

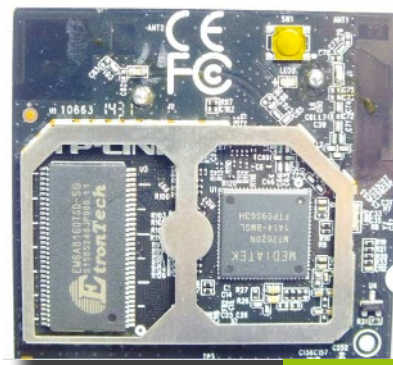


Figure 1

Il est aussi souvent très payant de simplement regarder les versions des logiciels utilisées dans le firmware, ce qui pourrait vous permettre d'utiliser une vulnérabilité déjà divulguée dans vos chaînes d'exploits. Par exemple, les versions de Linux et Busybox [6] (un ensemble d'utilitaires UNIX de base focalisé sur l'économie d'espace et donc très utilisé sur les systèmes embarqués) sont rarement à jour et peuvent vous donner un très bon départ pour très peu d'efforts.

1.2 FirmwareDB

Les firmwares de plusieurs appareils sont publics sur le site de leur fabricant. Cela dit, chaque site adopte sa propre façon de le publier, ce qui peut parfois être ardu à trouver ou tout simplement agaçant lorsque vous souhaitez simplement vous initier à la recherche de vulnérabilités sur des architectures différentes, et ce, sur de vrais produits toujours en vente. Dans cet ordre d'idée, je profite de cet article pour tenter de partir d'une collection publique et organisée de firmwares publics afin de faciliter la recherche et l'expérimentation. Pour l'instant, il s'agit d'un entrepôt GitHub où les firmwares qui seront touchés au cours de cet article seront accessibles pour tous [4].

2. EXTRACTION

2.1 Binwalk

Ensuite, salissons-nous les mains en les plongeant dans le firmware. Le premier outil que nous verrons est un incontournable : binwalk [7]. Cet outil en python permet de déterminer le contenu de l'image brute du firmware à l'aide de signatures inspirées du format de **libmagic** (bibliothèque issue de l'utilitaire file). Binwalk possède aussi plusieurs fonctions autres que l'extraction. Premièrement, il est possible de seulement faire un scan sans extraction. De plus, on peut effectuer une analyse d'entropie avec l'option **-E** sur le firmware ciblé pour identifier des parties intéressantes et qui pourraient avoir été manquées par binwalk ou tout simplement commencer l'analyse d'un format inconnu. Voici le graphique de l'entropie du firmware du NC230 :

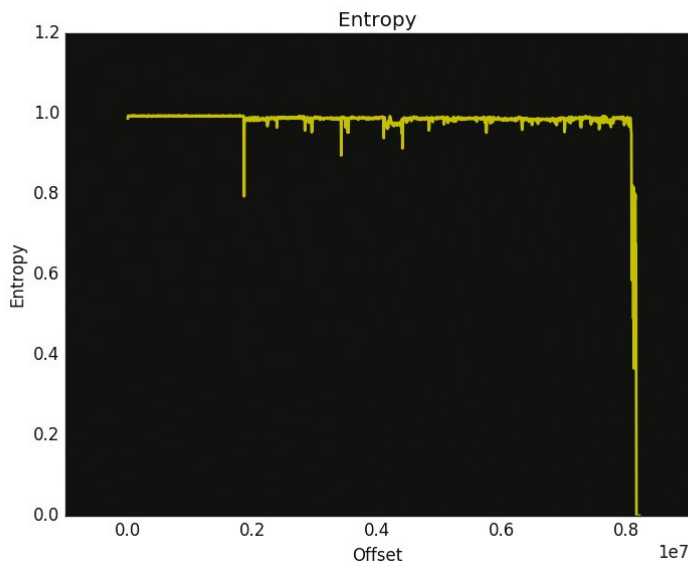


Figure 2

```
svieg@primarch ~/w/s/F/T/c/NC230> binwalk -e NC230_1.2.1_Build_170515_Rel.9A7F66_2017-05-15_16.03.14.bin
```

DECIMAL	HEXADECIMAL	DESCRIPTION
304	0x130	uImage header, header size: 64 bytes, header CRC: 0xB12C1B0D, created: 2017-05-15 16:03:14
368	0x170	LZMA compressed data, properties: 0x5D, dictionary size: 33554432 bytes, uncompressed size: 1866750 bytes
1866750	0x1C7BFE	JFFS2 filesystem, little endian

Par la suite, le *binary diffing* est aussi possible avec cet outil afin de comparer les différentes versions disponibles (patch diffing) d'un firmware, ce qui pourrait être utile afin de trouver des n-days ou analyser une mise à jour inefficace et qui peut donc être contournée. Par ailleurs, les utilisateurs avancés sont aussi gâtés, car il est aussi possible de scripter binwalk à l'aide de son API ainsi que d'ajouter ses propres signatures et plugins.

Maintenant que vous connaissez mieux l'outil, regardons ce que nous obtenons avec le firmware du NC230. Un simple scan identifie des informations importantes comme le fait que ce soit une image Linux MIPS et que nous avons un système de fichiers JFFS2. Veuillez noter que l'option **-t** ici utilisée n'est que pour le formatage de la sortie et n'affecte pas le scan (Figure 3).

Puisque nous avons un système de fichiers à investiguer, relançons la même commande avec l'option **-e** qui nous permettra d'extraire les différents fichiers que binwalk a trouvé. Nous avons donc maintenant une arborescence de fichiers qui ressemble à :

Terminal

```
$ ls *
fs_1:
bin/ config/ etc/ lib/ sbin/ share/ www/
fs_2:
Region Wireless.conf workmod.conf
```

Excellente nouvelle pour un reverser : nous avons enfin des binaires MIPS se trouvant dans le système de fichier fs_1 !

2.2 Sasquatch et Jefferson

Les systèmes de fichiers comme squashfs et JFFS2 sont utilisés dans le domaine de l'embarqué et posent quelques problèmes par rapport à un système usuel. Ils sont optimisés pour la taille et donc compressés et ils sont écrits sur des puces flash qui peuvent être en lecture seulement. Cela veut donc dire qu'il est impossible de transférer des outils nécessaires à notre tâche sur ces systèmes de fichiers. Heureusement, l'auteur de binwalk a aussi écrit des outils permettant de décompresser ces systèmes de fichiers facilement et qui ont été conçus pour être incorporés dans binwalk. Ces outils sont : sasquatch [8] pour squashfs et jefferson [9] pour JFFS2. Dans le cadre de l'exploitation, vous pouvez vérifier la présence de systèmes de fichiers temporaires comme tmpfs qui vous permettra de stocker vos programmes d'analyse et exploits jusqu'à ce que l'appareil soit mis hors tension.

3. REVERSING

Maintenant que nous avons une première idée de ce que contient notre firmware, il est temps de passer à l'analyse, et nous présenterons ici quelques outils qui peuvent nous aider.

Figure 3

```
-15 07:36:28, image size: 1866382 bytes, Data Address: 0x80000000, Entry Point: 0x8000C310, data CRC: 0
nel Image"
pressed size: 4812352 bytes
```

3.1 JEB

3.1.1 Architectures et fonctionnalités

Cet outil est une plateforme de reverse engineering contenant un désassembleur (ARM, MIPS, AVR), un décompilateur (dalvik pour android, ARM, MIPS, AVR et autres architectures bientôt disponibles), des outils de débogage, déobfuscation et autres domaines du reverse engineering. Il est aussi possible d'accéder à toutes les fonctionnalités utilisées par le client via l'API, comme la représentation intermédiaire utilisée par l'engin de décompilation par exemple. Il est possible d'ajouter des architectures à l'aide de processor plugins. Pour ce qui est du débogage, il est possible de se connecter à une instance de gdbserver ou lldb-server et de suivre le flot d'exécution dans JEB en inspectant la valeur des registres et autres fonctions normalement disponibles sur un débogueur. Trois différentes démos sont disponibles avec 85% des fonctionnalités sur le site web : décompilateur dalvik + ARM, MIPS et seulement ARM [10].

3.1.2 Pomme

Une des particularités du reversing de firmware est d'avoir un nombre important de binaires et, selon mon expérience, qui peuvent même avoir des noms qui vous ne donnerons aucun indice sur leur fonction. Pour pallier à cela, j'ai écrit un plugin JEB afin d'identifier les binaires utilisant certaines fonctions qui sont intéressantes d'un point de vue d'un chercheur en sécurité. Le plugin s'appelle Pomme et se configure avec un fichier contenant une liste des noms complets ou en partie des fonctions comme-ci :

```
svieg@primarch ~/t/JEB> cat functionList.json
["cpy", "gets"]
```

Figure 4

Pomme trouvera ainsi toutes les instances des fonctions '**strcpy**', '**memcpy**', '**fgets**' par exemple et vous fera un tableau. Prenons tous les binaires du dossier **fs_1/bin** pour fin d'illustration :

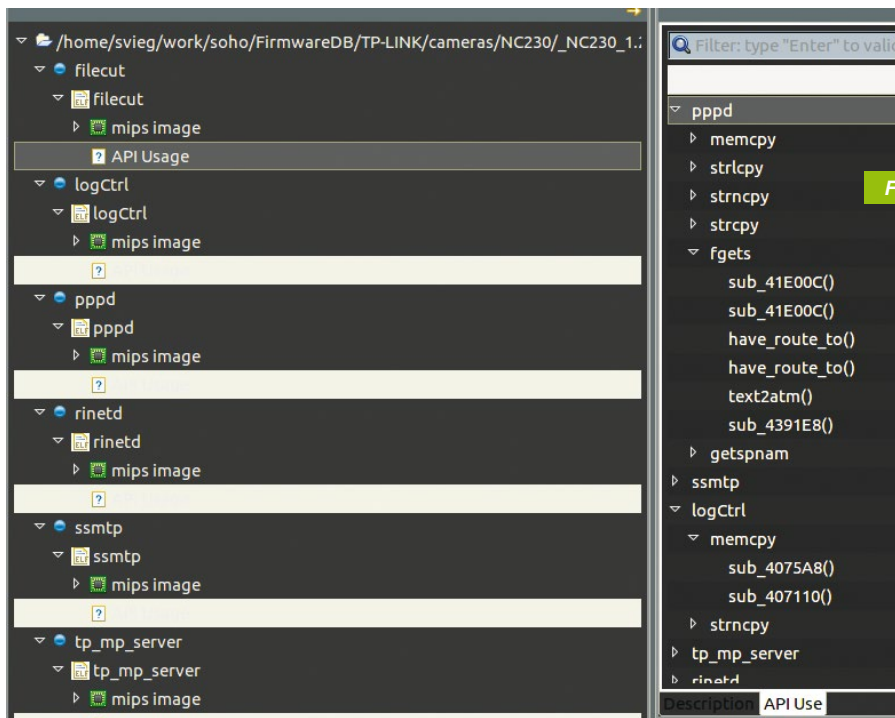


Figure 5

Pomme est disponible sur GitHub [11] et n'attend que vos suggestions !

3.1.3 Décompilation et désassemblage

Nous avons maintenant une meilleure idée de quels binaires pourraient contenir des vulnérabilités qui seraient plus faciles à trouver et, puisqu'il est toujours important de maximiser le nombre de bugs par quantité d'effort, commençons par ceux faisant des `memcpy/strcpy` !

La fonction **CheckMessageFromMMT** de `fs_1bin/tm_mp_server` semble intéressante, car elle sous-entend une entrée possiblement contrôlable par un utilisateur avec un message fabriqué par ce dernier. Également, le binaire est un serveur comme son nom l'indique. Regardons son graphe de flot de contrôle et le pseudo-code généré par JEB :

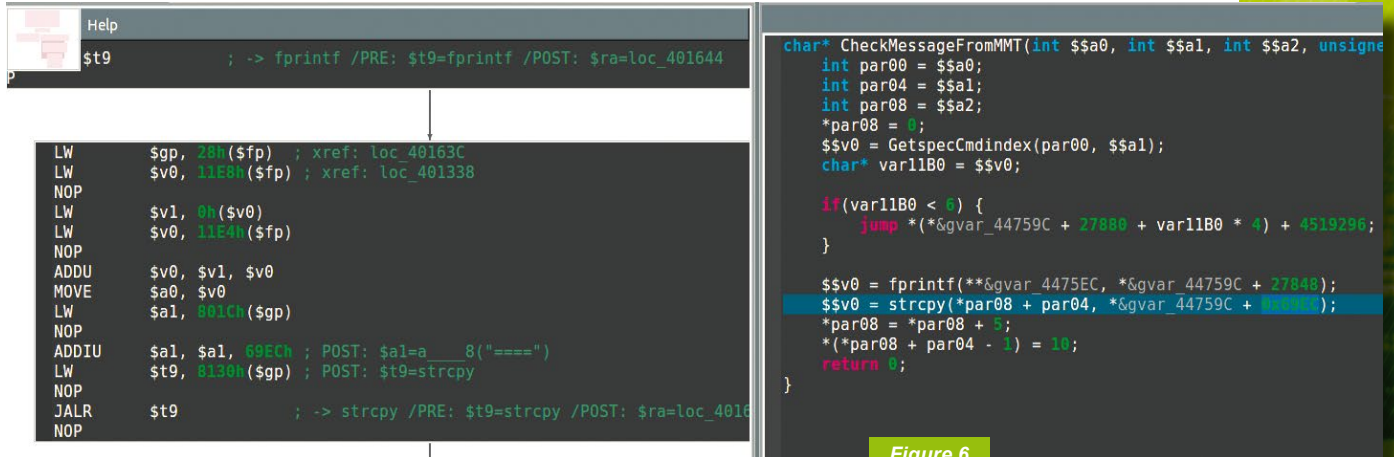


Figure 6

En cliquant sur **gvar_44759C**, on peut constater que celle-ci possède la valeur de **0x400000** et regardons donc ce qui se trouve à l'adresse **0x4069EC** :

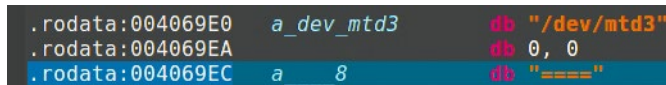


Figure 7

Ce n'est malheureusement pas contrôlable par un attaquant dans ce cas-ci. De plus, Il est important de noter qu'aucune vulnérabilité non résolue ne sera présentée dans cet article.

Une autre vulnérabilité malheureusement commune dans le monde de l'Internet des objets est le *OS command injection* qui peut aussi être assez simple à trouver. Dans ce type de vulnérabilité, certains scripts bash peuvent par exemple être appelés par l'interface web ou même le serveur http peut faire des appels à la fonction **system** ou **execve** pour faire un ping avec une adresse IP fournie par l'utilisateur. Ces vulnérabilités peuvent être trouvées facilement en trouvant les références aux fonctions comme **system** et **execve** et peuvent vous donner de l'exécution de code en quelques minutes !

3.1.4 Débogage

JEB vous permet de vous connecter à un gdbserver ou un lldb-server comme mentionné précédemment. Vous n'avez qu'à pointer le débogueur vers l'IP de votre gdbserver ainsi que son port et vous pourrez suivre l'avancement du programme ainsi que la valeur des registres et ce qui se trouve sur le stack en restant dans la même interface.

3.1.5 Autres formats

Puisqu'il n'y a pas que des binaires dans un firmware, il est intéressant de jeter un coup d'œil aux autres fichiers comme les fichiers HTML qui constituent l'interface web d'administration qui est un bon vecteur

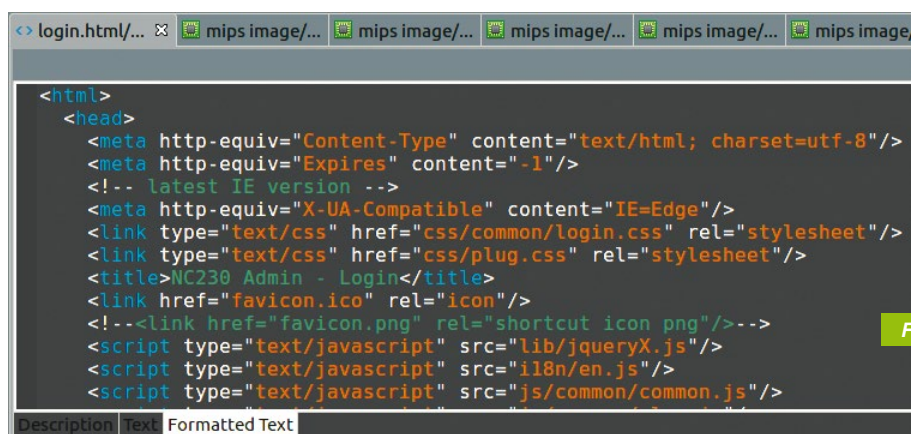


Figure 8

d'entrée pour un attaquant, car elle peut interagir avec des binaires qui contiennent des vulnérabilités ou en contenir elle-même directement comme l'histoire nous l'a souvent prouvé [12].

Dans notre cas, le NC230 possède une de ces interfaces web et, en inspectant les fichiers sources du portail, on peut identifier l'utilisation de fichiers avec l'extension fcgi qui signifie fast cgi. Cependant, ces fichiers ne sont pas dans le système de fichiers que nous avons extrait avec binwalk. En passant les chaînes de caractères de chaque programme à la loupe, on verra que certains de ces fichiers sont référencés dans le binaire lui-même. C'est le cas pour **fs_1/sbin/ipcamera** qui est responsable d'une bonne partie des fonctionnalités du portail. Cela nous permet donc de rendre notre recherche encore plus fructueuse en nous assurant un chemin entre notre binaire et une interface possiblement disponible sur le LAN de la caméra ou même publiquement sur Internet.

3.1.6 Android

JEB est à l'origine un outil pour l'analyse d'applications Android et, étant donné que l'utilisation d'app à des fins d'interfaces d'administration est fréquente, cela peut aussi être un aspect intéressant pour un chercheur en sécurité ciblant un appareil embarqué.

3.2 Hopper

3.2.1 Architectures et fonctionnalités

Malheureusement, Hopper ne supporte pas le MIPS, mais reste un outil très utilisé pour le reversing sur ARM, qui est une architecture très utilisée dans le monde de l'IoT. Prenons donc comme exemple la version ARM de busybox pour cet outil. Il est possible d'ajouter des architectures dans Hopper avec la SDK et des projets pour intégrer les architectures MIPS, m68k et Amiga ont été développés par des membres de la communauté.

Le cas d'utilisation est centré sur les plateformes d'Apple, ce qui peut être un avantage si vous ciblez un appareil ayant une application iOS comme interface d'administration. Hopper est aussi capable de faire du débogage avec une interface GDB, comme les tutoriels de Oxabad1dea le prouvent [13]. Une démo est aussi disponible [14], mais cette version n'est pas capable de faire du débogage, est limitée à des sessions de 30 minutes et rien ne peut être sauvegardé.

3.2.2 Décompilation et désassemblage

Comme on peut le voir, Hopper est capable de faire le désassemblage (la décompilation est aussi possible) de binaires d'architectures ARMv6 et v7 sans difficulté.


```

int sub_8c94(int arg0, int arg1, int arg2) {
    r2 = arg2;
    r4 = arg1;
    r1 = "busybox";
    r5 = arg0;
    if (sub_9354(arg0, r1) == 0x0) goto loc_8f6c;

loc_8cac:
    r0 = *(r4 + 0x4);
    r6 = r4 + 0x4;
    if (r0 != 0x0) goto loc_8d70;

loc_8cbc:
    r4 = sub_a138(0x2);
    sub_cf8d4(0x1);
    sub_a15c("BusyBox v1.26.2 (2017-01-11 08:43:16 UTC)");
    sub_a15c(" multi-call binary.\n");
    sub_a15c(0xee730);
    r6 = 0xf084a;
    r7 = r4 - 0x1;
    r4 = 0x0;
    while (*(int8_t *)r6 != 0x0) {
        r5 = sub_81d0(r6) + 0x2;
        if (r4 >= r7 - r5) {
            sub_a15c(0xee94c);
            sub_a15c(0xfef74);
            r4 = 0x6;
        }
        else {
            if (r4 == 0x0) {
                sub_a15c(0xfef74);
                r4 = 0x6;
            }
        }
    }
}

```

Figure 9

3.2.3 Autres formats

Hopper ne supporte que l'importation de binaires pour l'instant.

3.3 Binary Ninja

3.3.1 Architectures et fonctionnalités

Binary Ninja désassemble un bon nombre d'architectures comme sa FAQ [15] l'indique et donne également la possibilité d'ajouter des architectures avec des plugins. Binary Ninja reste un outil d'analyse purement statique pour l'instant et il n'est donc pas possible de déboguer à l'intérieur de cet outil. Une démo étant capable de désassembler du x86 et du ARMv7 est disponible sur son site [16].

3.3.2 Décompilation et désassemblage

Binary Ninja ne décompile aucune architecture pour l'instant, mais possède 2 niveaux de représentation intermédiaire, appelée le *medium-level intermediate language*, ressemblant à ceci pour la même fonction montrée précédemment avec JEB :

```

20 @ 00401648  int32_t* $v0_17 = arg_8
21 @ 00401650  int32_t $v1_6 = [$v0_17].d
22 @ 00401654  int32_t* $v0_18 = arg_4
23 @ 0040165c  void* $v0_19 = $v1_6 + $v0_18
24 @ 00401660  void* $a0_18 = $v0_19
25 @ 00401678  strcpy($a0_18, 0x4069ec) {"==="}
26 @ 00401684  int32_t* $v1_7 = arg_8

```

Figure 10

Il est intéressant de noter que Binary Ninja fait aussi le calcul du global pointer sur MIPS pour être capable de savoir quelle fonction est appelée (MIPS utilise ce pointeur global pour avoir du code indépendant de sa position). On n'a donc pas du pseudocode, mais cela peut donner une idée plus rapidement du but de la fonction que simplement avec l'assembleur, surtout si ce n'est pas une architecture dont vous êtes particulièrement à l'aise.

3.3.3 Autres formats

Comme IDA et radare2 (décrits plus bas), il est seulement possible d'analyser des binaires et d'ajouter des architectures de processeurs. Binary Ninja ne supporte pas non plus les formats d'applications mobiles.

3.4 IDA Pro

3.4.1 Architectures et fonctionnalités

IDA Pro, qui ne nécessite plus d'introduction, possède la plus grande liste des architectures supportées [17], ce qui peut être intéressant si vous pensez faire face à des architectures très obscures. Aussi, il est aussi possible d'ajouter des architectures qui ne sont pas supportées officiellement à l'aide de plugins. Pour le débogage d'appareils embarqués, il est possible de démarrer un gdbserver sur l'appareil et de s'y connecter avec IDA Pro pour faire une analyse dynamique en gardant les informations obtenues en analysant le binaire. La version 7.0 est disponible gratuitement sur leur site [18], mais n'inclut pas la décompilation avec Hex-Rays et est limitée à certaines architectures.

3.4.2 Décompilation et désassemblage

Dans notre cas, aucun décompilateur MIPS n'est disponible pour Hex-Rays donc voici une partie du Control Flow Graph pour la fonction **CheckMessageFromMMT** qui a servi d'exemple pour les outils précédents :

```
loc_401648:
lw      $v0, 0x11E0+arg_8($fp)
nop
lw      $v1, 0($v0)
lw      $v0, 0x11E0+arg_4($fp)
nop
addu    $v0, $v1, $v0
move    $a0, $v0
li      $a1, 0x400000
nop
addiu   $a1, (asc_4069EC - 0x400000) # "===="
la      [ra], strcpy
nop
jalr    [ra]; strcpy
nop
lw      $gp, 0x11E0+var_11B8($fp)
lw      $v1, 0x11E0+arg_8($fp)
lw      $v0, 0x11E0+arg_8($fp)
nop
```

Figure 11

On peut remarquer qu'IDA fait aussi le calcul du global pointer correctement pour vous indiquer les fonctions appelées.

3.4.3 Débogage

IDA Pro vous fournit des débogueurs qui peuvent se connecter à des gdbserver ou leur propre serveur pour déboguer Linux ainsi que quelques options pour les émulateurs avec notamment le plugin uEmu [19] basé sur Unicorn Engine qui vous permet de déboguer du x86, x64, ARM, ARM64, MIPS, MIPS64.

3.4.4 Autres formats

IDA Pro ne supporte pas l'analyse d'autres formats que les architectures CPU.

3.5 Radare2

3.5.1 Architectures et fonctionnalités

Radare2 possède un grand nombre d'architectures possibles et donne aussi la possibilité d'ajouter des architectures si vous en avez le besoin. Avec Cutter, r2 a maintenant une interface graphique pour ceux n'étant pas adeptes des interfaces purement en ligne de commandes. Radare2 permet aussi le débogage avec son propre engin. Radare2 [20] et Cutter [21] sont des logiciels libres.

3.5.2 Décompilation et désassemblage

Radare2 a maintenant son propre moteur pour produire du pseudocode qui se trouve dans la vue du même nom sur Cutter. Aussi, avec le plugin pour snowman [22], r2snow, Radare2 peut vous donner un pseudocode C des fonctions du binaire dans un fichier texte externe à l'interface principale ou en commentaire.

On peut voir ici que Cutter et radare2 sont toujours en développement et peuvent donc contenir des bugs, mais ils donnent une bonne alternative gratuite qui s'améliorera dans les prochaines années. Ici, les deux branches ont presque 0x400 octets d'écart et semblent causer problème dans leur analyse comme le commentaire « unlikely » le dénote. En réalité, ce saut existe vraiment, alors il ne devrait pas y avoir de break dans ce cas.

3.5.3 Déboguage

Radare2 est aussi capable de déboguer avec leur propre engin ou avec un gdbserver comme IDA Pro et JEB.

3.5.4 Autres formats

Radare2 peut analyser plusieurs formats incluant le dalvik pour les applications Android, certains systèmes de fichiers ainsi que des images EnCase pour le forensic.

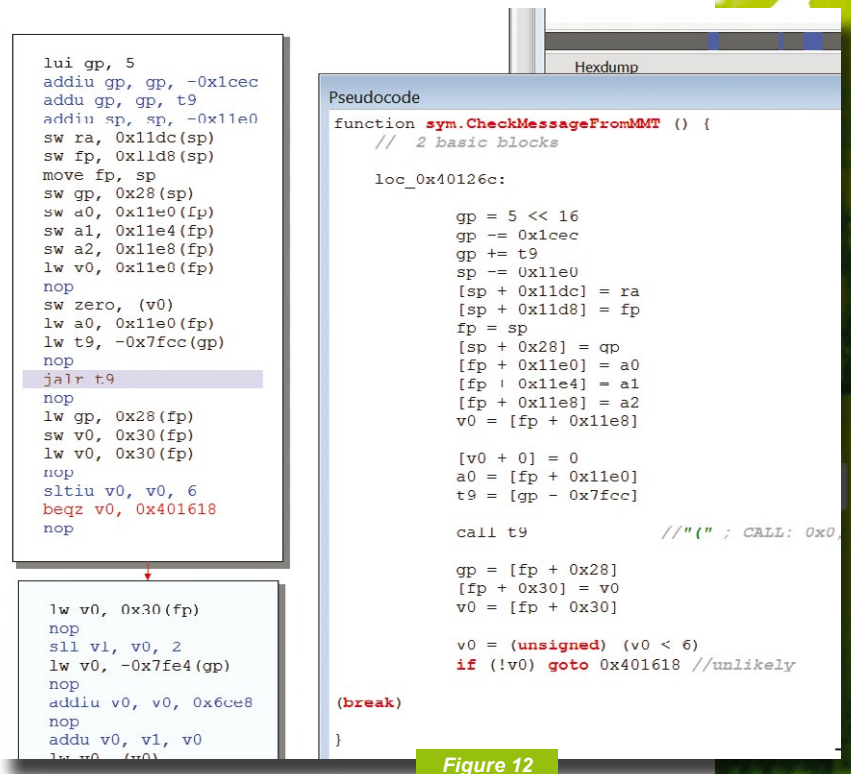


Figure 12

4. ÉMULATION

4.1 QEMU-user

Il est possible d'émuler un binaire d'une architecture différente de l'hôte avec qemu-user. En travaillant avec des appareils embarqués, vous comprenez à quel point cela peut-être utile. On peut donc prendre les binaires directement du firmware, tester et déboguer leurs fonctionnalités sans avoir à posséder l'appareil

en question. Une option pratique de QEMU-user est celle activant un gdbserver déboguant le programme émulé sur le port voulu en passant l'argument **-g** à qemu-user. Avec un chroot dans le système de fichiers de votre appareil IoT ciblé, il est possible d'avoir un programme fonctionnel sans trop d'efforts. Malheureusement, le fait que le code est émulé change un bon nombre de choses qui peuvent être importantes pour un développeur d'exploit et dont vous êtes habitués d'avoir. Il est alors parfois préférable d'émuler un système complet. Pour le NC230, les programmes sont linkés de façon dynamique, ce qui nous oblige à avoir toutes les bibliothèques utilisées par le programme cible dans le système de fichiers du firmware. Pour ce faire, commencez par copier la version statique de l'émulateur approprié, **qemu-mipsel-static** pour le NC230, qui devrait se trouver dans votre répertoire **/usr/bin** dans la racine du système de fichiers ciblé. Ensuite, vous pouvez effectuer un chroot sur la racine du système de fichiers du NC230 en exécutant l'émulateur sur le programme choisi, qui est **sbin/ipcamera** dans cet exemple, comme ceci :

Terminal

```
sudo chroot . ./qemu-mipsel-static sbin/ipcamera
```

Vous aurez alors une erreur de librairie manquante. En effet, la version du firmware du NC230 disponible sur Internet n'inclut pas les bibliothèques libc nécessaires pour ces programmes. Il faut alors copier les bonnes bibliothèques compilées avec la même version de la libc dans le répertoire **fs_1/lib** pour émuler correctement les programmes de cet appareil.

4.2 Binfmt

Un outil intéressant lorsque vous utilisez QEMU-user est binfmt. Cet outil s'occupe de lancer l'émulateur qemu-user approprié en analysant l'architecture du binaire. Vous pouvez alors travailler comme si votre cible était de la même architecture que votre hôte sans vous poser de questions.

4.3 QEMU system

QEMU-system vous permet d'émuler un système complet comme si vous étiez dans une machine virtuelle habituelle avec un userspace qui vous est familier. On peut ainsi utiliser un package manager, avoir une sortie crédible pour les mappings en mémoire avec gdb et la possibilité de tracer des programmes avec des utilitaires comme ltrace. De plus, grâce à hugsy, vous n'avez plus besoin d'utiliser les vieilles images d'Aurel32, mais vous pouvez utiliser les versions modernes de Debian pour vos systèmes émulés [23]. Il va sans dire que le système émulé sera plus lent qu'un système roulant directement sur le métal mais, avec assez de mémoire vive, il est facile de s'en tirer pour déboguer vos exploits ou vos programmes cibles. Les architectures présentement disponibles contiennent notamment du ARMv6, ARM64, MIPS, MIPS64, POWERPC et SPARC. Les images viennent même avec des binaires exploitables afin que vous puissiez vous pratiquer facilement à l'exploitation sur ces architectures moins traditionnelles. Hugsy a aussi écrit quelques tutoriels sur le processus et les difficultés rencontrées et leur solution pour vous aider à faire vos propres images si vous en avez le besoin ! Pour notre exemple du NC230, vous pouvez donc copier le système de fichiers décompressé dans votre machine virtuelle émulée et utiliser chroot pour utiliser les bonnes bibliothèques comme avec QEMU-user.

4.4 Firmadyne

Ceci est un projet utilisant qemu-system pour émuler un firmware complet [24]. On peut donc le voir comme un wrapper autour de qemu-system avec des bibliothèques utiles pour simuler des fonctions qui se retrouvent parfois dans les systèmes embarqués comme une puce de mémoire non volatile (NVRAM) qui peut être un obstacle au bon fonctionnement de l'émulation. Libnvrn définit donc des hooks aux accès à ce type de puce pour éviter de faire planter le firmware. En plus de cette bibliothèque, Firmadyne contient un extracteur, basé sur binwalk, des kernels modifiés pour faciliter l'instrumentation de ces derniers, un scraper pour télécharger plus facilement les firmwares et une console série pour faciliter le débogage. Cela dit, soyez avisés que l'émulation d'un système complet peut s'avérer ardue pour le bénéfice que cela peut vous donner dans certains cas. Dans le cas de la caméra NC230, nous n'avons pas de mémoire NVRAM et donc l'utilisation de Firmadyne n'est pas nécessaire pour l'émulation de programmes pour cette cible.

4.5 Avatar2

Avatar2 [25] va encore un peu plus loin dans ces capacités et vous permet d'instrumenter avec un module Python avec une fine granularité. On peut donc ajouter des mapper des plages mémoires avec du contenu spécifique, mettre des breakpoints, injecter du code assembleur et faire le traitement désiré avec leur API. Avatar2 a plusieurs targets possibles incluant : QEMU standard, PANDA qui est une version modifiée de QEMU avec des fonctionnalités utiles pour le reverse engineering, OpenOCD qui permet de se connecter à une interface JTAG pour le débogage et GDB pour la même chose, mais avec un gdbserver sur un port série ainsi qu'angr, la plateforme d'analyse de programmes d'UCSB permettant l'exécution symbolique. La deuxième version est complètement découplée des émulateurs et il est donc possible de définir votre propre engin d'émulation. Malheureusement, Avatar2 ne supporte que les architectures ARM, x86 et x86_64 et pas le MIPS, ce qui empêche son utilisation pour l'instrumentation du firmware du NC230.

5. FUZZERS

Même s'il est souvent difficile d'obtenir une émulation fiable qui correspondra au comportement réel de l'appareil étudié en émulant un seul binaire, il est important de savoir quelles sont nos options de fuzzers disponibles lorsque c'est le cas.

5.1 AFL-QEMU et AFL-other-arch

Si vous avez de l'expérience avec le fuzzing sous Linux, vous connaissez AFL. Même si l'instrumentation à la compilation reste un avantage important d'AFL, il est possible de faire du fuzzing sur des binaires sans accès au code source avec AFL-QEMU. Ceci ouvre aussi la porte aux architectures supportées par QEMU d'être utilisable avec cette version d'AFL. Dans la mêlée du Cyber Grand Challenge, Shellphish a même décidé de simplifier la tâche de la compilation d'AFL pour des architectures autres que le X86. AFL-QEMU est inclus dans les releases d'AFL [26] et AFL-other-arch a son propre entrepôt sur GitHub [27].

5.2 AFL-Unicorn

Un autre projet intéressant pour le fuzzing sur des architectures moins communes est AFL-Unicorn [28]. Cet outil utilise, comme son nom l'indique, *incorpore unicorn engine*, un wrapper de QEMU permettant de l'utiliser comme un API. Cela fait en sorte qu'il est possible de définir des *harness* comme une vraie

instrumentation usuelle avec AFL. Cela dit, ceux-ci sont plus complexes que des simples `__AFL_LOOP` comme il possible de le voir dans leur exemple. On peut ainsi définir un intervalle de code qui sera émulé et ainsi éviter l'émulation de tout le code du loader, ce qui peut vous donner une bonne accélération. AFL se chargera de la mutation du corpus des cas de tests et la mutation sera également guidée par l'avancement dans la couverture de code. Il est donc possible de mapper des zones de mémoires avec votre code et ajuster le compteur d'instructions à cette zone mémoire et après que la première instruction soit exécutée, le fork server d'AFL prendra la relève et vous aurez quelque chose qui s'approchera du *in-memory fuzzing*, mais émulé dans ce cas.

6. AUTRES OUTILS

6.1 Buildroot

Buildroot [29] a comme objectif de rendre la compilation pour systèmes embarqués plus facile. Cela est donc utile afin de compiler ses propres versions de gdb, gdbserver et autres outils qui sont utiles pour le reversing, mais qui ne seront pas inclus dans les systèmes de fichiers des appareils que vous allez rencontrer. En compilant ceux-ci avec un linking statique, on peut ainsi régler le problème de bibliothèques manquantes ou non compatibles. Buildroot permet aussi de compiler des outils pour votre hôte qui sont conçus pour interagir avec votre architecture cible. Mes versions statiques pour MIPS de gdb, gdbserver, strace et ltrace sont disponibles sur GitHub si vous ne souhaitez pas les compiler vous-même [30].

6.2 GEF

GEF [31] est un autre wrapper pour gdb comme un .gdbinit, pwndbg ou peda, mais en ayant comme but d'être utilisable sur un ensemble plus grand d'architectures que seulement la famille x86. Cela est donc attirant lorsque vous développez un exploit et que vous voulez être plus productif en voyant la stack et les registres à chaque arrêt de façon automatique ainsi qu'une pléthore d'autres fonctions comme des intégrations avec retdec pour décompiler le code à la volée, Ropper pour la recherche de gadget pour le ROP et Keystone pour assembler des instructions avec l'architecture à laquelle vous faites face.

6.3 PleaseROP

Cet outil est un plugin de JEB que j'ai écrit et qui a comme objectif de trouver des ROP gadgets en utilisant la représentation intermédiaire de JEB ce qui rend l'outil découplé d'une architecture quelconque. Avec la même implémentation, il est donc possible de trouver des gadgets sur MIPS et sur ARM par exemple. Toute architecture qui sera traduite par JEB sera ainsi compatible, ce qui augmente la maintenabilité de l'outil. Voici donc un exemple sur MIPS :

```
0x4195828: ADDIU $gp, $gp, 86E0h; ADDU $gp, $gp, $ra; LW $t9, 801Ch($gp); NOP ; ADDIU $t9, $t9, 9D0h; JALR $t9;
0x4195780: ADDIU $gp, $gp, 8710h; ADDU $gp, $gp, $ra; LW $t9, 801Ch($gp); NOP ; ADDIU $t9, $t9, 748h; JALR $t9;
0x4196044: JALR $t9;
0x4196284: JR $t9;
0x4196472: LI $a0, 1; LW $t9, 8034h($gp); NOP ; JALR $t9;
0x4196660: LI $v0, 41h; MOVE $sp, $fp; LW $ra, E4h($sp); LW $fp, E0h($sp); ADDIU $sp, $sp, E8h; JR $ra;
```

Figure 13

Et sur ARM :

```
0x61232: MOV R2, #2; STR R2, [R3, #A84h]; POP {R3, R4, R5, R6, R7, PC};
0x73880: MOV R3, #20h; STRB R3, [R1, #0Ah]; MOV R3, #0; STRB R3, [R1, #0Bh]; BX LR;
0x83176: MOV R3, #2Dh; STRB R3, [R8, #-1]; MOV R0, R7; POP {R4, R5, R6, R7, R8, PC};
0x104596: ORR R1, R1, #F00000h; MOV R0, #0; POP {R4, R5, PC};
0x98036: POP {LR}; ADD SP, SP, #4; BX LR;
0x90884: POP {PC};
0x105048: POP {R1, PC};
0x103940: POP {R1, R2, LR}; MUL R3, R2, R0; SUB R1, R1, R3; BX LR;
0x39236: POP {R3, PC};
0x51940: POP {R3, R4, R5, PC};
```

Figure 14

On peut voir ici l'utilité d'avoir accès à une représentation intermédiaire uniforme pour toutes les architectures lorsqu'on souhaite bâtir des outils pour les plateformes embarquées. L'idée derrière ce plugin est très simple : en gardant un concept de *program counter* pour toutes les architectures, on n'a qu'à regarder les assignations à ce registre et faire une fouille en marche arrière jusqu'à une autre instruction qui pourrait changer le flot d'exécution. On enlève aussi les sauts conditionnels pour faciliter la construction de notre rop chain. PleaseROP est également disponible sur GitHub sur le compte de PNF Software [32].

6.4 Bowcaster

Si vous développez des exploits pour l'architecture MIPS, vous verrez que les ressources sont plus rares que pour le x86 alors il est intéressant de réutiliser le travail de nos prédécesseurs. Bowcaster est un bon exemple pour illustrer cela. C'est un framework d'exploitation spécifiquement pour le MIPS qui contient du shellcode et des encodeurs pour cette architecture, qui est hébergé aussi sur GitHub [33]. Puisque le NOP est composé de quatre null bytes en MIPS, vous comprendrez l'utilité d'encodeurs pour faciliter le processus et pour vous faire gagner du temps.

6.5 DVRF

Si vous voulez vous pratiquer sur des défis plus accessibles avant de vous lancer sur un vrai produit, b1ack0wl a conçu de petits défis incorporés dans un firmware de Linksys E1550 et ce dernier est disponible sur GitHub [34]. Ils sont séparés en 2 catégories : introduction et ceux nécessitant du shellcoding et des vulnérabilités légèrement plus dures à exploiter. C'est aussi un bon moyen d'essayer les outils présentés dans cet article pour établir votre méthodologie pour la recherche de vulnérabilités et le développement d'exploits sur MIPS. Le *write-up* d'une partie de ces challenges se trouve sur le blog de PNF Software [35].

6.6 DVAR

Si vous n'êtes pas un fan de l'architecture MIPS, Saumil Shah a lancé sa propre version [36] d'un firmware de routeur ARM également pour pratiquer l'exploitation de cette architecture et sur ce genre de produit.

CONCLUSION

Pour finir, le reversing et la recherche de vulnérabilités sur IoT peuvent être un mélange de découvertes abondantes et aussi de frustration par rapport aux paradigmes différents et parfois absence d'outils disponibles pour votre tâche. Restez patients et vous serez récompensés !

REMERCIEMENTS

Un énorme merci à Joan Calvet pour l'idée de soumettre cet article, sa relecture et ses commentaires, tous ceux ayant répondu à mes questions sur leur projet et Axelle Apvrille pour ses commentaires. ■

RÉFÉRENCES

- [1] Site officiel du CES : <https://www.ces.tech/About-CES/CES-by-the-Numbers.aspx>
- [2] Couche connectée : <https://twitter.com/edzitrone/status/951607987264614400>
- [3] Reverse-engineering serial ports : <http://www.devtty0.com/2012/11/reverse-engineering-serial-ports/>
- [4] FirmwareDB : <https://github.com/Svieg/FirmwareDB/>
- [5] FCCID.io : <https://fccid.io>
- [6] Busybox : <https://busybox.net/>
- [7] Binwalk : <https://github.com/ReFirmLabs/binwalk>
- [8] Sasquatch : <https://github.com/devttys0/sasquatch>
- [9] Jefferson : <https://github.com/sviehb/jefferson>
- [10] Démo JEB : <https://www.pnfsoftware.com/jeb/demo>
- [11] Pomme : <https://github.com/pnfsoftware/jeb2-samplecode/tree/master/src/com/pnf/pommePlugin>
- [12] Linksys security vulnerabilities : https://www.cvedetails.com/vulnerability-list/vendor_id-833/Linksys.html
- [13] Tutoriels de Oxabad1dea : <https://www.youtube.com/watch?v=q7NTV7x4Q6o>
- [14] Démo Hopper : <https://www.hopperapp.com/download.html>
- [15] Liste d'architectures de Binary Ninja : <https://binary.ninja/faq/>
- [16] Démo binary ninja : <https://binary.ninja/demo/>
- [17] Liste des architectures d'IDA Pro : <https://www.hex-rays.com/products/ida/processors.shtml>
- [18] IDA Freeware : https://www.hex-rays.com/products/ida/support/download_freeware.shtml
- [19] uEmu : <https://github.com/alexhude/uEmu>
- [20] Radare2 : <https://github.com/radare/radare2>
- [21] Cutter : <https://github.com/radareorg/cutter>
- [22] Snowman : <https://github.com/yegord/snowman>
- [23] Images QEMU-System : <http://blahcat.github.io/2017/06/25/qemu-images-to-play-with/>
- [24] Firmadyne : <https://github.com/firmadyne/firmadyne>
- [25] Avatar2 : <https://github.com/avatartwo/avatar2>
- [26] AFL : <http://lcamtuf.coredump.cx/afl/>
- [27] AFL-other-arch : <https://github.com/shellphish/afl-other-arch>
- [28] AFL-Uncorn : <https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf>
- [29] Buildroot : <https://buildroot.org/>
- [30] StaticTools : <https://github.com/svieg/StaticTools>
- [31] GEF : <https://github.com/hugsy/gef>
- [32] PleaseROP : <https://github.com/pnfsoftware/PleaseROP>
- [33] Bowcaster : <https://github.com/zcutlip/bowcaster>
- [34] DVRF : <https://github.com/b1ack0wl/DVRF>
- [35] Write-ups DVRF : <https://www.pnfsoftware.com/blog/firmware-exploitation-with-jeb-part-1/>
- [36] DVAR : <http://blog.exploitlab.net/2018/01/dvar-damn-vulnerable-arm-router.html>

Quarkslab

SECURING EVERY BIT OF YOUR DATA

Les attaquants ciblent les données, et non les infrastructures qui sont régulièrement surveillées, testées et mises à jour. Quarkslab se concentre sur la sécurisation des données, au travers de 3 outils issus de notre R&D : IRMA (orchestrateur de threat intelligence), Epona (obfuscateur) et Ivy (reconnaissance réseau). Ces produits, qui complètent nos services et formations, visent à aider les organisations à prendre leurs décisions au bon moment grâce à des informations pertinentes.



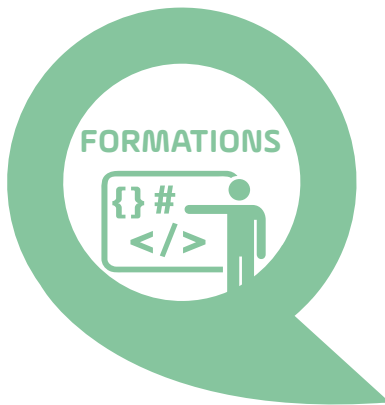
IRMA^{qb} orchestre votre threat intelligence pour déterminer la dangerosité des fichiers et fournir une vue détaillée des risques.

Epona^{qb} obfusque du code pour contrarier le reverse engineering et l'accès aux données des applications.

ivy^{qb} cartographie rapidement l'ensemble des services et informations exposés sur Internet pour des millions d'adresses.



- **Tests de sécurité** : analyse d'applications, de DRM, de vulnérabilités, de patch, fuzzing
- **Développement & analyse** : R&D à la demande, reverse engineering, design et implémentation
- **Cryptographie** : conception de protocoles, optimisation, évaluation



- Reverse engineering
- Recherche de vulnérabilités
- Développement d'exploits
- Test de pénétration d'applications Android / iOS
- Windows internals

Quarkslab
SECURING EVERY BIT OF YOUR DATA

13 rue St.-Ambroise - 75011 Paris - FRANCE
Phone: +33 (0)1 58 30 81 51 - Email: contact@quarkslab.com
@quarkslab - www.quarkslab.com



4

WEB

À découvrir dans cette partie...

PHP7 / VULNÉRABILITÉS / INJECTIONS / RECHERCHE PASSIVE

Recherche passive de vulnérabilités web avec Snuffleupagus

Pensé comme un module de sécurité pour PHP7, Snuffleupagus vous permet de renforcer la sécurité d'un site web en appliquant des règles (interdire l'usage de certaines fonctions, filtrer les tentatives d'injection SQL, etc.). Mais il va vous permettre d'appliquer ces jeux de règles en mode simulation, de manière statique donc, et d'ainsi découvrir de potentielles vulnérabilités. p.110

VULNÉRABILITÉS WEB / SCANNER / FUZZING / CRAWLER

Wapiti : chasser les vulnérabilités web

Découvrez Wapiti, un scanner de vulnérabilités web peu connu, et aussi la manière de développer des plugins pour celui-ci afin de cibler spécifiquement des types de vulnérabilités. p.116



PHP7 / VULNÉRABILITÉS / INJECTIONS / WEB / RECHERCHE PASSIVE

RECHERCHE PASSIVE DE VULNÉRABILITÉS WEB AVEC SNUFFLEUPAGUS

Julien VOISIN

Il existe un grand nombre de scanners de vulnérabilités web, mais très peu de moyens de trouver des failles de manière passive, sur un site en production. Snuffleupagus, module de sécurité pour PHP7 visant à augmenter le niveau de sécurité de sites web peut précisément être utilisé à cet usage.

1. PHP7, SUHOSIN, ET SNUFFLEUPAGUS

Nous sommes début 2018, et la fin du support de PHP5 arrive dans moins d'un an [1]. Il est donc grand temps de passer à son successeur, PHP7.

Sous PHP5, les administrateurs système soucieux de la sécurité utilisaient le module Suhosin [2], mais il n'est pas encore disponible sous PHP7, et son développement semble être à l'arrêt. C'est là que Snuffleupagus entre en scène ! Travaillant dans l'équipe sécurité d'un petit hébergeur web français (NBS System [3]), nous nous sommes donc mis en tête d'écrire un module de sécurité moderne, open source, répondant à nos besoins, et surtout avec un nom idiot.

Snuffleupagus propose tout un tas de contre-mesures pour les vulnérabilités web classiques, comme les exploitations d'**unserialize**, les injections de commandes de toutes sortes, les envois de fichiers malveillants...



Fig. 1 : Logo du projet.

2. USAGE ET RÈGLES

Dans son usage *classique*, en cas de détection d'attaque, le traitement de la requête web est interrompu, et l'évènement est inscrit dans le journal local. Il est possible, pour éviter que d'éventuels faux-positifs n'aient d'impact sur la production, d'utiliser les contre-mesures en mode *simulation* : dans ce cas, la requête n'est pas interrompue, mais l'évènement est tout de même inscrit dans le journal.

En effet, étant un module PHP, Snuffleupagus peut intervenir n'importe où : quand une requête arrive, quand une fonction particulière est appelée, ou juste après, quand un fichier est uploadé...

Au sujet des envois de fichiers, la différenciation des fichiers malveillants des fichiers légitimes s'effectue à l'aide d'un script fourni par l'utilisateur. Si le script retourne une valeur autre que 0, l'envoi du fichier ainsi que le traitement de la requête associée sont interrompus, et le fichier peut bien évidemment être sauvegardé. Le script fourni de base [4] utilise vld [5], un outil permettant de transformer du code php en bytecode php, pour détecter la présence de code php valide.

Il est donc possible de simplement déployer Snuffleupagus sur un site en PHP7, de tout passer en *simulation*, et de surveiller les journaux pour détecter de décevants faux-positifs, ou d'exploitations de juteuses nouvelles vulnérabilités.

Mais la magie de Snuffleupagus ne s'arrête pas là, puisqu'il fournit, en plus de contre-mesures génériques, un système de *virtual patching* directement branché dans le cœur de PHP, permettant d'écrire des règles spécifiques à des applications.

Par exemple, pour bloquer la présence de guillemets simples dans le paramètre **name** de la fonction **add_user** dans le fichier **/var/www/monsite/user_management.php** et de journaliser les requêtes enfreignant cette règle, il suffit d'écrire la règle suivante :

Fichier

```
sp.disable_function.function("add_user").filename("/var/www/monsite/user_
management.php").param("name").param_r("'").dump().drop();
```

Là aussi, il est possible d'utiliser le mode *simulation*, en ajoutant le filtre `.simulation()`, afin de ne pas bloquer les requêtes déclenchant des faux-positifs ; ce qui permet d'activer ce genre de filtres en production, et d'attendre qu'un mécréant vienne nous attaquer, pour récupérer sa requête.

Plutôt que d'exposer une application sur Internet en attendant qu'elle se fasse compromettre, il est possible d'utiliser un scanner de vulnérabilités, comme Burp Pro [6] (et son Backslash-powered scanner [7]), Arachni [8], Acunetix [9], netsparker [10]... dessus, et d'éplucher les logs émis par Snuffleupagus. L'avantage d'une telle approche, par rapport au scanner de vulnérabilité seul, c'est la possibilité de trivialement détecter des vulnérabilités n'ayant pas d'effets immédiatement détectables, comme une injection en aveugle, une exception rattrapée par un handler, ou encore une injection de second degré.

Par exemple, *imaginons* un site permettant de créer des comptes utilisateurs, avec un couple nom/mot de passe, puis utilisant une requête vers une base de données, dans un wrapper gérant les erreurs, pour les afficher. Il est relativement compliqué de détecter une possible injection dans ladite requête avec un scanner classique : l'injection SQL se fait en deux temps (injection de second degré), et les erreurs sont silencieusement ignorées. Avec Snuffleupagus, il suffit de filtrer sur la présence d'un nombre impair de quotes à l'aide d'une expression rationnelle, pour obtenir des résultats intéressants.

Pour aller encore plus vite, il suffit de parcourir l'application, énumérer ses entrées, et d'y injecter des marqueurs qui devraient être filtrés (simple quotes, double quotes, commentaires...) qui seront repérés par des règles spécifiques.

3. EXEMPLES

Il est possible d'écrire tout un tas de règles pour attraper un grand nombres de vulnérabilités :

Des appels à `chmod` trop permissifs :

Fichier

```
sp.disable_function.function("chmod").param("mode").value_r("^([0-9]){2}\[67\]$").drop();
sp.disable_function.function("chmod").param("mode").value_r("o\\.+w$").drop();
```

Des comportements typiques de phase de reconnaissance post-compromission :

Fichier

```
sp.disable_function.function("ini_get").param("var_name").value_r("(?:allow_url_fopen|open_basedir|suhosin)").drop();
sp.disable_function.function("function_exists").param("function_name").value_r("(?:eval|exec|system)").drop();
sp.disable_function.function("is_callable").param("var").value_r("(?:eval|exec|system)").drop();
sp.disable_function.function("ini_set").param("var_name").value("assert.active").drop();
sp.disable_function.function("ini_set").param("var_name").value("zend.assertions").drop();
sp.disable_function.function("ini_set").param("var_name").value("include_path").drop();
sp.disable_function.function("ini_set").param("var_name").value("open_basedir").drop();
```

Des inclusions de fichiers ne se terminant pas par une extension légitime :

Fichier

```
sp.disable_function.function_r("^(?:require|include)_once$").value_r("\\.(?:php|php7|inc|tpl)$").allow();
```



```
sp.disable_function.function_r("^require|include$").value_r("\\.(?:php|php7|
inc|tpl)$").allow();
sp.disable_function.function_r("^(?:require|include)_once$").drop();
sp.disable_function.function_r("^require|include$").drop();
```

Des injections de commandes :

Fichier

```
sp.disable_function.function("system").param("command").value_r("$|;&'\\n").
drop();
sp.disable_function.function("shell_exec").param("command").
value_r("$|;&'\\n").drop();
sp.disable_function.function("exec").param("command").value_r("$|;&'\\n").
drop();
sp.disable_function.function("proc_open").param("command").
value_r("$|;&'\\n").drop();
```

Et évidemment, des tentatives d'injections SQL :

Fichier

```
sp.disable_function.function_r("mysqli?_query").param("query").value_r("/\\*").drop();
sp.disable_function.function_r("mysqli?_query").param("query").value_r("--").drop();
sp.disable_function.function_r("mysqli?_query").param("query").value_r("#").drop();
sp.disable_function.function_r("mysqli?_query").param("query").value_r(";.*").drop();
sp.disable_function.function_r("mysqli?_query").param("query").value_r("benchmark").
drop();
sp.disable_function.function_r("mysqli?_query").param("query").value_r("sleep").drop();
sp.disable_function.function_r("mysqli?_query").param("query").value_r("information_
schema").drop();
sp.disable_function.function("PDO::query").param("query").value_r("/\\*").drop();
sp.disable_function.function("PDO::query").param("query").value_r("--").drop();
sp.disable_function.function("PDO::query").param("query").value_r("#").drop();
sp.disable_function.function("PDO::query").param("query").value_r(";.*").drop();
sp.disable_function.function("PDO::query").param("query").value_r("benchmark\\s*\\(").
drop();
sp.disable_function.function("PDO::query").param("query").value_r("sleep\\s*\\(").
drop();
sp.disable_function.function("PDO::query").param("query").value_r("information_
schema").drop();

sp.disable_function.function_r("mysqli?_query").ret("FALSE").drop();
sp.disable_function.function_r("PDO::query").ret("FALSE").drop();
```

4. CAS PRATIQUE

Prenons un fichier nommé **utils/users.php**, comportant du code vulnérable, dans le fichier :

Fichier

```
<?php
function get_current_id($a = "id"){
    return system($a);
}
?>
```

Si un attaquant arrivait à assigner autre chose qu'**id** à la variable **\$a**, il s'agit vraisemblablement d'une vulnérabilité. Nous allons donc écrire une règle pour autoriser uniquement la valeur **id** dans le paramètre **\$a** de la fonction **get_current_id**, et interdire son utilisation dans tout autre cas. Comme les règles de Snuffleupagus sont interprétées séquentiellement, rien de plus simple :

Fichier

```
sp.disable_function.function("get_current_id").param("$a").value("id").  
allow();  
sp.disable_function.function("get_current_id").drop();
```

En cas de violation, la ligne suivante apparaît dans les logs :

Fichier

```
[snuffleupagus][0.0.0.0][disabled_function][drop] The call to the function  
'get_current_id' in /var/www/mysite/utils/users.php:2 has been disabled,  
because its argument 'a' content (ls) matched a rule.
```

CONCLUSION

Snuffleupagus est le descendant spirituel de Suhosin, à qui il doit beaucoup. Il permet non seulement d'élever le niveau de sécurité de sites web en PHP7, mais également de forcer l'attaquant à réfléchir à deux fois avant de tenter de pirater quelqu'un. Lancer une attaque contre un site utilisant Snuffleupagus avec des règles un peu larges, c'est courir le risque de révéler un vecteur d'attaque, et donc de voir la faille corrigée.

REMERCIEMENTS

Merci à NBS System de nous avoir permis et encouragé, à mon (ex) collègue Sébastien Blot et moi-même, d'avoir pu écrire et publier ce module PHP. ■

RÉFÉRENCES

- [1] <https://secure.php.net/supported-versions.php>
- [2] <https://suhosin.org>
- [3] <https://nbs-system.com>
- [4] https://github.com/nbs-system/snuffleupagus/blob/master/scripts/upload_validation.py
- [5] <https://derickrethans.nl/projects.html#vld>
- [6] <https://portswigger.net/burp/>
- [7] <http://blog.portswigger.net/2016/11/backslash-powered-scanning-hunting.html>
- [8] <http://www.arachni-scanner.com/>
- [9] <https://www.acunetix.com/>
- [10] <https://www.netsparker.com/>

ACTUELLEMENT DISPONIBLE

GNU/LINUX MAGAZINE HORS-SÉRIE N°95 !



NE LE MANQUEZ PAS
CHEZ VOTRE MARCHAND DE JOURNAUX ET SUR :
<https://www.ed-diamond.com>



VULNÉRABILITÉS WEB / SCANNER / FUZZING / CRAWLER

WAPITI : CHASSER LES VULNÉRABILITÉS WEB

Nicolas SURRIBAS

Initiallement créé en 2006, Wapiti était un outil à l'origine dédié à la recherche de failles XSS. Au fil du temps, ce logiciel libre s'est étoffé de nouveaux modules lui permettant de rechercher les vulnérabilités web les plus connues. La version 3.0.0 sortie en janvier 2018 apporte de nombreuses nouvelles options et marque le passage à Python 3.

1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161, 2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172, 2173, 2174, 2175, 2176, 2177, 2178, 2179, 2180, 2181, 2182, 2183, 2184, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2192, 2193, 2194, 2195, 2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229, 2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261, 2262, 2263, 2264, 2265, 2266, 2267, 2268, 2269, 2270, 2271, 2272, 2273, 2274, 2275, 2276, 2277, 2278, 2279, 2280, 2281, 2282, 2283, 2284, 2285, 2286, 2287, 2288, 2289, 2290, 2291, 2292, 2293, 2294, 2295, 2296, 2297, 2298, 2299, 2300, 2301, 2302, 2303, 2304, 2305, 2306, 2307, 2308, 2309, 2310, 2311, 2312, 2313, 2314, 2315, 2316, 2317, 2318, 2319, 2320, 2321, 2322, 2323, 2324, 2325, 2326, 2327, 2328, 2329, 2330, 2331, 2332, 2333, 2334, 2335, 2336, 2337, 2338, 2339, 2340, 2341, 2342, 2343, 2344, 2345, 2346, 2347, 2348, 2349, 2350, 2351, 2352, 2353, 2354, 2355, 2356, 2357, 2358, 2359, 2360, 2361, 2362, 2363, 2364, 2365, 2366, 2367, 2368, 2369, 2370, 2371, 2372, 2373, 2374, 2375, 2376, 2377, 2378, 2379, 2380, 2381, 2382, 2383, 2384, 2385, 2386, 2387, 2388, 2389, 2390, 2391, 2392, 2393, 2394, 2395, 2396, 2397, 2398, 2399, 2400, 2401, 2402, 2403, 2404, 2405, 2406, 2407, 2408, 2409, 2410, 2411, 2412, 2413, 2414, 2415, 2416, 2417, 2418, 2419, 2420, 2421, 2422, 2423, 2424, 2425, 2426, 2427, 2428, 2429, 2430, 2431, 2432, 2433, 2434, 2435, 2436, 2437, 2438, 2439, 2440, 2441, 2442, 2443, 2444, 2445, 2446, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2454, 2455, 2456, 2457, 2458, 2459, 2460, 2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2505, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2528, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2549, 2550, 2551, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2587, 2588, 2589, 2590, 2591, 2592, 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601, 2602, 2603, 2604, 2605, 2606, 2607, 2608, 2609, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2646, 2647, 2648, 2649, 2650, 2651, 2652, 2653, 2654, 2655, 2656, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664, 2665, 2666, 2667, 2668, 2669, 2670, 2671, 2672, 2673, 2674, 2675, 2676, 2677, 2678, 2679, 26

Pour cet article, les scans seront lancés sur un [WACKOPICKO], une application web intentionnellement vulnérable.

Le périmètre par défaut appliqué à cette URL est de type « folder », cela signifie que seules les URLs se trouvant sous le même dossier web seront prises en compte par le scanner.

```
$ wapiti -u http://127.0.0.1/WackoPicko/
```

[illegible]

Un mécanisme de session conserve les données du scan dans une base de données au format sqlite3.

Le nombre d'URLs et formulaires pris en considération pour l'attaque (39 ici) et la liste des modules existants sont affichés.

Wapiti se base sur des règles très simples (présence de paramètres passés, type MIME associé à la ressource, extension du fichier) pour faire le tri parmi toutes les pages scannées afin d'exclure les ressources statiques (JavaScript, images, etc.).

1.2 Seconde phase

Le logiciel lance ensuite les différents modules d'attaque. Ces derniers injectent des payloads et observent le comportement de l'application web pour déterminer la présence ou l'absence d'une vulnérabilité.

On peut séparer l'observation de ces comportements en trois catégories :

- ⇒ La recherche d'un résultat spécifique dans la réponse HTTP provoquée par la bonne exécution du payload. Par exemple, le module **exec** (voir plus bas) va chercher la présence de la chaîne « PATH= » lorsqu'il tente de faire exécuter le programme Unix **env**. Cette technique a l'avantage d'avoir peu de faux positifs.
- ⇒ La recherche de messages d'erreurs suite à l'injection de caractères destinés à provoquer une erreur de syntaxe (*You have an error in your SQL syntax, Failure evaluating code, Failed opening required*, etc.). Cette technique peut malheureusement provoquer des faux positifs sur une application boguée.
- ⇒ L'observation d'un temps de traitement anormal de la requête par l'application web suite à l'injection d'un payload destiné à provoquer une temporisation (emploi de la commande Unix **sleep** ou utilisation de **waitfor delay** sur SQL Server). Depuis la version 3.0.0, les requêtes générant un timeout sont testées à trois reprises pour s'assurer qu'il ne s'agit pas de faux positifs.

Si l'option **-m** (ou **--module**) n'est pas passée à Wapiti avec une liste spécifique de modules d'attaque, seuls les modules les plus courants sont exécutés. Il s'agit de :

- ⇒ **exec** : un module pour chercher les failles d'exécution de commandes ou de code ;
- ⇒ **file** : un module pour détecter les failles d'inclusion PHP et autres « directory traversal » ;
- ⇒ **sql** : un module de détection de failles d'injections SQL. Il se base sur la présence de messages d'erreurs présents dans les pages retournées ;
- ⇒ **xss** : un module de détection d'injections HTML/JS. Il envoie d'abord un identifiant unique pour déterminer s'il est reflété dans la page et le cas échéant tente d'injecter un payload XSS prêt à l'emploi (PoC qui prend soin de refermer des tags HTML si nécessaire) ;
- ⇒ **blindsqli** : comme son nom l'indique, ce module n'est pas basé sur la présence d'erreur. La technique employée est ici time-based (utilisation de payloads provoquant une temporisation) ;
- ⇒ **permanentxss** : ce module prend la suite du module XSS en procédant à un nouveau scan de la cible. S'il retrouve les identifiants uniques précédemment injectés, il tente de nouvelles attaques XSS pour déterminer s'il s'agit de « stored XSS » (XSS non-réfléchi). Ce fonctionnement permet de trouver les failles XSS où le point d'injection n'est pas le même que le point d'exécution du XSS.

La suite de l'affichage (réduit pour des raisons de lisibilité) concerne l'exécution des modules d'attaque :

Fichier

```
[*] Lancement du module exec
---
Interprétation de code PHP dans http://127.0.0.1/WackoPicko/admin/index.php via
une injection dans le paramètre page
Evil request:
  GET /WackoPicko/admin/index.php?page=data%3A%3Bbase64%2CPD9waHAgZWNoYAndzRwMX
  QxJywnX2V2YWwnOyA%2FPg%3D%3D HTTP/1.1
  Host: 127.0.0.1
---

[*] Lancement du module file
---
Faille d'inclusion distante dans http://127.0.0.1/WackoPicko/admin/index.php via
une injection dans le paramètre page
Evil request:
  GET /WackoPicko/admin/index.php?page=http%3A%2F%2Fwww.google.fr%2F%3F HTTP/1.1
  Host: 127.0.0.1
-

[*] Lancement du module sql
---
Injection MySQL dans http://127.0.0.1/WackoPicko/users/login.php via une
injection dans le paramètre username
Evil request:
  POST /WackoPicko/users/login.php HTTP/1.1
  Host: 127.0.0.1
  Referer: http://127.0.0.1/WackoPicko/users/login.php
  Content-Type: application/x-www-form-urlencoded

  username=%C2%BF%27%22%28&password=letmein
---

[*] Lancement du module xss
---
Faille XSS dans http://127.0.0.1/WackoPicko/pictures/search.php via une injection
dans le paramètre query
Evil request:
  GET /WackoPicko/pictures/search.php?query=%22%2F%3E%3Cscript%3Ealert%28%27wcjz
  vjeod5%27%29%3C%2Fscript%3E&x=1&y=1 HTTP/1.1
  Host: 127.0.0.1
  Referer: http://127.0.0.1/WackoPicko/
---

[*] Lancement du module blindsql
---
Faille d'injection SQL en aveugle dans http://127.0.0.1/WackoPicko/users/login.
php via une injection dans le paramètre username
Evil request:
  POST /WackoPicko/users/login.php HTTP/1.1
  Host: 127.0.0.1
  Referer: http://127.0.0.1/WackoPicko/users/login.php
  Content-Type: application/x-www-form-urlencoded

  username=%27%20or%20sleep%28%29%231&password=letmein
-

```

1.3 Troisième phase

À la suite de la phase d'attaque Wapiti génère un rapport (par défaut en HTML, les formats texte, XML et JSON sont supportés).

Certains formats offrent une ligne de commandes Curl permettant de reproduire facilement la requête d'attaque.

Wapiti vulnerability report

Target: http://127.0.0.1/WackoPicko/

Date of the scan: Wed, 17 Jan 2018 19:44:54 +0000. Scope of the scan: folder

Category	Number of vulnerabilities found
Contournement de htaccess	0
Injection CRLF	0
Divulcation de fichier	2
Injection SQL	1
Exécution de commandes	2
Injection SQL en aveugle	2
Fichier potentiellement dangereux	0
Cross Site Scripting	1
Copie de sauvegarde	0
Consommation anormale de ressource	0
Erreur interne au serveur	6

Divulcation de fichier

Description

Ces techniques permet à l'attaquant d'accéder à des fichiers auxquels il n'est pas sensé accéder car en dehors de la racine du serveur web. En utilisant certaines séquences comme '../', il peut remonter dans l'arborescence pour ainsi lister des répertoires ou obtenir le contenu de fichiers.

Vulnerability found in /WackoPicko/admin/index.php

Description	HTTP Request	cURL command line
Faille d'inclusion distante via une injection dans le paramètre page		

Bravo ! Vous avez effectué votre premier scan avec Wapiti !

2. SOUS LES CORNES LE CERVEAU

Le crawler est basé sur un parseur HTML (utilisant par défaut **libxml2**) qui connaît HTML5 dans ses moindres recoins. Wapiti est aussi capable de fouiller dans les binaires Flash (.swf).

Pour ce qui est de JavaScript, les paramètres de fonctions ainsi que les valeurs des variables sont surveillés via l'utilisation d'un parseur et d'un interpréteur plus que sommaire. Des expressions régulières sont aussi appliquées pour déterminer s'il s'agit ou non d'URL.

Depuis la version 2.3.0, Wapiti atteint un score de 48 % sur [WIVET] (un logiciel permet de déterminer les capacités d'exploration des scanners de vulnérabilité). Ce score était bon pour un logiciel libre au moment de cette version, mais a depuis été dépassé par des solutions concurrentes.

En interne, Wapiti gère les URLs et formulaires trouvés sous la forme d'objets Request qui conservent les paramètres et leur valeur respective (la répétition d'un même paramètre est ici autorisée).

Les modules d'attaque effectuent pour la plupart du fuzzing en injectant une série de payloads dans les différentes entrées.

Ces payloads sont simplement stockés dans des fichiers textes pour certains modules rendant très facile la customisation. Il faudra tout de même s'assurer que le module correspondant puisse détecter le résultat attendu pour ce nouveau payload.

Depuis la version 3.0.0, un moteur de mutation (classe **Mutator**) permet d'injecter automatiquement une liste de payloads dans les entrées de chaque objet Request. L'utilisation du Mutator permet ainsi d'écrire des modules plus facilement avec moins de lignes de code.

3. PRENDRE LE WAPITI PAR LES CORNES

Tous les scanners web ont à faire face au même cas d'école : les calendriers web permettant de passer sur la page du mois suivant indéfiniment et qui, sans protection, ferait tourner le code du scanner éternellement.

Wapiti a bien évidemment des garde-fous pour s'assurer que le scan se finisse à un moment. Toutefois lors d'un scan lancé sur un site Internet avec beaucoup de contenu (exemple : un CMS bien rempli), la totalité du process (incluant les attaques) peut facilement atteindre plusieurs jours.

Pour restreindre la quantité d'URLs et formulaires remontés lors du scan (et donc réduire le temps d'attaque en conséquence), la nouvelle version apporte 7 nouvelles options qui s'ajoutent aux 2 déjà existantes.

On peut citer :

- ⇒ la possibilité de retirer certains paramètres des URLs avec les options **-r** (ou **--remove**) ;
- ⇒ ignorer certains paramètres lors des attaques avec **--skip** ;
- ⇒ exclure des URLs avec la possibilité de passer des wildcards (*) via **--x** ou **--exclude** ;
- ⇒ limiter la profondeur de scan via **-d** ou **--depth** (l'URL de base a une profondeur de 0, les liens extraits de la première page une profondeur de 1, etc.) ;
- ⇒ limiter le nombre de variantes d'une même URL avec **--qs-limit** (pour un script donné, compter le nombre de query string différentes) ;
- ⇒ limiter le nombre de liens sous une même arborescence (scripts directement sous cette arborescence) avec **--max-files-per-dir** ;
- ⇒ définir le temps (en minutes) à partir duquel stopper le scan avec **--max-scan-time** ;
- ⇒ ne pas attaquer les URLs et formulaires ayant plus d'un certain nombre d'entrées avec **--max-parameters** ;
- ⇒ définir une force de scan via **-S** ou **--scan-force** reprenant les valeurs de l'option **-T** de Nmap.

4. PÉRIMÈTRES DE SCAN

Les sessions générées par Wapiti sont définies par l'URL de base associée au périmètre utilisé.

Par conséquent même si vous changez certains paramètres (verbosité, format du rapport, limites de scan, etc.), la précédente session ayant les mêmes valeurs de **--url** et **--scope** sera chargée si elle existe.

En dehors du périmètre « folder » mentionné plus tôt, les autres valeurs possibles sont « url » (le périmètre se restreint à l'URL exacte), « page » (on s'attaque au script, avec ou sans query string) et « domain » (on peut sauter d'un site à un autre s'ils sont sous le même domaine).

5. AUTHENTIFICATION

Pour accéder à une partie d'un site nécessitant une authentification, Wapiti dispose d'une option **-c** (ou **--cookie**) qui permet de charger les cookies nécessaires depuis un fichier JSON.

La récupération des cookies est à effectuer via l'utilitaire **wapiti-getcookie** qui vient avec le logiciel.

Ce dernier requiert l'URL où trouver le formulaire de connexion ainsi que le chemin vers le fichier JSON où seront stockées les données (il est possible d'utiliser le même fichier JSON pour tous les sites scannés).

Terminal

```
$ wapiti-getcookie -u http://127.0.0.1/WackoPicko/users/login.php -c
cookies.json
<Cookie PHPSESSID=egmco2amj8on3887dh4oe0pr5fhom31q9u75m4b3ujf426t857p1 for
127.0.0.1/>
```

```
Sélectionnez le formulaire de votre choix ou entrez 'q' pour quitter :
0) GET http://127.0.0.1/WackoPicko/pictures/search.php?query=&x=1&y=1 (0)
1) POST http://127.0.0.1/WackoPicko/users/login.php (0)
   data: username=&password=
```

```
Entrez un numéro : 1
```

```
Veuillez renseigner les champs pour le formulaire suivant :
url = http://127.0.0.1/WackoPicko/users/login.php
username: bryce
password: bryce
<Cookie PHPSESSID=egmco2amj8on3887dh4oe0pr5fhom31q9u75m4b3ujf426t857p1 for
127.0.0.1/>
```

Pour relancer le scan en étant authentifié, on spécifiera le chemin du fichier JSON. L'option **-x** permet quant à elle d'exclure une URL du scan (évitant ici de fermer notre session fraîchement créée sur le site) :

Terminal

```
$ wapiti -u http://127.0.0.1/WackoPicko/ -c cookies.json -x \
http://127.0.0.1/WackoPicko/users/logout.php --flush-session -v 2
```

```
WAPITI
```

```
Wapiti-3.0.0 (wapiti.sourceforge.net)
[*] Soyez prudent ! Nouvelle lune ce soir.
[+] GET http://127.0.0.1/WackoPicko/ (0)
[+] GET http://127.0.0.1/WackoPicko/pictures/search.php?query=&x=1&y=1 (1)
[+] GET http://127.0.0.1/WackoPicko/users/home.php (1)
[+] GET http://127.0.0.1/WackoPicko/pictures/search.php (1)
```



```
[+] GET http://127.0.0.1/WackoPicko/pictures/upload.php (1)
[+] GET http://127.0.0.1/WackoPicko/pictures/recent.php (1)
[+] GET http://127.0.0.1/WackoPicko/admin/index.php?page=login (1)
[+] GET http://127.0.0.1/WackoPicko/calendar.php (1)
[+] GET http://127.0.0.1/WackoPicko/admin/index.php (1)
[+] GET http://127.0.0.1/WackoPicko/cart/review.php (1)
[+] GET http://127.0.0.1/WackoPicko/tos.php (1)
[+] GET http://127.0.0.1/WackoPicko/users/sample.php?userid=1 (1)
[+] GET http://127.0.0.1/WackoPicko/users/view.php?userid=297 (2)
[+] GET http://127.0.0.1/WackoPicko/pictures/purchased.php (2)
[+] GET http://127.0.0.1/WackoPicko/users/view.php (2)
[+] GET http://127.0.0.1/WackoPicko/action.swf (2)
[+] POST http://127.0.0.1/WackoPicko/pictures/upload.php (2)
      data: MAX_FILE_SIZE=10485760&tag=default&name=default&title=default&price=
      default
      files: pic=pix.gif
[+] GET http://127.0.0.1/WackoPicko/pictures/view.php?picid=11 (2)
```

Ici le scan a été relancé en mode verbeux (**-v 2**) et l'historique de la session a été vidé (**--flush-session**). Le mode verbeux nous liste les URLs et formulaires trouvés par le crawler. En fin de ligne, un chiffre indique le niveau de profondeur de la ressource.

On peut remarquer ici les références à un panier électronique (*cart*) accessible seulement si des identifiants ont été saisis.

Dans ce même mode, on peut voir les requêtes modifiées en plus des requêtes originales :

Terminal

```
Note
=====
Ce scan a été sauvé dans le fichier /home/devloop/.wapiti/scans/127.0.0.1_
folder_30e1d821.db
Ce scan sera repris la prochaine fois à moins que vous ne passiez l'option
--skip-crawl.
[*] Wapiti a trouvé 61 URLs et formulaires lors du scan
[*] Chargement des modules :
      mod_crlf, mod_exec, mod_file, mod_sql, mod_xss, mod_backup, mod_
      htaccess, mod_blindsql, mod_permanentxss, mod_nikto, mod_delay, mod_buster,
      mod_shellshock

[*] Lancement du module exec
[+] GET http://127.0.0.1/WackoPicko/ (0)
[+] GET http://127.0.0.1/WackoPicko/action.swf (2)
[+] GET http://127.0.0.1/WackoPicko/admin/index.php?page=login (1)
[~] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%3Benv (1)
---
Une erreur HTTP 500 a été obtenue avec http://127.0.0.1/WackoPicko/admin/
index.php
Evil request:
      GET /users/WackoPicko/website/admin/index.php?page=%3Benv HTTP/1.1
      Host: 127.0.0.1
---
[~] GET http://127.0.0.1/WackoPicko/admin/index.php?page=a%3Benv (1)
[~] GET http://127.0.0.1/WackoPicko/admin/index.php?page=a%29%3Benv (1)
[~] GET http://127.0.0.1/WackoPicko/admin/index.php?page=login%3Benv (1)
[~] GET http://127.0.0.1/WackoPicko/admin/index.php?page=login%0Aenv (1)
[~] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%2Fe%00 (1)
[~] GET http://127.0.0.1/WackoPicko/admin/index.php?page=a%3Bexit%28base64_
      decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%2F%2F (1)
[~] GET http://127.0.0.1/WackoPicko/admin/index.php?page=a%3Bexit%28base64_
      decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%23 (1)
```

```
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%22%3Bexit%28base64_
decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%2F%2F (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%22%3Bexit%28base64_
decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%23 (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%27%3Bexit%28base64_
decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%2F%2F (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%27%3Bexit%28base64_
decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%23 (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%22.exit%28base64_
decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%2F%2F (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%22.exit%28base64_
decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%23 (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%27.exit%28base64_
decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%2F%2F (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=%27.exit%28base64_
decode%28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%23 (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=exit%28base64_decode%
28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%2F%2F (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=exit%28base64_decode%
28%27dzRwMXQxX2V2YWw%3D%27%29%29%3B%23 (1)
[ ] GET http://127.0.0.1/WackoPicko/admin/index.php?page=data%3A%3Bbase64%2CPD9w
9waHAgZWNobyAndzRwMXQxJywnX2V2YWwnOyA%2FPg%3D%3D (1)
---
```

Interprétation de code PHP dans http://127.0.0.1/WackoPicko/admin/index.php
via une injection dans le paramètre page

Evil request:

```
GET /users/WackoPicko/website/admin/index.php?page=data%3A%3Bbase64%2CPD9w
aHAgZWNobyAndzRwMXQxJywnX2V2YWwnOyA%2FPg%3D%3D HTTP/1.1
Host: 127.0.0.1
---
```

6. ÉCRIRE UN NOUVEAU MODULE D'ATTAQUE

Nous allons écrire ici un module permettant de trouver sur la cible les scripts autorisant des méthodes HTTP autres que les classiques **GET**, **POST**, etc. Le principe est bien connu, car il existe des modules similaires pour Nmap et Metasploit. L'intérêt sera par exemple de détecter la présence d'un WebDAV.

Un tel module prend environ 60 lignes et seulement quelques minutes si l'on connaît bien les classes de l'outil :

Fichier

```
from itertools import chain
from wapitiCore.attack.attack import Attack
from wapitiCore.net.web import Request
from requests.exceptions import RequestException

class mod_methods(Attack):
    """This class detects interesting HTTP methods"""

    name = "methods"
    PRIORITY = 6
    KNOWN_METHODS = {"GET", "POST", "OPTIONS", "HEAD", "TRACE"}

    def attack(self, http_resources, forms):
        excluded_path = set()
```



```

        for original_request in chain(http_resources, forms):
            try:
                page = original_request.path
                if page in excluded_path:
                    continue

                excluded_path.add(page)
                option_request = Request(
                    page,
                    "OPTIONS",
                    referer=original_request.referer,
                    link_depth=original_request.link_depth
                )

                if self.verbose == 2:
                    print("[+] {}".format(option_request))

                try:
                    response = self.crawler.send(option_request)
                except RequestException:
                    continue
                else:
                    if 200 <= response.status < 400:
                        methods = response.headers.get("allow", '').upper().split(',')
                        methods = {method.strip() for method in methods if method}

                        interesting_methods = sorted(methods - self.KNOWN_METHODS)

                        if interesting_methods:
                            self.log_orange("---")
                            self.log_orange(
                                "Interesting methods allowed on {}: {}".format(
                                    page,
                                    ", ".join(interesting_methods)
                                )
                            )
                            self.log_orange("---")
                        except (KeyboardInterrupt, RequestException) as exception:
                            yield exception
                        yield original_request

strip()

```

Dans un premier temps, on crée un nouveau module baptisé ici **mod_methods.py** dans le dossier **wapitiCore/attack** puis on y rajoute une classe **mod_methods** héritant de la classe de base des modules (baptisée **Attack**).

Notre nouvelle classe disposera d'un attribut **name** correspondant au nom qui servira à l'activer avec l'option **-m**.

On surdéfinira uniquement la méthode « attack » qui reçoit en argument la liste des ressources obtenues via **GET** ainsi que la liste de celles obtenues par **POST**.

Pour chaque ressource, on en extrait le path (via l'attribut du même nom sur la classe **Request**) que l'on utilise pour créer un nouvel objet **Request** en indiquant la méthode **HTTP OPTIONS** (la classe **Request** est immuable comme l'est la classe de base **str** de Python).

Il suffit alors d'envoyer notre requête via **self.crawler.send**, discriminer sur le code de statut de la réponse et extraire les méthodes intéressantes autorisées depuis l'entête **HTTP Allow**.

On affiche alors nos trouvailles via **self.log_orange** (Wapiti a un code couleur pour la sévérité où le rouge représente une vulnérabilité, l'orange un avertissement, le cyan une information).

Les modules doivent remonter les exceptions **KeyboardInterrupt** et **RequestException** qui sont gérées en amont. Les requêtes originales reçues en argument sont aussi remontées (dans l'objectif d'intégrer ultérieurement un indicateur de progression), mais c'est un détail d'implémentation susceptible de changer.

Pour terminer, il faudra référencer le module **mod_methods** dans la liste présente dans le fichier **attack.py**.

Exemple d'exécution de Wapiti avec notre nouveau module (chargé via **-m methods**) :

Terminal

```
[*] Lancement du module methods
---
Interesting methods allowed on http://127.0.0.1/users/dav/: COPY, DELETE,
LOCK, MOVE, PROPFIND, PROPPATCH, UNLOCK
---
```

Le code du module peut se trouver sur le SVN du projet [MODULE].

7. FUTURS DÉVELOPPEMENTS

Wapiti a toujours eu une todo list longue comme le bras et, bien qu'elle ait été raccourcie avec la dernière version, il reste de nombreuses améliorations à faire au projet.

La prochaine version se concentrera principalement sur l'amélioration du module XSS existant, réduire toujours plus le nombre de faux positifs sur les différents modules ainsi que rajouter de nouveaux modules d'attaque (principalement SSRF, XXE).

Des pistes sont aussi étudiées pour améliorer le support du JavaScript entre l'utilisation d'un module de proxy intercepteur ou l'utilisation d'un navigateur headless.

Enfin des outils additionnels seront rajoutés comme un outil de brute force de nom de fichiers et dossiers indépendant de l'application (très utile pour les Capture The Flag) ainsi qu'un outil de brute force de page de connexion.

Les avancées sur le développement de Wapiti pourront être suivies sur le [BLOG] de l'auteur.

REMERCIEMENTS

Je tiens à remercier Émilien Gaspar pour la proposition d'écrire un article pour mon magazine favori ainsi que son aide à la rédaction. Merci également à Laurent Butti pour la relecture de l'article. ■

RÉFÉRENCES

[INSTALLATION] <https://sourceforge.net/p/wapiti/wiki/Installation/>

[WACKOPICKO] <https://github.com/adamdoupe/WackoPicko>

[MODULE] <https://sourceforge.net/p/wapiti/code/417/>

[WIVET] <https://github.com/bedirhan/wivet>

[BLOG] <http://devloop.users.sourceforge.net/>

VISITEZ NOTRE BOUTIQUE ET DÉCOUVREZ NOS GUIDES !



RENDEZ-VOUS SUR **www.ed-diamond.com**
POUR DÉCOUVRIR TOUS LES GUIDES DE VOS MAGAZINES PRÉFÉRÉS !





exécution concolique

fuzzing / rétro-ingénierie

afl

angr

radare2

ANALYSE STATIQUE & DYNAMIQUE

Fuzzing, exécution symbolique, reverse engineering : d'afl à r2, reprenez les bases pour la recherche de vulnérabilités



fuzzing différentiel

analyse automatisée

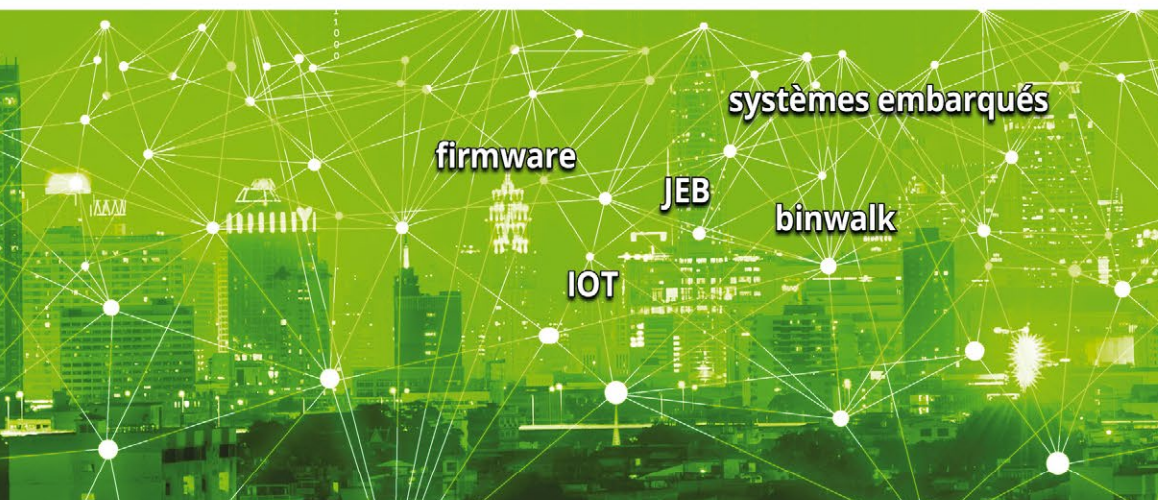
CDF

grap

Wycheproof

CRYPTOGRAPHIE

De l'identification de primitives vulnérables dans des binaires aux tests automatisés : initiez-vous à grap et à CDF - Crypto Differential Fuzzing



systèmes embarqués

firmware

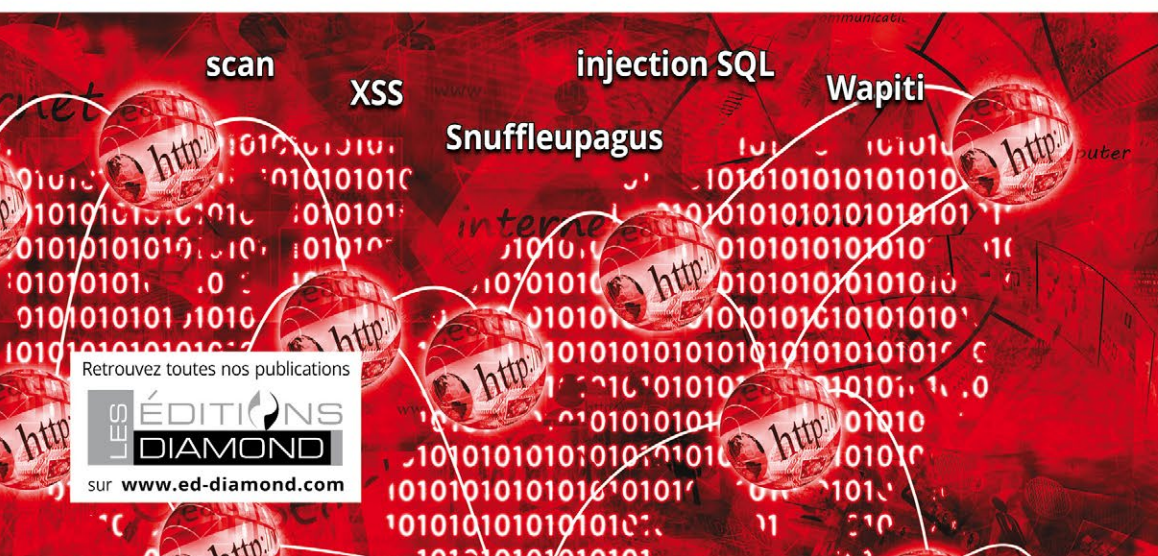
JEB

binwalk

IOT

IOT

Prenez en main l'ensemble des outils nécessaires pour tester la sécurité des objets connectés



scan

XSS

injection SQL

Wapiti

Snuffleupagus

WEB

XSS, injection SQL : mettez en œuvre les outils pour déceler les problèmes de sécurité dans vos applications web

Retrouvez toutes nos publications

ÉDITIONS
DIAMOND

sur www.ed-diamond.com

