TogDu

Rapport d’analyse forensique

DGSIB/FOFO/RP19_5

Fañch
30 a viz ebel 2019

PUBLIQUE / BZH SEUL :

Contexte :

TogDu a été mandatée en date du 29/03/2019 (29 a viz Meurzh 2019) par la Direction Générale de la Sécurité Informatique Bretonne, pour donner suite à la saisie d'un matériel informatique (ordiphone Android sécurisé ThéoBreizh), dans le cadre de l'affaire « marée brune » - voir ANNEXE/ PVRB1

Les éléments techniques mis à disposition par M. [redacted] comprennent :

- Une note technique (PVRB2) 61c101d80e2cc528224aee067b715dc3d4f9fbf2
- Une copie (REF :DGSIB/MB/00-0319-1897X1-001/2) de la saisie technique :
    - flash.bin 2b674e8645c42092c60071859518ff9537d19b7c
    - rom.bin b515a3a0c02fc82801e95200ba79aee5d0237418
- Une trace de consommation électrique effectuée au démarrage de l'appareil (power_consumption.npz /4791e9d1870b155d1bd0d8a218b0c2c6b8fd0ea9)

Le mandat concerne le déchiffrement, l'analyse et la recherche de preuves incriminantes pouvant être contenues dans la copie de saisie technique DGSIB/MB/00-0319-1897X1-001/2.

Sur demande de la DGSIB cette analyse s'accompagnera d'un rapport didactique (ce document) expliquant la démarche de notre expert (Fañch) et d'une livraison de l'ensemble des outils développés dans le cadre de cette analyse.

Note légale :

Le présent document n'engage pas la responsabilité de TogDu en l'usage qu'il pourrait être fait des informations, scripts, outils ou méthodes décrites, notamment dans l'usage qu'il pourrait en être fait pour affaiblir, déchiffrer ou contourner les protections des ordiphones ThéoBreizh.

TogDu signale que l'ensemble des outils, scripts, méthodes décrites ne peut être utilisé qu'en conformité à la loi du 10 a viz ebel 2017 concernant la protection des données personnelles. La présente analyse est couverte par le mandat accordé par la DGSIB.

TogDu rappelle que toute analyse ne peut être exhaustive et que le présent document reflète le travail effectué, de bonne foi, dans un temps contraint. Il ne saurait être le reflet que d'un travail équivalent, effectué dans un temps similaire.

Table des matières

Prise en main et déchiffrement du keystore :

1.1 Premières hypotheses

En suivant les conseils de la note technique PVRB2, on démarre la copie, effectuée par DGSIB, du matériel cible.

```
PS> bash.exe -c "qemu-system-aarch64 -nographic -machine virt,secure=on -cpu max -smp 1 -m 1024 -bios rom.bin -semihosting-config enable,target=native -device loader,file=./flash.bin,addr=0x04000000"
###############################################
#     virtual environment detected            #
#         QEMU 3.1+ is needed                 #
###############################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: keystore doesn't exist
ERROR:   KEYSTORE: Can't read keystore, reset keystore, try to boot again
```

On constate effectivement la création, sur le système hôte (via un hypercall qemu, voir 4.7), d'un fichier keystore, contenant deux entrées « KEY »

```
00000000  4B 45 59 00 00 00 00 00 01 00 00 00 00 01 00 00  KEY.............
00000010  D0 72 DD 3F 63 9C 2D 0E 1F F6 9A B9 65 78 2F 66  Ðr Ý?cœ-..ö šºex/f
00000020  8D 7B FF AC 76 6A EE AB 5D A8 4A E3 73 F8 C0 DC  .{ÿ¬vjî«]¨JãsøÀÜ
00000030  AA 18 C6 BD CB D5 51 BB 01 2F 42 AF C0 F3 FC 4A  ª.Æ½ËÕQ».¯B¯Àóü J
00000040  AD 43 8C 21 2F F8 A4 E5 97 05 3C D6 0B 54 AC 48  .CŒ!/ø¤å—.<Ö.T¬H
00000050  47 14 FB AE 05 9C 97 F8 13 8C ED E9 A8 00 60 D3  G.û®.œ—ø.Œíé¨.`Ó
00000060  C8 16 28 8F F2 C7 7D 82 08 1E 58 6F 73 EE 78 E0  È.(.òÇ},..Xosîxà
00000070  6F 69 D7 D5 75 96 AC F1 95 9D 6F 3E D0 0D 2C DD  oi×Õu–¬ñ•.o>Ð.,Ý
00000080  B1 1D EC F2 5A 37 9D EF 4D CE 3B 26 4A D7 E8 19  ±.ìòZ7.ïMÎ;&J×è.
00000090  EF BE AD DE 00 00 00 00 00 00 00 00 00 00 00 00  ï¾.Þ............
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000110  4B 45 59 00 00 00 00 00 03 00 00 00 80 00 00 00  KEY.........€...
00000120  C5 A8 7B BE 1D 22 9B DB 2B 67 28 11 59 8F 30 F5  Å¨{¾."›Û+g(.Y.0õ
00000130  0D C7 8E 4B E8 53 08 DC 87 6F 75 2D 9B D2 1E 2F  .ÇŽKèS.Ü‡ou-›Ò./
00000140  D1 42 60 17 18 71 5A 85 40 CC 6D F3 77 87 9D 7E  ÑB`..qZ…@Ìmów‡.~
00000150  B2 E8 16 A5 22 03 08 A9 64 09 A9 06 FF B6 7F DB  ²è.¥"..©d.©.ÿ¶.Û
00000160  B7 94 E6 31 A4 E0 81 19 2A A0 3F 5E 5B 76 77 37  ·"æ1¤à..* ?^[vw7
00000170  41 8C 8B 97 8B EE D6 E6 8B 44 0C 79 D2 57 81 75  AŒ‹—‹îÖæ‹D.yÒW.u
00000180  A4 4B 91 34 C7 12 B7 19 FD A4 C4 4F 7D FF 91 BF  ¤K'4Ç.·.ý¤ÄO}ÿ'¿
00000190  FC DB 45 5A BF 57 2B 6A 3B A3 37 92 C0 93 7B F9  üÛEZ¿W+j;£7'À"{ù
000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 1 : keystore - extrait

En redémarrant, comme demandé par le système cible :

```
###############################################
#     virtual environment detected            #
#         QEMU 3.1+ is needed                 #
###############################################
NOTICE:  Booting SSTIC ARM Trusted Firmware
KEYSTORE: AES Key is still encrypted, need decryption
KEYSTORE: Need RSA key to decrypt
KEYSTORE: RSA private exponent is not set, please set it in the keystore or enter hex value :
_
```

Nous nous retrouvons face à une demande de clé RSA privée, malheureusement perdue lors du dump effectué par le DGSIB (comme mentionné dans le PVRB1).

La trace de consommation fournie semble contenir une phase de calcul intensif (ci-après entre les barres rouges). En zoomant sur ce bloc, on observe assez distinctement deux formes de sinusoïdes :

- Une épaisse (première période sur le graphe de gauche)
- Une courte (12 périodes suivantes)

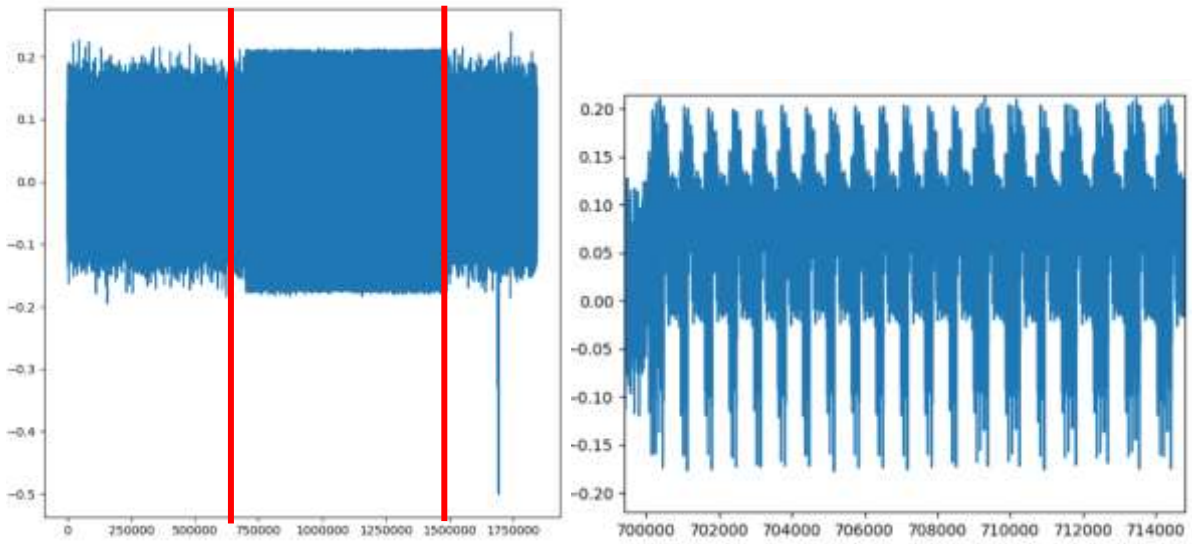


Figure 2 : trace de consommation + focus

On pourrait alors émettre l'hypothèse que l'implémentation du calcul RSA ne se protège pas contre une analyse des traces de consommation. La clé recherchée serait alors décrite par l'ensemble des périodes (et il y en a 1024…).

Une période « grasse » encodant un 1 et une période « courte » encodant un 0 (ou l'inverse). Nous aurions alors une clé commençant par :

```
1000000000000110111
```

Vérifions ces hypothèses en ouvrant 'ROM.bin' (le bootloader) dans IDA.

1.2 Retro ingénierie BL1

Le fichier « ROM.bin » fourni correspond à un dump du primo bootloader du matériel cible. Il s'agit d'un firmware « à plat », le vecteur de reset se trouve donc à l'offset 0 du fichier.

Le système cible utilise l'architecture ARM64 (aarch64), les outils disponibles (IDA PRO ou GHIDRA) permettent une analyse rapide de cette architecture, en proposant notamment des décompilateurs.

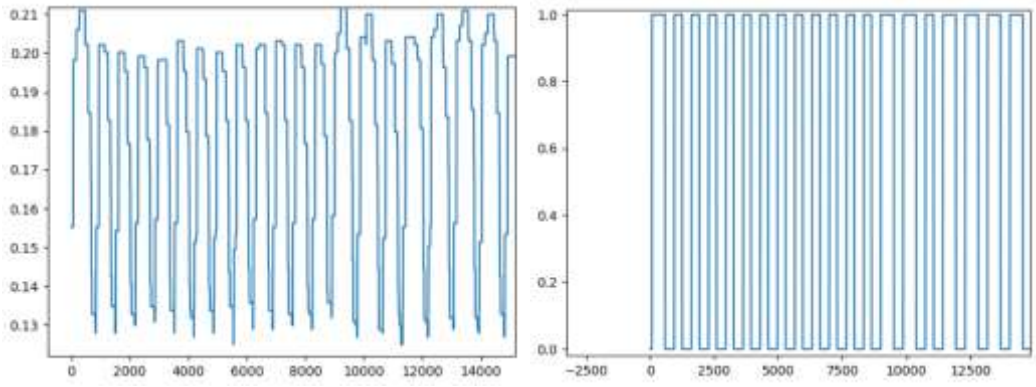
Le bootloader (que l'on nommera après BL1), étant assez riche en chaines de debug, on identifie rapidement :

- La fonction principale (à l'offset 1198), responsable de l'affichage de la chaine « virtual environment detected » ;
- La fonction (à l'offset DE8), responsable de la recréation du keystore (dont les clés AES sont hardcodées dans BL1), mais aussi de la lecture de la clé RSA (devant être fournie en hexa, et devant être de taille 0x100) ;
- La fonction de déchiffrement RSA (à l'offset 1470). On note que les clés sont inversées (représentation de l'endianness) ;
- Et enfin la fonction d'exponentiation modulaire en elle-même (à l'offset 1348). Cette fonction implémente un algorithme de type « exponentiation binaire », dont on trouvera un exemple en pseudo code sur Wikipedia[1].

La fonction d'exponentiation effectuant trois opérations (multiplication, modulo et affichage) supplémentaires lorsque le bit de la clé est à 1, cela explique les périodes « grasse » et « courte », confirmant nos hypothèses.

1.3 Récupération de la clé RSA

Il ne reste plus qu'à écrire un petit script python[2] pour filtrer le bruit présent (à l'aide de scipy.ndimage.filters.maximum_filter), puis transformer la sinusoïde en onde carrée (valeur supérieure à seuil).



Il ne reste plus qu'à mesurer la largeur des « bandes » pour reconstruire la clé privée RSA. Ce qui permet de déchiffrer les bootloaders suivant et d'accéder à un système Linux.

[1] https://en.wikipedia.org/wiki/Modular_exponentiation et
[2] TODO

```
KEYSTORE: RSA private exponent is not set, please set it in the keystore or enter hex value :
xxxxxx
KEYSTORE: Key read:
HEXDUMP :
------------------------------------------------
[REDACTED]
------------------------------------------------
KEYSTORE: Decrypting ...
[REDACTED]
Bravo, envoyez le flag [REDACTED]
à l'adresse challenge2019@sstic.org pour valider votre avancée
NOTICE:  Loading image id=1
NOTICE:  BL1: Booting BL2
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
------------------------------------------------
[REDACTED]
------------------------------------------------
NOTICE:  Loading image id=3
KEYSTORE: bad keystore magic
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
------------------------------------------------
44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44
44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44
------------------------------------------------
NOTICE:  Loading image id=4
KEYSTORE: BL2 got key : key_type:0x02, key_len:0x20
HEXDUMP :
------------------------------------------------
[REDACTED]
------------------------------------------------
NOTICE:  Loading image id=5
NOTICE:  BL1: Booting BL31
ERROR:   Secure-OS not not available : need decryption key
UEFI firmware (version  built at 00:01:39 on Feb 25 2019)
EFI stub: Booting Linux Kernel...
```

On note qu'il nous manque encore une clé pour permettre le déchiffrement d'un « Secure-OS ».

Déchiffrement du conteneur n°1

On trouve dans le dossier « /root/ » plusieurs conteneurs chiffrés, ainsi qu'un script python « get_safe1_key », un schéma logique (voir Annexe/ Pièce n°1), ainsi que divers scripts d'administration dans le dossier « tools ».

2.1 Composant physique de sécurité

La clé de déchiffrement du conteneur suivant est liée à un composant de sécurité matériel (non mentionné dans PVRB1 / clavier arrière des ThéoBreizh). Fort heureusement, les développeurs de cet ordiphone sécurisé ont omi de supprimer les docs de conception de ce composant, il nous a donc été possible d'en faire une copie logicielle.

Pour information, le composant de sécurité est composé de :

- Deux registres (commandés par les boutons poussoirs 3 et 4) permettant d'effectuer une rotation binaire de 1 sur les entrées A et B.
- Un registre (partie basse du schéma logique) permettant d'effectuer 4 opérations mathématiques distinctes entre A et B (ET logique, OU logique, XOR, …). L'opération effectuée dépendant de l'entrée OP et des boutons poussoirs 1 et 2 (XOR logique entre OP et BP1:BP2)

Ce composant est donc relativement simple à émuler.

2.2 Récupération de la clé

La séquence de boutons entrée est transformée par le composant de sécurité en une suite de 8 octets, dont le hash (SHA256) est ensuite dérivé (SCRYPT[3]) pour générer la clé de déchiffrement.

L'absence d'information concernant la séquence de boutons permettant le déchiffrement nous a conduits à effectuer une attaque par force brute, sur une séquence de 8 « caractères », chacun de ces caractères appartenant à un alphabet (distinct) de longueur 16.

Cette attaque par force brute permet de retrouver la clé de déchiffrement du conteneur n°1 en une petite heure sur un core I7 (sans optimisation)[4]

[3] Scrypt : https://tools.ietf.org/html/rfc7914.html

[4] Pourvu que l'on ait un python supportant la méthode scrypt ou qu'on ait pensé à logger la clé pré dérivation… Deux heures dans le cas contraire.

Déchiffrement du conteneur n°2 :

3.1 Prise en main

Le premier conteneur contient un binaire ELF permettant de valider la clé de déchiffrement du second conteneur.

```
# file decrypted_file
decrypted_file: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, BuildID[sha1]=5b5be1337d13c986d0e21441d771a36e41a34d17, stripped
# ./decrypted_file
Usage : ./decrypted_file <flag>
# ./decrypted_file a
Not good
```

Au premier abord, le binaire, une fois ouvert dans IDA PRO/GHIDRA semble bien vide. Le code peut d'ailleurs se simplifier ainsi :

```
int main(int argc, const char **argv, const char **envp)
{
        if ( argc != 2 )
        {
                printf("Usage : %s <flag>\n", argv[0]);
                exit(1LL);
        }
        try
        {
                Throw_Exception(argv[1]);
        }
        catch()
        {
                puts(reg_X28)
        }
}
```

On peut vérifier, par debug, que le registre x28 (pourtant jamais initialisé dans le binaire) pointe vers la chaine « not good », et que le code mort situé entre 402EC0 et 402F20 est bien mort. La clé n'est donc pas « congolexicomatisation », ce qui aurait été un poil trop évident.

On peut également constater que le binaire, très petit, ne semble, à priori, contenir aucun autre code. Sa seule particularité étant une section gnu_hash_chain (en 40024C) inhabituellement longue.

3.2 A la chasse au Korrigan

Après une première phase de perplexité, puis de recherche de code pouvant être situé ailleurs : le binaire faisant appel à la libstd et la libgcc, du code aurait pu être inséré dans ses deux bibliothèques sur l'équipement cible (ce qui n'est pas le cas). On se rabat sur une analyse du flot de données : où va l'entrée ? Qui la manipule ? Qui donc a bien pu initialiser le registre x28 ?

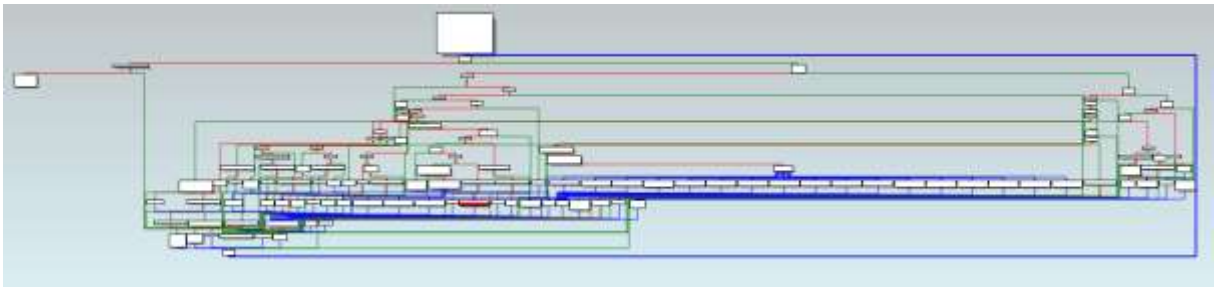
Une solution serait d'utiliser un outil de teinte de donnée automatisé, n'en connaissant aucun on se rabat sur GDB pour suivre la clé entrée en mémoire. Pour la partie visible, on observe qu'en 402E50, le pointeur vers la chaine entrée par l'utilisateur est transféré dans un buffer alloué par __cxa_allocate_exception. Ce buffer sera passé en paramètre lors du throw.

On va donc breaker en 402E50, puis poser des breakpoints matériels sur les valeurs contenues dans x0 (pointeur vers la chaine entrée) et x3 (buffer alloué).

```
Breakpoint 1, 0x0000000000402e50 in ?? ()
(gdb) i r x0 x3
x0             0xfffff85a1f0d
x3             0x701bef0
(gdb) awatch *0x701bef0
(gdb) awatch *0xfffff85a1f0d
Hardware access (read/write) watchpoint 3: *0xffffe6f4df0d

Value = 1230263123
0x0000ffff9aabad10 in ?? () from /lib64/libgcc_s.so.1
(gdb) x/10i $pc-8
   0xffff9aabad08:     b      0xffff9aababa4
   0xffff9aabad0c:     ldr    x6, [x6]
=> 0xffff9aabad10:     b      0xffff9aababa4
   0xffff9aabad14:     neg    x6, x6
   0xffff9aabad18:     b      0xffff9aababa4
   0xffff9aabad1c:     mov    x0, x25
   0xffff9aabad20:     mov    x1, x22
   0xffff9aabad24:     bl     0xffff9aab9acc
   0xffff9aabad28:     mov    x25, x0
   0xffff9aabad2c:     ldr    x0, [x29, #88]
```

Après avoir récupéré et ouvert la libgcc présente sur le matériel cible, on se retrouve dans une fonction (non nommée) dont le graphe semble assez caractéristique d'une machine virtuelle :



On observe en effet une boucle, un bloc déréférençant un pointeur (de code) vers une valeur, valeur (opcode) passant dans un switch (dans le cas d'une machine virtuelle, l'implémentation des différents opcodes), puis incrément de notre pointeur, avant de reboucler.

Nous allons donc poser un breakpoints à l'offset B928 (début du bloc déréférençant le pointeur de code émulé afin d'obtenir l'opcode courant).

```
(gdb) b *0xffff9aaba928
Breakpoint 6 at 0xffff9aaba928
(gdb) c
Continuing.

Breakpoint 6, 0x0000ffff9aaba928 in ?? () from /lib64/libgcc_s.so.1
(gdb) x/10i $pc-4
   0xffff9aaba924:     ret
=> 0xffff9aaba928:     ldrb    w7, [x0]
   0xffff9aaba92c:     add     x25, x0, #0x1
   0xffff9aaba930:     cmp     w7, #0x20
   0xffff9aaba934:     b.hi    0xffff9aabaa60  // b.pmore
   0xffff9aaba938:     cmp     w7, #0x1f
   0xffff9aaba93c:     b.cs    0xffff9aabaa88  // b.hs, b.nlast
   0xffff9aaba940:     cmp     w7, #0x10
   0xffff9aaba944:     b.eq    0xffff9aababe8  // b.none
   0xffff9aaba948:     b.hi    0xffff9aaba9d0  // b.pmore
(gdb) i r x0
x0             0x400269 4194921
(gdb) x/10x $x0
0x400269:      0x22080816      0x08160612      0x06122208      0x22080816
0x400279:      0x44280194      0x15031500      0x00492f03      0x8e1c240e
0x400289:      0x8302a68e      0x0e162765
```

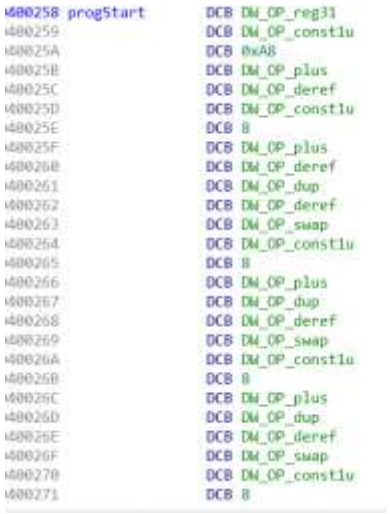
On observe que le registre x0 (notre pointeur de code émulé) pointe dans le fichier decrypted_file, et plus précisément dans le bloc elf_gnu_hash_chain… Nous avons donc identifié le code utile. Reste à comprendre le langage. Il semble, à priori, étrange que libgcc contienne une machine virtuelle, machine qui serait appelée lors du lancement d'une exception.

En recoupant le code de libgcc avec notre pile d'appel, on identifie le fichier unwind-dw2.c[5] et plus précisément la fonction execute_stack_op . Notre VM serait donc une VM DWARF… On peut se demander pourquoi et comment une VM a pu se glisser dans le mécanisme de gestion d'exception mais passons…[6]

3.3 Début de (la vrai) analyse

On pourrait, comme suggéré par James Oakley, commencer le debug de notre programme en instrumentant, via gdb, la VM DWARF implémentée dans libgcc. Mais le processus est fastidieux. Il semble plus simple d'écrire un rapide émulateur (voir annexe), qui nous permettra de poser hooks, tracepoints et autre facilités d'analyse.

Au passage le script accompagnant la présentation d'Igor Skochinsky à la Recon 2012 « Compiler Internals: Exceptions and RTTI »[7], contient une structure ENUM pouvant être plaquée sur le bloc elf_gnu_hash_chain afin d'obtenir un programme DWARF lisible.



Après diverses tentatives pour recréer la logique de ce programme, en identifiant, à la main, les différentes boucles :

```
400258 progStart
40027B =>if size != 32 4002C2
400283 JMP g_prog_StartCrypto

4002CE g_prog_StartCrypto
lbl:4002CF
4002D6 JMP 40030B

40030B hashFunc
internal loop (40030C - 40039B ) * 32
…
```

[5] https://github.com/gcc-mirror/gcc/blob/master/libgcc/unwind-dw2.c
[6] "Exploiting the Hard-Working DWARF:Trojan and Exploit Techniques WithoutNative Executable Code", James Oakley, aura été d'une grande aide. (Https://www.cs.dartmouth.edu/~trdata/reports/TR2011-688.pdf)
[7] http://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf

On commence à comprendre, petit à petit, la logique de ce programme. On identifie la taille d'entrée attendue (32 charactères), des tables de lookups, des blocs ressemblant à de la crypto ou à un hash. Mais rien qui ne semble connu (les différentes constantes ou tables ne semblant pas correspondre à des constantes connues).

On identifie que notre entrée, après être passée dans ce truc cryptographique, doit être égale (enfin par QWORD) aux valeurs suivantes :

```
a = 0x65850b36e76aaed5
b = 0xd9c69b74a86ec613
c = 0xdc7564f1612e5347
d = 0x658302a68e8e1c24
```

On profite aussi de notre émulateur pour sortir une trace « concrète » des opérations « importantes » (opérations mathématiques, branchements conditionnels, valeur contenues dans la pile à différents points) . Par exemple en toute fin du programme on a :

```
0458 : shl (4fe30b84 << 20 = 4fe30b8400000000)
0459 : or (286fcdda | 4fe30b8400000000 = 4fe30b84286fcdda)
045a : xor (aea142e3b91b8530 ^ 4fe30b84286fcdda = e1424967917448ea)
0461 : plus (e + 1 = f)
0467 : min (f - f = 0)
0477 : bra 047d
047d : skip 02f4
02fc : plus (3 + 1 = 4)
02ff : min (4 - 4 = 0)
[TRACE] EndLoop
3131313130303030
3333333332323232
…
028e : xor (e1424967917448ea ^ 658302a68e8e1c24 = 84c14bc11ffa54ce)
```

En commençant à suivre, en partant de la fin, les valeurs ayant aboutis à notre sortie (ici aea142e3b91b8530 et 4fe30b84286fcdda) on commence à prendre conscience que tout ceci semble inversible, et que beaucoup de code ne soit en fait que de l'obfuscation de constantes. Et on commence à réimplémenter, en python, le programme DWARF (voir annexe).

Une fois implémenté, il n'y a plus qu'à inverser l'algorithme pour obtenir la clé de déchiffrement du conteneur .

Déchiffrement du conteneur n°3

4.1 Un bien gros buffer

Comme dans le précédent conteneur, nous nous retrouvons face à un exécutable ELF, prenant une clé en paramètre :

```
# file *
decrypted_file: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux), statically linked, for GNU/Linux 3.7.0, stripped
# ./decrypted_file
usage: ./decrypted_file [32-bytes-key-hex-encoded]
# ./decrypted_file aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaa
Loose
```

On note que le temps de calcul pour aboutir au message « Loose » est relativement long (quelques secondes).

En ouvrant decrypted_file dans IDAPro, on observe un code relativement simple :

- La taille de l'entrée utilisateur est vérifiée, puis cette entrée est désencodée ;
- Le programme envoie au device /dev/sstic un buffer de taille 0x101010, probablement chiffré, à l'aide de l'IOCT C0105300 ;
- Le programme envoie la clé au device /dev/Sstic à l'aide de l'IOCTL C0105301 ;
- Puis une boucle alterne IOCTL C0105302 et C0105303, jusqu'à ce que le module kernel signale une erreur, ou que la clé soit fausse ou juste.

Direction donc le module kernel implémentant le device /dev/sstic

4.2 Le passe plat

Après avoir récupéré le fichier sstic.ko dans le dossier lib du matériel cible, on ouvre ce module dans IDAPro. On se rend vite compte que ce module se contente de faire passe plat entre des programmes utilisateurs (comme safe_02/decrypted_file, mais aussi comme l'ensemble des utilitaires permettant la gestion des clés et le déchiffrement de conteneurs) et le Moniteur Sécurisé ARM. On observe en effet que :

- L'ioctl C0105300 est transformé en appel SMC 83010004 ;
- L'ioctl C0105301 est transformé en appel SMC F2005003 ;
- L'ioctl C0105302 est transformé en appel SMC F2005001 ;
- L'ioctl C0105303 est transformé en appel SMC F2005002.

Le déchiffrement du dernier conteneur nécessite donc une analyse plus poussée de la chaine de boot, afin d'identifier le moniteur sécurisé et sa fonction gérant les appels SMC.

On peut aussi noter, en redémarrant la machine virtuelle, que nous avons débloqué le démarrage d'un « Secure-OS ».

4.2 Reconstruire la chaine

Les différents étages de la chaine de boot étant chiffrés, il nous faut d'abord récupérer un dump mémoire, en clair, des différents composants (d'après les logs de démarrages, BL2, BL31 et BL32).

Une première façon, simple, de récupérer ces dumps est de profiter du fait que la chaine de boot va, par design, déchiffrer les différents composants. Composants qui resteront, pour pouvoir être utilisés, en clair dans la mémoire du système cible. Dans notre cas, ils resteront, en clair, dans la mémoire de QEMU. Mémoire accessible (bien que QEMU, soit un processus PICO/LXSS) via des outils comme ProcessHacker.

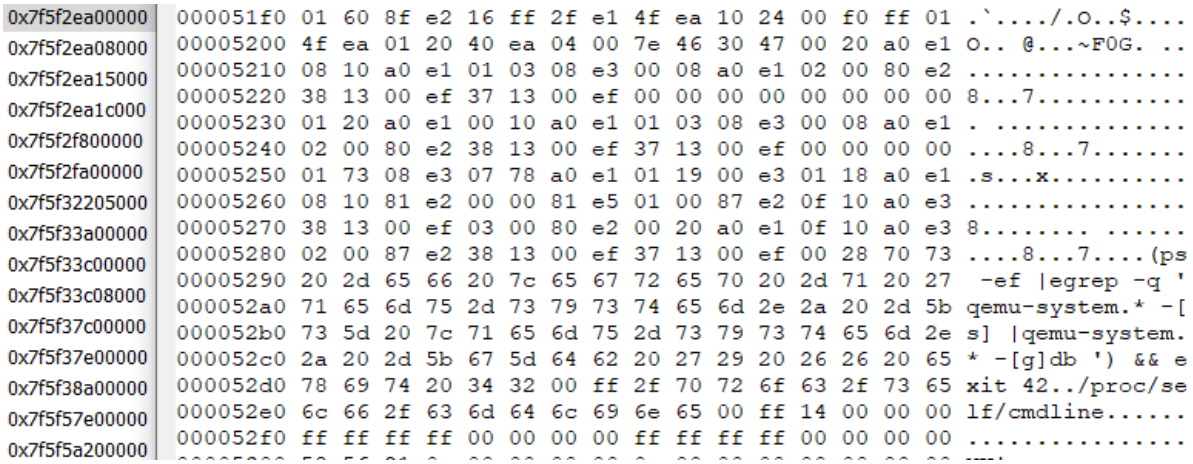


Figure 3 : dump mémoire du Secure-OS

On perd cependant toute notion de l'agencement mémoire sur le matériel cible.

Une deuxième approche est de brancher un débuggeur sur QEMU (en spécifiant les options -S -s) afin de pouvoir debugger la chaine de boot elle-même. On identifie ensuite la fonction de déchiffrement de BL2 dans BL1 (à l'offset 2168) pour y poser un point d'arrêt et récupérer BL2 en clair. BL2 sera chargé à l'offset E00B000, on peut donc enrichir notre base d'analyse avec le nouveau dump mémoire.

La même analyse sur BL2 permet d'identifier une nouvelle primitive de déchiffrement (à l'offset E00C584) et de récupérer le contenu de BL31. On peut noter qu'une structure, en E014000, décrit la taille et le futur emplacement de BL31 (en E030000). On peut également récupérer le contenu de BL32 (le Secure-OS), qui sera chargé en E200000.

L'analyse de BL31 permet de constater que ce dernier s'enregistre comme moniteur TrustZone (en initialisant notamment, le registre MSR vbar_el3). On tient donc le composant qui devrait gérer nos appel SMC.

4.3 ARMv8: gestion d'exception

L'architecture ARMv8 utilise, pour sa gestion d'exception un vecteur de gestionnaire d'exception, en différenciant 4 types d'exceptions (Synchrone, IRQ, FIQ et SError), le trusted level source (inférieur ou égal), l'architecture source (aarch63 ou aarch32) et le pointeur de Stack à utiliser (trustzone ou pas). Chaque gestionnaire d'exception ayant 0x80 octets disponibles[8].

Il faut noter qu'IDA pro ne décode pas les registres MSR, contrairement à GHIDRA.

Dans notre cas, le code kernel s'exécutant en EL1, et étant du AArch64, les appels SMC seront gérés par le handler situé à vbar_el3+0x400 (donc par la fonction située en E036C00).

On observe une table (en E0386F0), contenant des handlers pour différentes classes d'appel SMC :

- arm_arch_svc : SMC de type 8000XXXX;
- std_svc : SMC de type 8400FXXX ;
- oem_svc : SMC de types 8301XXXX et 8300FXXX ;
- tspd_std et tspd_fast : SMC de type F200XXXX.

Notre premier ioctl (C0105300, transformé en SMC 83010004) serait donc géré par oem_svc. Les 3 suivants seraient gérés par tspd_std.

SMC 83010004

Le handler pour cet appel va copier, après vérifications, le buffer à l'adresse E053000. On notera qu'il ne sert à rien de poser des breakpoints hardware dans cette zone, le seul code y accédant étant ce handler SMC.

On sait cependant, par observation, que ce très gros buffer sert à quelque chose, ne pas le passer ou le modifier entraine un message de type « failure »… On reviendra sur ce point plus tard.

[8] http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/CHDEEDDC.html, AArch64 exception table

4.4 SMC F2005001 à F2005003

Le handler pour ces trois appels va effectuer une transition vers le Secure OS (trustzone), plus précisément par la fonction située en E200C08.

- F2005001 effectue en fait un appel SMC 83010001 (qui sera traité par BL31::oem_svc) puis met à jour une global ;
- F2005002 est gérée par un switch, sur la valeur globale mise à jour par F2005001. Les différentes valeurs donnent lieu à divers appels à BL31::oem_svc (sur des codes de type 830100XX)…
- F2005003 va créer un SEGFAULT, géré par le handler d'exception situé en E203204 (EL courants, exception dans la trustzone). Puis faire appel à SMC 83010003 et SMC 83010002. Le gestionnaire de SEGFAULT va effectuer un ensemble d'opérations SM4 (un algorithme de cryptographie chinois[9])…

Et on commence donc à doucement paniquer…

4.5 Gestion des segfaults

On note qu'IDA Pro ne gère pas les instructions du coprocesseur cryptographique, GHIDRA permet d'en décoder la plupart (mais pas toutes…), notamment les instructions sm4e (chiffrement sm4), aese et aesd (chiffrement/déchiffrement AES), et autres instructions flottantes utilisées par BL31::oem_svc…

La fonction de BL32 gérant les segfault (en E200E84) va déchiffrer, en utilisant une clé hardcodée, une valeur contenue dans la section mémoire 413000, à l'offset ayant provoqué une faute.

Dumpons donc cette section ! Nous nous retrouvons avec un bloc mémoire dont les 0x1000 premiers bytes sont strictement identiques à ceux de notre programme virtuel, la suite est différente…

4.6 Le moment de doute

Il est temps de faire une pause, pour réfléchir à la problématique et essayer de récapituler un peu…

Intuitivement, on se dit que notre gros buffer est un programme de machine virtuelle, très probablement chiffré, mais il disparait, à priori sans être touché, dans la section E0530000.

On se doute aussi que l'ioctl C0105302 (et donc le SMC F2005001) soit une sorte d'instruction step. On peut donc supposer que notre global soit l'instruction courante (par expérimentation, les valeurs ne correspondant pas à celles dans notre programme, donc très certainement déchiffrées).

Il reste donc l'ioctl C0105303/SMC F2005002, qui serait une instruction « exécute l'opcode courant ». On aurait donc une partie de notre machine virtuelle dans BL32 (la partie gestion et décodage des opcodes), une autre dans BL31 (l'implémentation des opcodes)…

Reste que nous ne savons pas comment le programme virtuel est transmis de BL31 à BL32, ce dernier ne touchant pas à la section E053000… Et si le code a magiquement été copié en 0x413000, d'où viennent les données situées après 0x414000 ?

[9] https://en.wikipedia.org/wiki/SM4_(cipher)

4.7 Anti-debug : au fait, j'exécute des commandes sur ta machine hôte.

A force de tourner en rond, on en revient à chercher ailleurs, notamment à ce qu'on n'a pas fait : regarder la fonction reset de BL32.

On observe (en E2017AC) un appel à une fonction servant à déclarer des alias mémoire (ou à remapper la mémoire ?), remappant les premiers 0x1000 octets de la section E0530000 sur la section 413000 ! Juste ce qu'on cherchait ! On bénit au passage l'absence de crossref et la manie d'ARM de séparer les assignations de registres en deux :

```
MOV     X1, #0x3000
MOV     X0, #0x3000
MOVK    X1, #0x41, LSL#16
MOVK    X0, #0xE05, LSL#16
```

En fin de bloc assembleur, X0 vaut E053000, x1 vaut 413000. Pas de cross référence possible sous IDAPro…

On observe également que la suite du buffer en 0x413000 dépend du résultat de la fonction E201650 (ce qui corrobore notre observation).

L'analyse de E201650, permet de constater que BL32 va exécuter (via l'instruction HLT F000) une ligne de commande (légèrement obfusquée) sur notre machine hôte… C'est le même mécanisme qui est utilisé pour créer, lire et écrire dans le keystore.

BL32 va, dans un premier temps, essayer de lire le fichier /proc/self/cmdline, afin de chercher les options '-s' ou '-gdb' (présence du mod debug).

Si ce fichier n'est pas présent BL32 exécutera la commande :

```
(ps -ef |egrep -q 'qemu-system.* -[s] |qemu-system.* -[g]db ') && exit 42
```

Si les options '-s' ou '-gdb' sont présentes, la fonction retourne 1, sinon 0.

A partir de ce point, on ajoute un breakpoint (en E2017B4) dans gdb permettant de changer la valeur de X0 à 0 (ce qui permet de bypasser le check).

4.8 Analyse de la VM : partie BL32

L'analyse de la fonction E2005A4 permet de récupérer le format des opcodes de notre VM :

17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
OPC1				OPC2		REG1				REG2				PARAM									

- OPC1 et OPC2 désignant l'opération ;
- REG1 et REG2 les registres destination et source ;
- PARAM une valeur immédiate.

On instrumente ensuite un breakpoint GDB (placé par exemple en E200C80) pour récupérer notre programme virtuel déchiffré.

```
(gdb) b *0xE200C80
Breakpoint 2 at 0xe200c80
(gdb) commands 2
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
>printf "%08x\n", $x0 & 0xFFFFFF
>if $i < $max
 >set $pc = 0xE200C78
 >set $i = $i+3
 >set $x0 = $i
 >c
 >end
>end
(gdb) set $pc = 0xE200C78
(gdb) set $i = 0
(gdb) set $max = 0x1000
(gdb) c
Continuing.

Breakpoint 2, 0x000000000e200c80 in ?? ()
002d0002
004d0010
002d0020
000f4010
004f4010
002f4020
```

On peut d'ailleurs faire varier la valeur de $max pour couvrir l'ensemble du programme virtuel.

4.9 Analyse de la VM : partie BL31

L'analyse de la fonction E032014 permet d'identifier les opérations des différents opcodes. Encore une fois, cette fonction utilisant massivement des instructions ARMv8 non décodées par IDAPro, il est préférable de passer par GHIDRA de temps à autres.

L'analyse de E032014 permet de déterminer que cette VM est (contrairement à celle de la précédente étape) une machine à registres (qui sont stockés chiffrés). On identifie 14 registres généraux et un registre « PC » (ou r15). Ces registres sont chiffrés et déchiffrés en AES à la demande.

Typiquement, le SMC F2005001, qui lui-même fait un SMC 83010001 (F), va en fait permettre à BL32 de récupérer la valeur courante (et déchiffrée) du registre PC.

On identifie ainsi les opérations suivantes :

OPC1	OPC2	Mnémonic	Description
C	XX	MOV [0x10000+R1], cst	Initialisation d'un secret
A	XX	RET r0	Fin du programme
D	XX	OP_d	Non géré par E032014
E	XX	OP_e	Non géré par E032014
0	0	OP_0_0 R1,R2	Non géré par E032014
1		DEC R1	
2		ADD R1, R2	Utilise l'instruction fcadd#270
3		SUB R1, R2	Utilise l'instruction fcadd#90 (rotation d'un vecteur => négation de R2)
6		XOR R1, R2	
B		OP_B_0 R1	Non géré par E032014
0	3	OP_0_3 R1, Imm	Non géré par E032014
2		ADD R1,Imm	
3		SUB R1,Imm	
4		OP_4_3 R1,Imm	Non géré par E032014
5		OP_5_3 R1,Imm	Non géré par E032014
6		OP_6_3 R1	Instruction non présente.
7		AND R1, Imm	
8		CACHE R1	Utilité non déterminée
9		OP_9_0 R1,Imm	Non géré par E032014
xx	1	MOV R1, [R2]	
xx	2	MOV [R1], [R2]	

4.10 Retour à BL32 : passage en aarch32

Les instructions manquantes sont gérées par E2005A4, et ont pour point commun un appel à la fonction E2025A4. Cette fonction va, en modifiant le registre spsr_el1 provoquer un passage en 32bits, et utiliser le registre elr_el1 pour modifier l'adresse de destination de l'instruction ERET, afin de sauter sur la fonction aarch32 passée en paramètre.

Les fonctions aarch32 utilisent l'appel SVC 1338 pour effectuer un SMC call vers BL31 (on retrouve les SMC précédents), et l'appel SVC 1337 pour revenir au code aarch64. On peut se reporter à la fonction E203600 pour avoir les détails d'implémentation.

L'analyse des différentes fonctions aarch32 (certaines passant en thumb mode) permet de reconstruire les opcodes manquant :

OPC1	OPC2	Mnémonic	Description
C	XX	MOV [0x10000+R1], cst	Initialisation d'un secret
A	XX	RET r0	Fin du programme
D	XX	OP_d	Non analysé
E	XX	OP_e, Imm	Non analysé, probablement un saut conditionnel
0	0	MOV R1, R2	
1		DEC R1	
2		ADD R1, R2	Utilise l'instruction fcadd#270
3		SUB R1, R2	Utilise l'instruction fcadd#90 (rotation d'un vecteur => négation de R2)
6		XOR R1, R2	
B		SWAP R1	Inverse les deux octets de R1 (0xAABB => 0xBBAA)
0	3	MOV R1, Imm	
2		ADD R1,Imm	
3		SUB R1,Imm	
4		LSH R1,Imm	Utilise du code 16bits
g		RSH R1,Imm	Utilise du code 16bits
6		OP_6_3 [R1]	Instruction non présente.
7		AND R1, Imm	
8		CACHE R1	Utilité non déterminée
9		JA R1,Imm	Saut conditionnel si R1 > 0
xx	1	MOV R1, [R2]	
xx	2	MOV [R1], [R2]	

Les instructions D et E utilisent une SEGFAULT en 0x901000 0x9010008 pour obfusquer leurs codes. Leur comportement exact nous semble, à ce stade, non pertinent, on laisse donc cette analyse de côté. On peut juste constater que OP_E est un saut conditionnel, on suppose alors un autre mécanisme d'anti debug, on choisit alors de ne pas tenir compte de ce saut.

4.11 Dit pourquoi mon python il ne marche pas ?

Arrivé à ce stade, nous avons tout ce qu'il faut pour commencer la ré-implémentation de notre programme virtuel en python.

On constate que les 4 QWORDs (séparés en DWORD) constituant le flag sont chiffrés de manière indépendante et qu'il va nous falloir récupérer la suite de notre fichier programme, qui constitue en fait une très grande table de lookup. C'est cette table qui aurait été corrompue si l'on n'avait pas passé le test d'anti-debug décrit en 4.7.

Le reste de l'algorithme étant inversible, on se lance dans cette tâche, pour enfin obtenir un script python permettant d'inverser notre programme (le secret étant initialisé par 0P_C). Au passage on notera que le ROT13 du secret est une adresse mail (bien entendu fausse).

Et tout heureux, on passe notre clé à decrypted_file, qui nous répond :

```
# ./decrypted_file adac8baa555b6f30c6b3c3dfd1b207c8087044465f22d7eb911aaa89ec260
e74
Loose
```

On passe donc une soirée à vérifier et revérifier son implémentation, à vérifier et revérifier l'inversion de son algo, à effectuer des traces concrètes (en posant des breakpoints à des endroits choisis de E032014) qui ne donnent rien car au bout de quelques cycles, l'instruction OP_E de ce bloc :

```
0000010e 009e012f JA r8 012f
        00000111 00020c00 MOV r8, r3
        00000114 0000f800 MOV r3, r14
        00000117 002cc001 ADD r3, 0001
        0000011a 0060c000 XOR r3, r0
        00ecc14a OP_E 14A
        00000120 0060c400 XOR r3, r1
        00000123 00002400 MOV r0, r9
        00000126 00004800 MOV r1, r2
        00000129 0000a000 MOV r2, r8
        0000012c 008c014a CACHE r0
```

devient bloquante et provoque un saut des instructions 120 à 12C. Or l'algorithme semble tout de suite bien plus dur à inverser sans ces instructions. On constate lors d'une seconde trace que le passage de OP_E en bloquante semble intervenir au bout de plus ou moins de boucles…

Le non déterminisme n'ayant pas sa place dans cette histoire (comment obtenir une clé si celle-ci change aléatoirement ?) on se penche un peu plus sur OP_E.

4.12 Anti debug n°2 : une histoire de durée

Une analyse plus poussée de l'instruction OP_E permet de se rendre compte qu'elle effectue une soustraction entre le temps courant et une valeur précédemment enregistrée (par OP_D ?), la différence entre les deux valeurs ne devant pas être supérieur à 5.

Lors de nos traces concrètes, on utilise de nombreux breakpoints pour tracer les valeurs de registres. Ce qui ralentit l'exécution (ralentissement dépendant de la position et du nombre de breakpoints, d'où l'aspect non déterministe) et provoque le saut vers 14A.

On choisit alors de patcher sauvagement l'instruction aarch32 en E2050E4 pour modifier la valeur seuil à 0xFFFF. Ce qui nous permet d'effectuer une nouvelle trace concrète, cette fois représentative.

4.13 Ne soit pas stupide

Et on se rend compte que la sortie du programme pour le premier DWORD n'est pas 612E7270 mais 7270612E. Le second DWORD n'est pas 6766722E mais 722E6766… Bref, qu'on a juste pris en compte une instruction SWAP de trop…

Le patch de notre python est immédiat, ce qui nous donne la clé (correcte) du dernier conteneur. Ça y est, le matériel cible est déchiffré. On prévient la DGSIB et on file donner un cours, dont la préparation aura sauté dans la bataille.

PUBLIQUE / BZH SEUL

Et au fait ! Vous n'auriez pas oublié pourquoi on est là ?

PVRB3 : Message du 17/04/2019 (17 a viz ebel 2019) DGSIB

> Merci pour le message. Vous avez pu récupérer des preuves du coup ?
>
> M.[redacted]

On l'avoue, cette histoire de marée brune nous était un peu sortie de la tête. Et puis on était quand même en train de donner un cours. On profite d'une vieille ruse (un peu éculée, mais toujours au combien efficace) de prof voulant dégager du temps perso : « Bon je vous laisse faire ce TP ? », pour ouvrir le dernier conteneur.

On y trouve un système de fichier Android on ne peut plus classique. Le regard des élèves se faisant suspicieux, on se rabat sur de la preuve facile, les SMS et le fichier mmssms.db. On en extrait les conversations suivantes (les noms proviennent de contact2.db):

+33612345678 (Mister X03)
2018-05-15 09:05:12.370 :
[<=] Bonjour. Une action d'ampleur est prévue pendant la conférence SSTIC le 13 juin après-midi et dans la soirée. Pouvez-vous profiter du rassemblement d'un grand nombre d'experts pour essayer de les mettre hors d'état de nuire
2018-06-01 07:18:17.947
[=>] OK, l'opération suit son cours, je vous confirmerai les résultats.
2018-06-13 21:52:30.418
[=>] Je vous confirme le semi-succès de l'opération "Ça coule de source". Je n'ai malheureusement pu avoir accès qu'à la moitié des préparations et je n'ai touché que la moitié du public.
2018-06-14 07:22:26.503
[<=] OK, félicitations tout de même. À renouveler l'an prochain avec plusieurs infiltrés.
2019-03-29 11:40:24.928
[<=] Bonjour, Cette année nous allons profiter du social event pour effectuer nos attaques ciblées. Merci de recruter en conséquence pour atteindre le maximum de personnes
2019-04-01 07:01:17.892
[=>] Bien noté, je pense cibler le stand des huîtres pour plus de discrétion et d'efficacité.

+33612345679 (SuperPharma)
2018-05-24 23:06:29.336
[<=] Bonjour. Votre commande pour les produits "DUPHALAC 1L" (10 articles) a bien été expédiée. Vous pouvez suivre votre commande à l'adresse suivante : megasuivi.com/abXdSS
2018-05-28 16:06:12.360
[<=] Bonjour. Merci d'avoir réceptionné vos produits. Vous pouvez noter votre article à l'adresse superpharma.com/duphalac-1l.html

+33642424242 (Kevin Leserveur)
2018-06-01 09:43:19.214
[=>] Bonjour Kévin. Désolé finalement nous n'avons plus besoin de vous le 13/06/2018, car nous avons déjà trop de personnel. Nous vous recontacterons pour une future mission.
Agence Interim'expert

PUBLIQUE / BZH SEUL

+33623232323 (Traiteur SSTIC)
2018-06-10 06:52:24.872

[=>] Bonjour, l'agence Interim'expert m'a contacté pour remplacer Kevin qui est souffrant pour le service du 13/06/2018 à la Halle Martenot. Pouvez-vous me préciser les horaires.

2018-06-10 10:11:12.535

[<=] Bonjour, merci d'être présent à partir de 9h30 le 13/06/2018. Vous serez affecté au poste des plats chauds.

2018-06-10 10:11:26.545

[ENV] Bien reçu. Bonne journée.

+33601020304 (Mister X02)
2019-02-02 10:45:04.469

[<=] Bonjour, nous avons identifié un profil idéal pour l'usurpation d'identité dans le cadre du plan visant à focaliser l'attention des pentesteurs. Vous trouverez les identifiants du blog et du compte twitter sur notre outil interne.

2019-02-02 10:45:23.308

[=>]Bien reçu, je lance la campagne tout de suite.

2019-02-13 15:50:04.515

[<=] Bravo. Ce billet "Pentest et pentesteur" nous a paru un peu gros lors de la publication, mais cela semble marcher au-delà de nos espérances. Continuez à alimenter la conversation, nous avons déjà constaté une perte d'attention chez la plupart des gens visés.

+33689888786 (Mister X01)
2019-03-01 08:32:05.449

[<=] Bonjour, comme chaque année, l'opération "challenge SSTIC" se prépare avec des attaques ciblées courant la première semaine d'avril. Pouvez-vous faire en sorte de focaliser l'attention à ce moment-là.

2019-03-15 21:07:20.324

[=>] Mission accomplie, comme d'habitude la perspective d'une tournée de shooter au cactus a suffi à corrompre le CO et à chosir la date de publication du challenge. Pour être sûr que les experts sont toujours occupés, j'ai redirigé l'adresse 9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org vers votre boîte mail. Tant que vous ne voyez passer aucun mail, la voie est libre...

«Ça coule de source » démasquée

TogDu a fourni l'ensemble des preuves à la DGSIB le 17/04/2019 (17 a viz ebel 2019), comme mentionné dans la note technique TOGDU/IMPSALIVE/ 00-0319-1897X1/CR1 a destination unique de la DGSIB.

Comme convenu, le présent rapport technique est fourni avec l'ensemble des scripts et outils permettant à la DGSIB de déchiffrer les ordiphones de types ThéoBreizh (de génération semblable à celle du matériel cible). Ces données techniques seront mises en ligne, avec l'accord de la DGSIB sur github[10]

[10] https://github.com/TogDu/SSTIC_2019

ANNEXES

PVRB1: Mandat DGSIB – message du 24/03/2019

Bonjour,

Récemment un individu au comportement suspect nous a été signalé. Il semblerait qu'il s'attaque à la communauté
sécurité informatique française avec notamment l'intention de lui nuire.

Sans preuve, il est difficile d'agir à son encontre. Ainsi, nous avons décidé de saisir son téléphone portable
afin de collecter des éléments confirmant nos hypothèses. Cependant son téléphone semble posséder plusieurs couches
de chiffrement qui nous empêchent d'accéder à ses données.

Dans l'incapacité de contourner ces systèmes de chiffrement, nous avons décidé de faire appel à vous pour nous aider.
Nous avons consacré du temps à rendre possible le démarrage du téléphone sécurisé dans un environnement virtualisé.
Malheureusement le coffre de clef du téléphone ciblé n'a pas pu être copié. Avant de devoir restituer le téléphone,
nous avons été en mesure d'enregistrer une trace de consommation de courant lors du démarrage du téléphone. Nous
espérons que cela pourra vous être utile.

Des instructions techniques plus précises vous seront fournies.

Bonne chance pour votre mission et nous comptons sur vous pour nous communiquer toutes les preuves que vous pourrez
trouver au cours de votre investigation à l'adresse mail suivante : challenge2019@sstic.org.

La communauté sécurité informatique française dépend de vous !

PVRB2: informations techniques (24/03/2019)

Pour démarrer le téléphone dans un environnent virtuel vous devez utiliser qemu dans une version supérieure ou égale à 3.1.

La ligne de commande suivante permet d'amorcer le système :

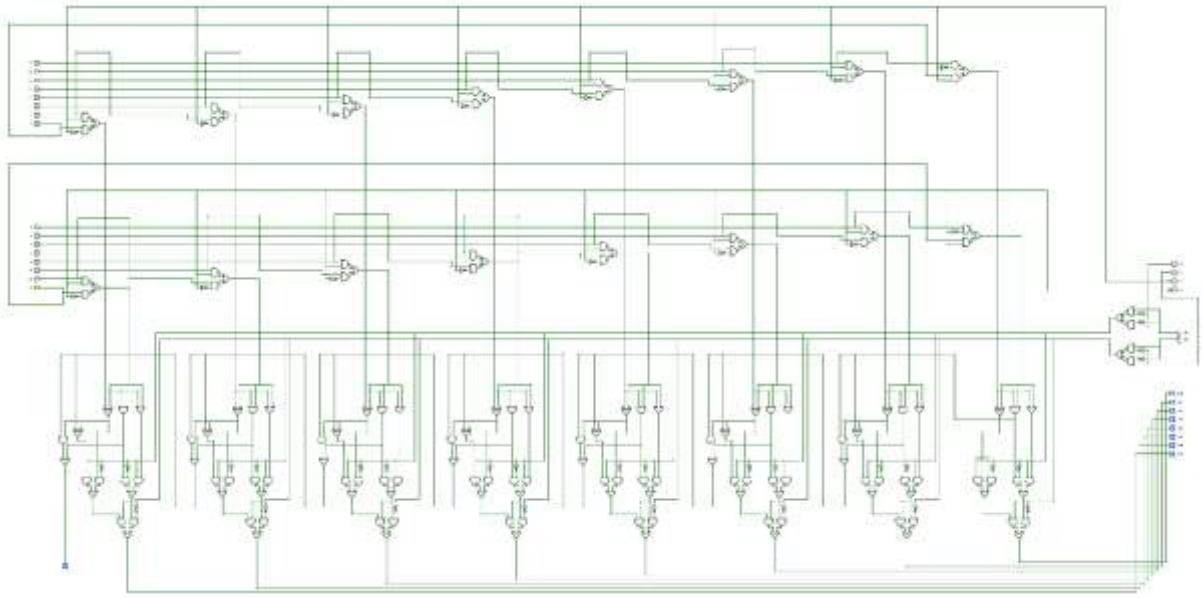
qemu-system-aarch64 -nographic -machine virt,secure=on -cpu max -smp 1 -m 1024 -bios rom.bin -semihosting-config enable,target=native -device loader,file=./flash.bin,addr=0x04000000

En ajoutant les options suivantes, vous pouvez obtenir un accès SSH dans le téléphone virtuel (sur le port 5555 de votre localhost):

-netdev user,id=network0,hostfwd=tcp:127.0.0.1:5555-192.168.200.200:22 -net nic,netdev=network0

Bonne chance

Pièce n°1 : schéma logique d’un composant de sécurité



PUBLIQUE / BZH SEUL