Solution du challenge SSTIC 2019

Stanislas Arnoud

29 mars 2019 - 12 avril 2019

Table des matières

1 Préambule

1.1 Informations générales

En premier lieu, afin de ne pas surcharger cette solution de longues pages de code source de mauvaise qualité, ce fichier est un *fichier polyglote*.[1] Plus précisemment, c'est un fichier *pdf* valide, qui peut etre lu par la plupart des lecteurs *pdf*. C'est également un fichier au format *zip*, contenant l'ensemble des scripts utilisés pour la résolution du challenge. Malheureusement, certains *utilitaires* de décompression ne considèrent pas le fichier comme valide.[2] C'est pour cette raison que ce fichier est également une *ROM AARCH64*, qui permet la copie du fichier *zip* dans le répertoire courant lorsqu'elle est lancée avec *qemu*[3].

[1] L'auteur s'étant inspiré de la solution de David Bérard pour le challenge SSTIC 2018, ainsi que des travaux d'Ange Albertini.

[2] Utilitaires acceptant le fichier: `unzip`, `WinRAR`, ...

[3] `qemu-system-aarch64 -nographic -machine virt,secure=on -cpu max -bios sstic_2019_stanislas_arnoud.pdf -semihosting-config enable,target=native`

Plusieurs outils et logiciels ont été utilisés pour résoudre ce challenge: *qemu*, *gdb*, *ghidra*, *keystone-engine*, *miasm* et *z3*. Les scripts ont été écrits en *C* et *python3*.

1.2 Présentation du challenge

Le challenge démarre avec le texte suivant:

```
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 hyp
othèses. Cependant son téléphone semble posséder plusieurs couches de chiffremen
t qui nous empêchent d'accéder à ses données.

Dans l'incapacité de contourner ces systèmes de chiffrement, nous avons décidé d
e faire appel à vous pour nous aider. Nous avons consacré du temps à rendre poss
ible le démarrage du téléphone sécurisé dans un environnement virtualisé. Malheu
reusement le coffre de clef du téléphone ciblé n'a pas pu être copié. Avant de d
evoir restituer le téléphone, nous avons été en mesure d'enregistrer une trace d
e 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 !
```

Tout au long du challenge, nous allons devoir déchiffrer les différentes couches de chiffrement du téléphone. Le but final étant de récupérer une adresse mail, probablement présente dans le téléphone déchiffré. Le fichier contenant le challenge SSTIC 2019 est téléchargeable ici.

2 Étape 1 - Analyse statique de consommation sur *RSA*

2.1 Récupération de l'exposant

L'archive téléchargée contient 4 fichiers:

- `README`: Une brève information nous indiquant la version de `qemu` à utiliser ainsi que la ligne de commande à lancer pour faire démarrer le système.
- `flash.bin`: Un fichier à l'entropie élevée, probablement chiffré.
- `rom.bin`: Le fichier de *boot* du téléphone.

- `power_consumption.npz`: Une archive zip contenant un fichier au format `NumPy`, représentant la consommation électrique du téléphone au démarrage.

Au lancement de la commande décrite par le fichier `README`, le téléphone démarre, initialise un *keystore*, et nous demande de redémarrer.
Au prochain démarrage, l'exposant privé d'une clé RSA nous est demandé. Il faut analyser la consommation électrique du téléphone au démarrage, et en extraire la clé. L'affichage du *graph* de consommation électrique permet de visualiser à quel moment de la séquence de *boot* la clé RSA est utilisée.

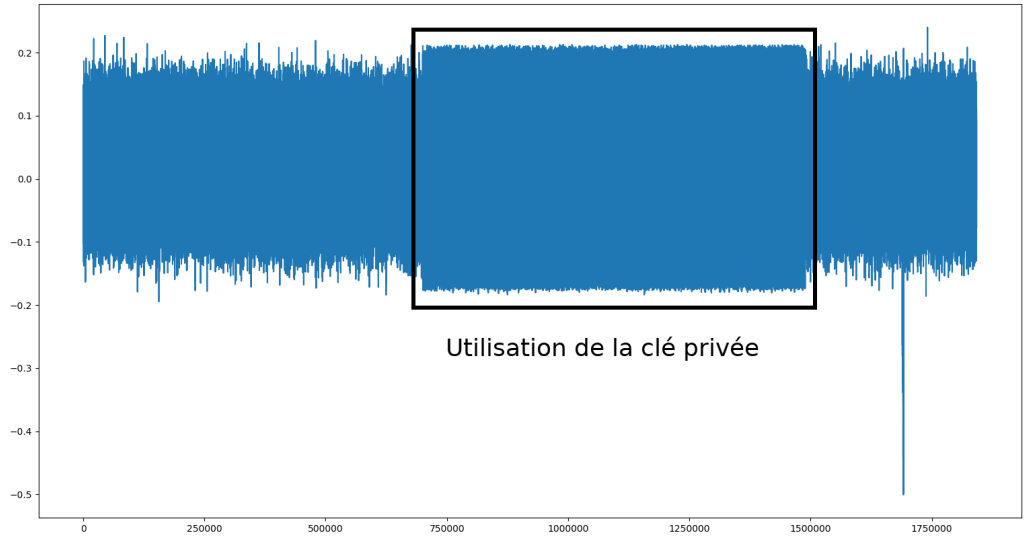

Figure 1: Graphique de consommation du téléphone au démarrage

En zoomant sur la portion intéressante, on distingue la répétition de deux schémas distincts.

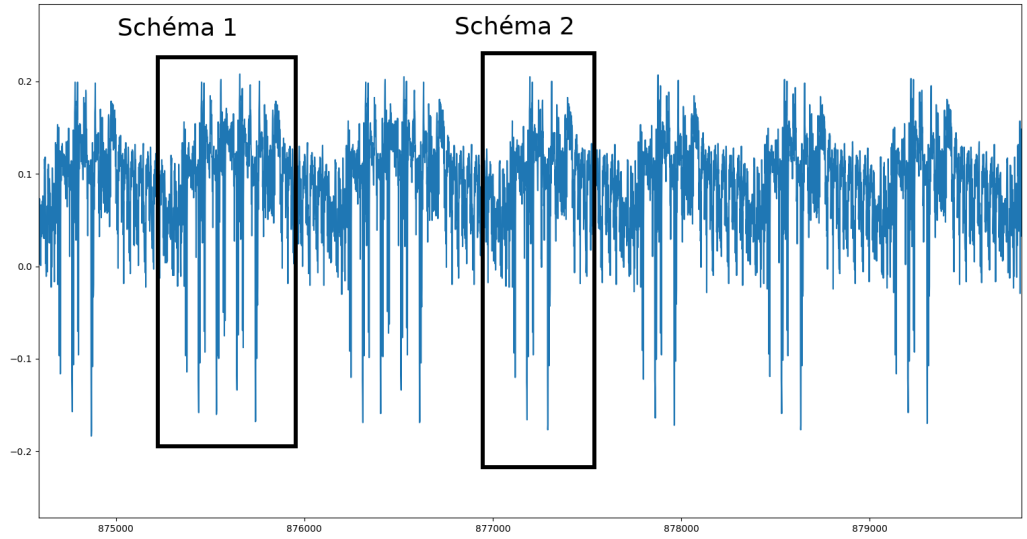

Figure 2: Graphique de consommation du téléphone au démarrage

Ces deux schémas proviennent des deux opérations de l'algorithme `Square and Multiply` permettant d'elever un nombre `N` à la puissance `D`. Plus précisemment, l'algorithme boucle sur chaque bits de `D`, et effectue l'opération `Square` si le bit est à 0, et l'opération `Square and Multiply` si le bit est à 1.

```
def square_and_multiply(n, d):
    exp = bin(d)
    value = n

    for i in range(3, len(exp)):
        value = value * value
        if(exp[i:i+1]=='1'):
            value = value*n
    return value
```

En notant les ordres d'apparition de ces deux opérations, il est possible de retrouver les bits de `D` un à un, `D` étant l'exposant de la clé privée RSA recherchée. Le script `step_1/key.py` permet de déterminer l'exposant à partir du fichier au format `NumPy`.

```
io :: 2019/step_1 => python3 key.py
23d87cdf97bb95abe6273c384190c765f552ab86f6de30a8db74435c95e6e3138f54a
f689812d8f9359cf0f4d453a0c11ec68ce470216c09e74c8947adaf23e902415d61dd
f2c0ffe459cbb40f7de42bdb7cd14093100a570e8c29819765e2d8d276f86471b52ac
29aa2ce2bb72cd45006279e82bec253ae9675fe45824f6001
```

La clé obtenue permet au téléphone de déchiffrer le système d'exploitation, et de terminer sa séquence de *boot*. On obtient ainsi un shell avec les droits root sur le téléphone. Le flag intermediaire est affiché pendant la séquence de *boot*.

Flag de validation:
`SSTIC{a947d6980ccf7b87cb8d7c246}`

2.2 Fichiers de l'archive

L'archive contient les fichiers:

- `step_1/key.py` permettant de déterminer l'exposant de la clé privée RSA.

3 Étape 2 - *Secure element* logiciel

3.1 Découverte du système de fichiers

L'exploration du système de fichiers du téléphone permet d'obtenir de nombreuses informations:

- Le dossier `/root` contient 3 conteneurs chiffrés `safe_01`, `safe_02`, `safe_03`
- Le dossier `/root` contient une image `schematics.png`, et un script `get_safe1_key.py`, qui constituent à eux deux la deuxième étape du challenge.
- Le dossier `/root/tools` contient des scripts pythons permettant d'ajouter des clés au *keystore*, et de déchiffrer les conteneurs.
- Le dossier `/lib` contient un module kernel `sstic.ko`

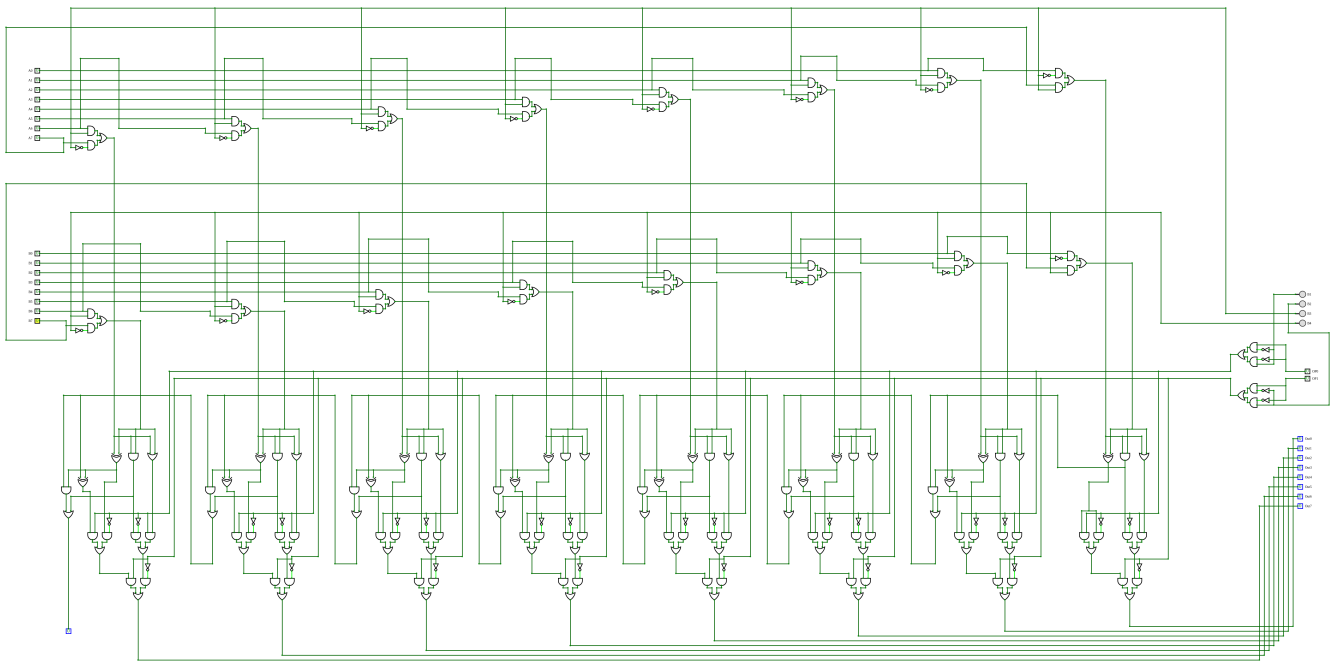

Figure 3: L'image `schematics.png` présente dans le dossier `/root`

Le fichier `get_safe1_key.py` est un script *python* décrivant la communication entre un utilisateur et un *Secure element* logiciel non-implémenté dans le script. Le but de l'étape est d'implémenter le *Secure element*, puis d'utiliser notre implementation afin de determiner la valeur d'une clé, qui nous permettra de déchiffrer le premier conteneur. Plusieurs choses sont à noter dans le script:

- Le fichier `schematics.png` présente les entrées, sorties et portes logiques du *Secure element* à implémenter.
- Le *Secure element* accepte en entrée:
  - 8 bits correspondants à l'octet `A`, représentés `A0,...,A7`
  - 8 bits correspondants à l'octet `B`, représentés `B0,...,B7`
  - 4 bits correspondants à 4 boutons. Un bit correspondant à un bouton vaut 1 lorsque le bouton est *appuyé*, 0 s'il est *relaché*.
  - 2 bits correspondants à l'entrée `OP`, représentés `OP0, OP1`
- Le *Secure element* génère en sortie:
  - 8 bits correspondants à l'octet `Out`, représentés `Out0,...,Out7`
- La fonction `main` du script commence par vérifier l'implémentation logicielle du `Secure element` en calculant les valeurs de sorties lorsque les 4 boutons sont successivement dans l'état *appuyé*, puis dans l'état *relaché* via la fonction `init`. Pour chacun des deux états des boutons, le *Secure element* est appelé plusieurs fois, son octet de sortie étant reporté dans l'octet d'entrée `B`.

```
# secure_device(A, B, op)

def init():
    r = secure_device(0x46,0x92,0)
    r = secure_device(0xdf,r,2)
    r = secure_device(0x3e,r,0)
    r = secure_device(0x3a,r,3)
    r = secure_device(0x36,r,2)
    r = secure_device(0x8e,r,2)
    r = secure_device(0xc9,r,3)
    r = secure_device(0xe7,r,1)
    r = secure_device(0x29,r,2)
    r = secure_device(0xc2,r,2)
    r = secure_device(0x79,r,0)
    r = secure_device(0x2a,r,2)
    r = secure_device(0x4c,r,3)
    r = secure_device(0xde,r,0)
    r = secure_device(0x88,r,0)
    r = secure_device(0x8b,r,2)
    r = secure_device(0x97,r,3)
    r = secure_device(0x6a,r,2)
    r = secure_device(0x60,r,1)
    r = secure_device(0x0f,r,0)
    r = secure_device(0x5b,r,3)
    r = secure_device(0xd0,r,2)
    r = secure_device(0xa9,r,1)
    r = secure_device(0xe3,r,3)
    r = secure_device(0xd0,r,1)
    r = secure_device(0x27,r,0)
    r = secure_device(0x90,r,0)
    r = secure_device(0x3b,r,1)
    r = secure_device(0x66,r,2)
    r = secure_device(0xe2,r,0)
```

```
    r = secure_device(0x24,r,3)
    r = secure_device(0xee,r,1)
    r = secure_device(0xf2,r,3)
    return r
```

- Le script calcule ensuite les 8 octets de la clé de déchiffrement du conteneur en utilisant 8 entrées de l'utilisateur, et le *Secure element*. Les entrées utilisateur correspondent, comme dans l'initialisation, aux 4 boutons. Chaque octet est calculé par une séquence unique d'appel au *Secure element*, correspondant aux fonctions `step1,...,step8`. Pour donner un ordre d'idée, voila l'implémentation de la fonction `step2`.

```
def step2():
    r= secure_device(0xde,0xab,0)
    r= secure_device(0x67,r,3)
    r= secure_device(0x2a,r,2)
    r= secure_device(0x6d,r,1)
    r= secure_device(0x4a,r,3)
    r= secure_device(0xe7,r,0)
    r= secure_device(0x1c,r,1)
    r= secure_device(0x35,r,0)
    r= secure_device(0xde,r,3)
    r= secure_device(0xf7,r,0)
    r= secure_device(0xda,r,2)
    return r
```

- Une fois les 8 octets calculés, le hash *SHA-256* de la clé est comparé à la valeur hardcodée `00c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea`.

- Si le hash correspond à la valeur hardcodée, une clé *AES* est dérivée de la clé, et permet de déchiffrer le conteneur.

3.2 Implémentation du `Secure element`

L'implémentation du *Secure element* est relativement simple si l'on décompose les rassemblements de portes logiques en blocs. On peut observer 4 blocs distincts sur le schema. Une fois ces 4 blocs implémentés, il suffit de chaîner leurs entrées et sorties et de vérifier notre implémentation avec les valeurs fournies par le script pour les boutons dans les états *appuyés* et *relachés*.

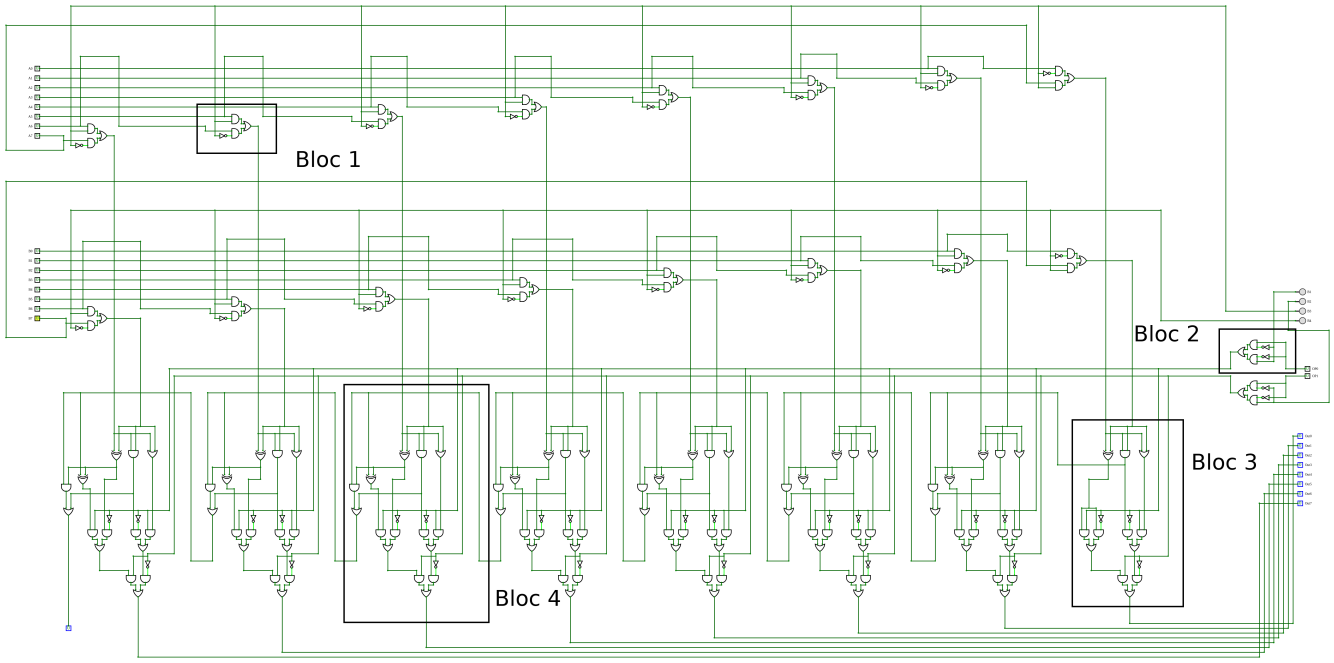

Figure 4: Les 4 blocs de portes logiques

```
def bloc_1(self, x1, x2, x3, x4):
    return ((x3 & x4) | (x2 & (0xff - x1)))

def bloc_3(self, x1, x2, x3, x4):
    i1 = x1 ^ x2
    i2 = x1 & x2
    i3 = x1 | x2
    r1 = self.bloc_1(x3, i1, x3, i1)
    r2 = self.bloc_1(x3, i2, x3, i3)
    y2 = self.bloc_1(x4, r2, x4, r1)
    return i2, y2

def bloc_4(self, x1, x2, x3, x4, x5, x6):
    i1 = x3 ^ x4
    i2 = x3 & x4
    i3 = x3 | x4
    y1 = (i1 & x1) | i2
    r1 = self.bloc_1(x5, i1, x5, (i1 ^ x2))
    r2 = self.bloc_1(x5, i2, x5, i3)
    y2 = self.bloc_1(x6, r2, x6, r1)
    return y1, y2

def bloc_2(self, x1, x2, x3, x4):
    return(((0xff - x1) & x2) | ((0xff - x3) & x4))
```

3.3 Bruteforce de la clé

Après avoir implémenté le *Secure element* correctement, l'objectif est de déterminer les 8 octets de la clé dont le condensat *SHA-256* correspond à la valeur hardcodée dans le script.

Il faut ainsi déterminer les positions des 4 boutons pour les 8 étapes (correspondants aux 8 octets de la clé). Cela correspond à trouver les `8 x 4 = 32` bits de l'entrée utilisateur. 2 solutions sont envisagées:

- Bruteforcer les 32 bits en entrée, calculer la valeur de sortie du *Secure element* pour toutes les positions des boutons, et calculer leur condensat *SHA-256*.
- Determiner les sorties possibles pour chaque étape (chacune ne pouvant générer au maximum que $2^4 = 16$ octets différents), puis calculer et comparer les condensats *SHA-256* des 16^8 possibilités.

Dans les deux cas, le bruteforce s'effectue sur 32 bits. La deuxième solution est cependant bien plus rapide, car elle ne nécessite que le calcul du condensat, contrairement à la première, qui nécessite le calcul des opérations du *Secure element*, puis le calcul du condensat. Ainsi, on commence par calculer toutes les valeurs de sortie, pour les 8 étapes:

```
io :: 2019/step_2 => python3 gen_output.py
steps:      01  02  03  04  05  06  07  08
in: 00      d7  29  ed  aa  fd  31  dd  83
in: 01      19  d1  8d  38  00  b2  54  54
in: 02      df  08  d9  a9  af  f6  4d  78
in: 03      40  db  df  28  47  64  4f  fb
in: 04      af  52  db  55  ff  4e  bb  07
in: 05      62  a4  1c  71  01  65  a9  aa
in: 06      bf  10  b0  53  60  ed  9a  f0
in: 07      81  b5  bf  52  8e  c8  9e  f3
in: 08      47  32  bd  1a  ed  3d  60  97
in: 09      39  d9  4f  4e  41  cf  86  85
in: 0a      c7  90  dc  ad  de  ee  05  d8
in: 0b      c2  da  dd  00  d4  e0  df  fc
in: 0c      90  f3  7b  35  cb  1b  ca  4d
in: 0d      72  b4  9f  9c  82  9d  e8  48
in: 0e      8f  20  01  5b  bd  dd  0a  b0
in: 0f      89  f5  bb  40  a9  c1  bf  fd
```

puis on bruteforce pour chaque possibilité:

```
io :: 2019/step_2 => time ./bf
8f, a4, df, a9, d4, ed, bb, f0
Bouttons:
1 1 1 0
0 1 0 1
0 0 1 1
0 0 1 0
1 0 1 1
0 1 1 0
0 1 0 0
0 1 1 0
./bf  706,81s user 0,00s system 99% cpu 11:46,86 total
io :: 2019/step_1 => echo -n "8fa4dfa9d4edbbf0" | xxd -r -p | sha256sum
00c8bb35d44dcbb2712a11799d8e1316045d64404f337f4ff653c27607f436ea  -
```

La solution est trouvée sur un `Intel CORE i7, 8th gen` en moins de 12 minutes. On entre la clé dans le script et on le relance. La clé AES dérivée permet de déchiffrer le conteneur `safe_01` via le script `/root/tools/add_key.py`

Flag de validation:
SSTIC{5fb3a83d1fd97137076019ad6e96c6a366fb6b32618d162e00cdee9bad427a8a}

3.4 Fichiers de l'archive

- `step_2/gen_output.py` implémente le *Secure element* logiciel, et génère l'ensemble des valeurs de sortie pour chaque `step`.
- `step_2/schematics.png` est l'image au format PNG représentant les entrées, sorties, et portes logiques du *Secure element*.
- `step_2/bf.c` implémente le bruteforce de la clé, en fonction des valeurs de sortie des différentes `steps`.

4 Étape 3 - Une *machine virtuelle* dans le nettoyeur de pile

4.1 Découverte du binaire

Le conteneur `safe_01` ne contient qu'un seul fichier `decrypted_file`. C'est un fichier au format *ELF*[4], strippé[5].

```
io :: 2019/step_3 => file decrypted_file
decrypted_file: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynami
cally linked, interpreter /lib/ld-, for GNU/Linux 3.7.0, BuildID[sha1]=5b5be1337
d13c986d0e21441d771a36e41a34d17, stripped
```

Son exécution sur le téléphone laisse penser que nous allons devoir résoudre un `crackme`[6].

```
# ./decrypted_file
Usage : ./decrypted_file <flag>
# ./decrypted_file toto
Not good
```

La commande `strings` affiche des chaînes de caractères intéressantes, notamment:

- `SSTIC{congolexicomatisation}`
- `That's the correct flag :)`

Bien entendu, `SSTIC{congolexicomatisation}` ne permet pas de résoudre l'étape. Pour avancer, il est nécessaire de désassembler le binaire. Plusieurs choses notables apparaissent lorsqu'on ouvre le binaire avec un désassembleur:

- La fonction `main` du programme vérifie que le binaire a bien été appelé avec un argument `argv[1]`. Si c'est le cas, la fonction à l'adresse `0x402e34` est appelée, avec le même argument passé en paramètre.

[4] C'est le format classique des exécutables Linux.
[5] Cela signifie que les symboles de l'exécutable ne sont plus présents dans le binaire.
[6] Challenge de *reverse engineering* où l'objectif est de déterminer le mot de passe affichant le message de validation.

- La fonction à l'adresse `0x402e34` initialise une exception de taille 8 octets via la fonction`__cxa_allocate_exception` de la librairie `libc++abi`, copie le pointeur sur `argv[1]` dans l'exception nouvellement allouée, puis `throw` l'exception via la fonction `__cxa_throw` de la même librairie.

- L'exception est gérée par la fonction à l'adresse `0x402f20`. Le programme affiche simplement la chaîne de caractères pointée par le registre `x28`.

- Le binaire contient une grosse portion de données non identifiées. La portion commence à l'adresse `0x400258`. L'entropie de cette portion est trop faible pour que les données soit chiffrées, et les octets ne sont pas désassemblables en instructions *AARCH64*.

4.2 Gestion des exceptions et *machine virtuelle*

Afin de comprendre les mécanismes de ce binaire, il convient de s'interesser au fonctionnement des exceptions C++. Lorsqu'une exception est levée, le programme vérifie qu'il existe une section de code permettant de *gérer* cette exception. Si c'est le cas, le flot d'exécution est transféré vers cette section de code. Avant de rediriger le flot d'exécution, le programme doit *restaurer* la pile et les registres dans un état lui permettant de *gérer* l'exception correspondante. Toutes ces informations sont contenues dans 2 sections spécifiques au format *ELF*:

- `.eh_frame` contient plusieurs structures de données (`Common Information Entry`, `Frame Description Entry`, ...) nécessaires à la restauration de la pile et des registres.

- `.gcc_except_table` contient une structure de données (`Language-Specific Data Array` ou `LSDA`) indiquant l'adresse de la redirection du flot d'exécution en fonction de l'adresse ayant généré l'exception.

Plus précisemment, les tables `Frame Description Entry` contiennent dans leur structure un champ `Call Frames Instructions`, contenant du *bytecode*, qui sera interprété par une machine virtuelle présente dans la librairie `libgcc` pour *restaurer* la pile et les registres. Une table `Frame Description Entry` est associée à une portion de code.

Cela étant dit, revenons à notre binaire, et tentons de comprendre comment cette exception est gérée. La section `.gcc_except_table` contient une table `LSDA` indiquant que, pour une exception levée entre les adresses `0x402eb4` et `0x402eb8`, le flot d'exécution sera redirigé à l'adresse `0x402f20`. Cela correspond bien à ce que nous avons vu à l'exécution du programme. L'instruction à l'adresse `0x402eb4` est un appel à la fonction `0x402e34` générant l'exception, et l'adresse `0x402f20` contient le code affichant la valeur du registre `x28` (dans notre cas `Not good`) à l'écran.

Analysons maintenant la section `.eh_frame` du binaire. Celle-ci contient une table `Frame Description Entry` correspondant à la fonction à l'adresse `0x402e34`. Dans cette table, le bytecode compris dans le champ `Call Frames Instructions` est assez court, et peut etre analysé via la commande `readelf`:

```
io :: 2019/step_3 => readelf --debug-dump=frames decrypted_file
Contents of the .eh_frame section:
[...]
00000090 000000000000001c 00000094 FDE cie=00000000 pc=0000000000402
e34..0000000000402e68
```

```
  DW_CFA_advance_loc: 1 to 0000000000402e35
  DW_CFA_def_cfa_offset: 32
  DW_CFA_offset: r29 (x29) at cfa-32
  DW_CFA_offset: r30 (x30) at cfa-24
  DW_CFA_val_expression: r28 (x28) (DW_OP_skip: -12222)
  DW_CFA_nop
  DW_CFA_nop
[...]
```

Le *bytecode* `DW_CFA_val_expression: r28 (x28) (DW_OP_skip: -12222)` est particulièrement intéressant. L'instruction `DW_OP_skip` correspond à une modification du pointeur de programme de la *machine virtuelle* interpretant le *bytecode*. Plus simplement, l'exécution du bytecode continue à l'adresse `$-12222 = 0x400258`. à l'adresse `0x400258` se trouve la grosse portion de données non-identifiée. On peut maintenant affirmer que cette portion correspond à du *bytecode*, interprété par la fonction de nettoyage de la pile et des registres, associée à notre exception.

Il est temps d'analyser le *bytecode*. La *machine virtuelle* qui l'exécute est en fait une fonction de la librairie `/lib64/libgcc_s.so.1`, présente à l'adresse `0x10b8b4`. Le fonctionnement de cette *machine virtuelle* est assez particulier, puisqu'elle ne contient aucun registre! Toutes les informations sont stockées sur la *pile* de la *machine virtuelle*.

La *machine virtuelle* a la possibilité:

- D'effectuer des opérations arithmétiques ainsi que des opérations bit à bit: *additions, ou-exclusif, décalages, etc...*
- D'écrire et de lire à des adresses du programme, via des *déréférencements*.
- D'accéder aux registres du programme lorsque l'exception à été générée.
- D'effectuer des opérations de modifications de la pile (*push, pop*)

Je ne détaillerai pas la signification de chaque *instruction* par soucis de clarté (il en existe plus de 50), et également parce que le code de la *machine virtuelle* est disponible sur internet.[7]

4.3 Désassemblage, assemblage, analyse

Pour comprendre les opérations effectuées par le *bytecode*, j'ai écrit un petit désassembleur traduisant le *bytecode* en instructions au format `X86_64`. À la vue du code généré (plusieurs milliers d'instructions `X86_64`), j'ai décidé d'*assembler* le code `X86_64` afin d'*émuler* le *bytecode* original. L'outil `keystone-engine` m'a permis d'*assembler* les instructions `X86_64` générées, et `miasm` m'a permis de les *émuler*.

J'ai rencontré un principal problème dans ma transformation du *bytecode* en `X86_64`: la gestion des instructions de modifications du pointeur de programme. En effet, les instructions du programme recompilées ne font pas la même taille que les instructions du *bytecode*. Cela entrainait un décalage dans les sauts[8], et cassait le flot d'exécution du programme généré. Ma solution a été de réécrire tous les sauts à la main (pardonnez moi).

[7] Code source

[8] En assembleur X86_64, les sauts sont fait à une adresse *relative* par rapport à l'adresse du pointeur de programme.

Il a été assez aisé de vérifier que le comportement de mon programme assemblé correspondait bien à celui de la *machine virtuelle* grâce à l'émulation. Il me suffisait de vérifier que la pile de mon programme était la même que la pile de la *machine* à un instant t de l'exécution.

Après avoir assemblé le programme, on peut commencer à l'analyser:

- Le programme commence par récupérer l'adresse de notre argument, le pousse sur sa pile, puis pousse les 32 premiers caractères de notre argument comme 4 mots de 64 bits. Le programme pousse ensuite le 33eme caractère sur la pile, et vérifie qu'il est égal au byte nul. Si ce n'est pas le cas, le programme sort en renvoyant la valeur `0x4030b8`, qui est l'adresse de la chaîne de caractères `Not good`. C'est la première vérification de l'algorithme: notre entrée utilisateur doit contenir au maximum 32 caractères.

- Le programme entame ensuite une longue transformation de nos 4 mots de 64 bits.

- Lorsque la transformation est terminée, le programme compare les 4 mots de 64 bits transformés à 4 mots de 64 bits inscrits en dur dans le programme.

- Si les 4 mots transformés correspondent, le programme se termine et retourne la valeur `0x403098`, qui est l'adresse de la chaîne de caractère `That's the correct flag :)`.

Il faut donc analyser l'algorithme de transformation:

- La transformation peut être vue comme un ensemble de 7 fonctions indépendantes, répetées 8 fois de suite.

- Les 7 différentes fonctions prennent en paramètres des entiers non-signés de 32 ou 64 bits, effectuent des opérations inversibles (addition, ou-exclusif, décalage de bits), non-inversibles (ou-logique, et-logique), et utilisent des tableaux de valeurs.

- Une des fonctions attire particulièrement notre attention de par sa taille: elle effectue plusieurs milliers d'instructions `X86_64`. Heureusement pour nous, cette fonction ne prend qu'un seul entier de 32 bits en paramètre, et cet entier ne peut prendre que 2 valeurs distinctes lors de l'exécution du programme. La valeur de retour de cette fonction ne peut donc prendre que 2 valeurs distinctes. Il nous reste 6 fonctions à analyser.

Ci-dessous se trouve un apercu de l'implémentation des différentes fonctions.

```
def func1(a, b):
    e = a & 0xffffffff
    f = (a >> 32) & 0xffffffff
    g = b & 0xffffffff
    h = (b >> 32) & 0xffffffff
    for i in range(4):
        t1 = (f + g) & 0xffffffff
        t2 = e ^ t1
        t3 = f & g
        t4 = (g - t2 + 0x100000000) & 0xffffffff
        b = (h & 0xff) << 2
        s = int.from_bytes(table_func1[b:b+4], byteorder='little')
        t5 = (t2 + s) & 0xffffffff
```

```
        t6 = t5 ^ (h >> 8)
        res1 = (t6 << 32) | t4
        res2 = (t3 << 32) | t5
        h = res1 >> 32
        g = res1 & 0xffffffff
        f = res2 >> 32
        e = res2 & 0xffffffff
    return res2, res1
```

```
def func2(a, b, c):
    t1 = (4 * c)
    t2 = int.from_bytes(table_func2[t1:t1+4], byteorder='little')
    t3 = t2 ^ (a & 0xffffffff)
    t4 = (0x45786532 + ((a >> 32) & 0xffffffff) & 0xffffffff)
    t5 = t4 ^ (b & 0xffffffff)
    t8 = ROR32(b, 0x1c)
    t9 = (t3 - t4 + 0x100000000) & 0xffffffff
    t10 = (t5 & 0x80000000) == 0
    return t5, t8, t4, t9, t10
```

```
def func3(a, b, c, d, e):
    return ((e ^ c ^ b) << 32) | d, (b << 32) | a
```

```
def func4(a):
    b = (a >> 32) & 0xffffffff
    a = a & 00xffffffff
    t2 =    [0x489dddde, 0x95bf74a9, 0xe6d80e3,
             0xfb92cd42, 0xf2b3a3fb, 0xe74f99e0]
    t6 =    [0x67990f1, 0x77941ee7, 0x2dedaf8b,
             0xd0e867c0, 0x6c39ce47, 0x5a24f221]
    for i in range(6):
        t3 = (b + t2[i]) & 0xffffffff
        t4 = t3 ^ a
        t7 = (t4 | t6[i])
        a = t4
        b = t7 ^ b
    return a, b
```

```
def func5(a, b, c, d):
    t1 = d & 0xffffffff
    t2 = (d >> 32) & 0xffffffff
    t3 = c ^ t2
    t4 = t1 ^ b
    t7 = ROR32(t4, 0x1c)
    t8 = t7 ^ c
    t11 = ROR32(t3, 0x12)
    t12 = b ^ t11
    t13 = t12 << 32
    t14 = (t13 | t8) ^ a
    return t14
```

```
def func6(a, b, c, d):
    t1 = d & 0xffffffff
    t2 = (d >> 32) &0xffffffff
    t3 = c ^ t1
```

```
    t7 = ROR32(t3, 0x6)
    t4 = t2 ^ b
    t4 ^= c
    t8 = t7 ^ c
    t11 = ROR32(t4, 0xe))
    t13 = t11 << 32
    t14 = (t13 | t7) ^ a
    return t14
```

```
def algo(a, b, c, d):
    for i in range(8):
        a1, b1 = func1(c, d)
        for k in range(15):
            a3, b3 = a1, b1
            l = 0
            while l <= k:
                b2, c2, d2, e2, f2 = func2(a3, b3, l)
                w = 0x60bf080f
                if f2:
                    w = 0x818f694a
                a3, b3 = func3(b2, c2, d2, e2, w)
                l += 1
            a4, b4 = func4(b & 0xffffffff, (b >> 32) & 0xffffffff)
            a = func5(a3, a4, b4, a)
            a4, b4 = func4(a & 0xffffffff, (a >> 32) & 0xffffffff)
            b = func6(b3, a4, b4, b)
        a, b, c, d = c, d, a, b
    return a, b, c, d
```

4.4 Inversion de l'algorithme

La résolution du challenge consiste à trouver les valeurs `a`, `b`, `c`, `d`, pour lesquelles la fonction `algo(a, b, c, d)` retourne les valeurs `0xdc7564f1612e5347`, `0x658302a68e8e1c24`, `0x65850b36e76aaed5` et `0xd9c69b74a86ec613`. Etant donné que l'algorithme boucle 8 fois sur les mêmes opérations, il nous suffit d'inverser un seul tour pour inverser entierement la fonction `algo`. Un tour travaille également sur 4 mots de 64 bits. et possède une particularité intéressante: il ne modifie que 2 des 4 mots sur lesquels il opère.

Pour faire simple, on peut résumer un tour de l'algorithme à la fonction suivante:

```
def tour(a, b, c, d):
    t1, t2 = operations(a, b, c, d)
    return c, d, t1, t2
```

Connaissant les valeurs de `c, d, t1 et t2`, on peut déterminer les valeurs de retour des fonctions `func1, func2, et func3` pour chaque tour. Ainsi, l'inversion de l'algorithme consiste à déterminer, 15 fois pour chaque tour, les valeurs de `a` et `b` dans le système d'équations suivant:

```
a4, b4 = func4(b)
new_a = func5(a3, a4, b4, a)
a4, b4 = func4(a)
new_b = func6(b3, a4, b4, b)
```

où seuls les mots `new_a`, `new_b`, `a3` et `b3` nous sont connus.

Pour résoudre le système, j'ai utilisé le résolveur de contraintes `z3-solver`. J'ai pu inverser le tour grâce à l'algorithme suivant:

```
def inverse_round(a, b, c, d):
    a1, b1 = func1(c, d)
    a3, b3 = a1, b1
    k = 14
    while k >= 0:
        l = 0
        while l <= k:
            b2, c2, d2, e2, f2 = func2(a3, b3, l)
            w = 0x60bf080f
            if f2:
                w = 0x818f694a
            a3, b3 = func3(b2, c2, d2, e2, w)
            l += 1

        a_orig = z3.BitVec('a_orig', 64)
        b_orig = z3.BitVec('b_orig', 64)
        z3a = z3.BitVec('z3a', 64)
        z3b = z3.BitVec('z3b', 64)

        a4, b4 = func4(b_orig)
        z3a = func5(a3, a4, b4, a_orig)
        a4, b4 = func4(z3a)
        z3b = func6(b3, a4, b4, b_orig)

        s = z3.Solver()
        s.add(a == z3a, b == z3b)
        s.check()
        a = s.model()[a_orig].as_long()
        b = s.model()[b_orig].as_long()
        a3, b3 = a1, b1
        k -= 1
    return c, d, a, b
```

A partir d'un tour, il est aisé d'inverser toute la fonction `algo`:

```
a = 0xdc7564f1612e5347
b = 0x658302a68e8e1c24
c = 0x65850b36e76aaed5
d = 0xd9c69b74a86ec613
for i in range(8):
    a, b, c, d = inverse_round(a, b, c, d)
s1 = c.to_bytes(8, byteorder='little').decode('utf-8')
s2 = d.to_bytes(8, byteorder='little').decode('utf-8')
s3 = a.to_bytes(8, byteorder='little').decode('utf-8')
s4 = b.to_bytes(8, byteorder='little').decode('utf-8')
print("{:s}{:s}{:s}{:s}".format(s1, s2, s3, s4))
```

```
io :: 2019/step_3 => ./solve.py
SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}
```

La clé permet de déchiffrer le conteneur `safe_02`.

Flag de validation:
`SSTIC{Dw4rf_VM_1s_co0l_isn_t_It}`

4.5 Fichiers de l'archive

- `step_3/decrypted_file` est le binaire *ELF* original.
- `step_3/disas.py` convertit le *bytecode* de la *machine virtuelle* en instructions *X86_64*
- `step_3/assemble.py` assemble le *bytecode* de la machine virtuelle en *machine code X86_64* au format *hexstring*
- `step_3/vm.bin` est le fichier contenant le *machine code X86_64* de la machine virtuelle avec les sauts modifiés à la main.
- `step_3/emul.py` permet d'émuler le fichier `step_3/vm.bin` via *miasm*.
- `step_3/solve.py` implémente l'algorithme de la *machine virtuelle* ainsi que la méthode d'inversion de ce dernier.

5 Étape 4 - Une *machine virtuelle* chiffrée dans le *Secure state*

5.1 Découverte du binaire

Lors du déchiffrement du conteneur `safe_02`, un message indique:
`[w] You must reboot in order to decrypt Secure OS`
Au redémarrage du téléphone, la séquence de *boot* présente de nouveaux messages:

```
NOTICE:  BL31: Initializing BL32
NOTICE:  Booting Secure-OS
```

Nous y reviendrons plus tard. Il convient d'abord de s'intéresser au fichier présent dans le conteneur déchiffré.

Comme le conteneur `safe_01`, `safe_02` ne contient qu'un binaire au format ELF nommé `decrypted_file`.

```
# cd safe_02
# ls -l decrypted_file
-rwxr-xr-x    1 root     root       1532432 Apr 15 11:03 decrypted_file
# file decrypted_file
decrypted_file: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux), s
tatically linked, for GNU/Linux 3.7.0, stripped
```

Le binaire est plutot volumineux, et contient une grosse portion de données à l'entropie élevée. La commande `strings` retourne énormément de chaîne de caractères provenant de librairies. Cela vient du fait que le binaire est statiquement *linké* au librairies. À l'exécution, le binaire nous demande de passer en paramètre une clé de 32 octets au format *hexstring*.

```
# ./decrypted_file
usage: ./decrypted_file [32-bytes-key-hex-encoded]
# ./decrypted_file 0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1
f20
Loose
```

Il est temps d'ouvrir le binaire dans un désassembleur. Je me suis surtout intéressé à la fonction `main`, qui se trouve à l'adresse `0x400778`. Plusieurs choses sont à noter:

- La fonction commence par vérifier que notre chaîne de caractère passée en paramètre `argv[1]` contient bien 64 caractères. Si la condition est vérifiée, le programme tente de convertir la chaîne du format *hexstring* en *rawbytes*.

- Si la conversion n'échoue pas, le programme tente d'ouvrir le driver linux `/dev/sstic`.

- Le programme envoie ensuite 2 *ioctl*[9] (ayant comme valeur de requête respectivement `0xc0105300` et `0xc0501301`) au driver nouvellement ouvert. Pour rappel, un ioctl est envoyé avec 3 paramètres: `ioctl(int fd, unsigned long request, char *argp)`. `fd` correspond au descripteur de fichier correspondant au driver, `request` est un entier non signé correspondant à la *valeur* de l'`ioctl`. Enfin, le dernier paramètre correspond à un pointeur sur la donnée que l'on souhaite envoyer.

- Le premier ioctl est envoyé avec en troisieme paramètre un pointeur sur une adresse située dans la section `.rodata`. À cette adresse se trouve le début de la portion à forte entropie.

- Le deuxième ioctl est envoyé avec en paramètre un pointeur sur l'adresse contenant notre argument au format *rawbytes*.

- Enfin, le programme boucle sur deux *ioctls* ayant comme valeur de requête `0xc0105302` et `0xc0105303`. Pour les deux requêtes, le 3eme paramètre est nul. Le programme sort de la boucle lorsque la valeur de retour de l'*ioctl* correspondant à la requête `0xc0105303` vaut `0xffff`.

La vérification de notre clé ne se fait donc pas dans le programme `decrypted_file`. Il faut analyser le module kernel `sstic.ko` afin de comprendre les différentes actions des *ioctls*. Le driver est disponible dans le répertoire `/lib/` du téléphone.

Le désassemblage du module permet d'identifier la liste des requêtes *ioctls* disponibles. Cependant, le module ne nous permet pas d'en apprendre beaucoup plus sur la vérification de la clé. En effet, celui-ci agit plus ou moins comme un *passe-plat*, en appellant pour chaque *ioctl* le *Secure monitor* du processeur, via la fonction `__arm_smccc_smc`.

5.2 Niveau d'exceptions et déchiffrement du code de l'EL3

Il est nécéssaire de bien comprendre le fonctionnement des différents *niveaux d'exceptions* des processeurs de la famille *ARMv8* avant de continuer.

Les processeurs de la famille *ARMv8* possèdent 4 *niveaux d'exceptions* (appelés *EL0*, *EL1*, *EL2* et *EL3*), et 2 *états* de sécurité (le *Non-secure state*, et le *Secure state*).

Les deux différents états du processeurs possèdent leur propre *espace d'adressage*. Lorsque le processeur est dans l'état *Secure state*, il peut accèder aux *deux* espaces d'adressage, contrairement à l'état *Non-secure state*, dans lequel le processeur n'accède qu'à l'espace d'adressage *Non-secure*.

Concernant les *niveaux d'exceptions*:

- En règle générale, les applications lancées sur le système d'exploitation sont exécutées par le processeur avec les privilèges les plus faibles: *EL0* dans l'état *Non-secure state*.

[9] Un *ioctl* est un appel système spécifique à un driver donné. Cela permet à un utilisateur de faire exécuter une action au driver cible.

- Le noyau du système d'exploitation est exécuté avec les privilèges *EL1* dans l'état *Non-secure state*.
- Le mode hyperviseur du processeur est exécuté avec les privilèges *EL2* dans l'état *Non-secure state*.
- Le *Secure monitor* du processeur est exécuté avec les privilèges *EL3*, dans l'état *Secure state*.

Une image vaut mieux qu'un long discours:

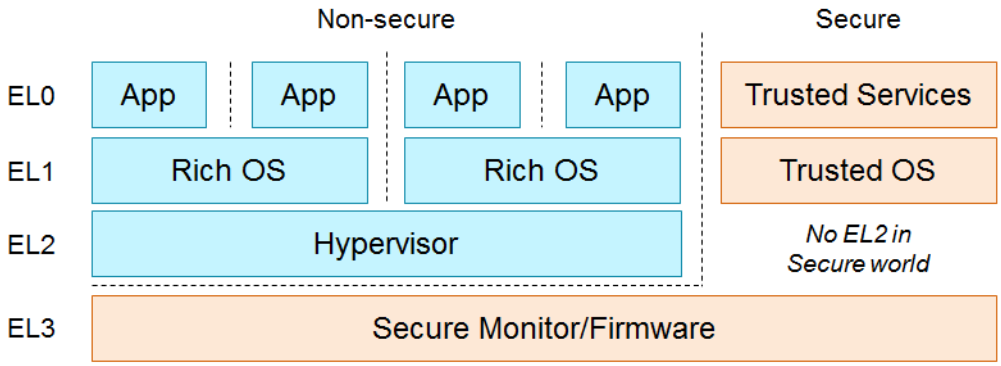

Figure 5: Les différents *niveaux d'exceptions* des processeurs *ARMv8*

Les transitions entre les différents *niveaux d'exceptions* s'effectuent grâce à plusieurs instructions:

- L'instruction `svc` appelée *Supervisor Call*, permet au processeur de passer du `niveau EL0` au *niveau EL1*.
- L'instruction `hvc` appelée *Hypervisor Call*, permet au processeur de passer du `niveau EL1` au *niveau EL2*.
- L'instruction `smc` appelée *Secure Monitor Call*, permet au processeur de passer du `niveau EL1` au *niveau EL3*.
- L'instruction `eret` génère une *exception*, permettant au processeur de passer d'un *niveau d'exception* avec des privilèges élevés à un *niveau d'exception* avec des privilèges plus faibles ou équivalents.

Chaque *niveau d'exception* possède une *table d'exception*, pointée par le registre `vbar_eln`, où `n` correspond au niveau d'exception. Cette table contient les instructions exécutées lors d'un changement de *niveau d'exception*.

L'image suivante (issue de la documentation *ARM*) détaille la structure d'une table.

Table 10.2. Vector table offsets from vector table base address

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
+ 0x080	IRQ/vIRQ	
+ 0x100	FIQ/vFIQ	
+ 0x180	SError/vSError	
+ 0x200	Synchronous	Current EL with SPx
+ 0x280	IRQ/vIRQ	
+ 0x300	FIQ/vFIQ	
+ 0x380	SError/vSError	
+ 0x400	Synchronous	Lower EL using AArch64
+ 0x480	IRQ/vIRQ	
+ 0x500	FIQ/vFIQ	
+ 0x580	SError/vSError	
+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	
+ 0x700	FIQ/vFIQ	
+ 0x780	SError/vSError	

Figure 6: Structure d'une *table d'exception*

Dans le cas de notre module kernel `/dev/sstic`, la fonction `__arm_smccc_smc` est simplement un *wrapper* vers l'instruction `smc`. On peut ainsi supposer que le code de vérification de notre clé se trouve dans le *Secure state* au niveau d'exception *EL3* du processeur. Ainsi, le code exécuté lors des *Secure Monitor Call* du module kernel se trouve à l'adresse `vbar_el3 + 0x400` (puisque l'exception génerée provient du *niveau d'exception EL1* en *AARCH64*).

Il faut donc récupérer le code s'exécutant dans le *Secure state*. Pour cela, on peut désassembler le fichier `rom.bin` et déterminer à quel moment le *bootloader* déchiffre le *Secure-OS*. On peut également lancer la commande de démarrage du téléphone en ajoutant les paramètres `-s -S`[10], puis débugger le *bootloader* avec `gdb-multiarch`. La fonction de déchiffrement est assez facile à trouver en utilisant le *débugger* et en suivant l'apparition des messages, la séquence de *boot* étant plutot verbeuse. La fonction de déchiffrement se trouve à l'adresse `0x2c64`. Nul besoin de déterminer l'algorithme (qui semble être un AES en mode *stream*[11]), il suffit de placer deux breakpoints aux adresses `0x2c64` et `0x2ca4`. Le premier breakpoint permet de récupérer l'adresse à laquelle sera déchiffrée la portion binaire dans le registre `x2`, ainsi que sa taille dans le registre `x3`. Le deuxième breakpoint

[10] L'option `-s` permet d'attacher un debugger a *qemu*. L'option `-S` freeze le CPU au démarrage.

[11] Un chiffrement en mode *stream* est reconaissable à l'opération *ou-exclusif* effectué entre le *stream* généré, et la portion à chiffrer.

permet ainsi de *dumper* la portion binaire déchiffrée via la commande *gdb*: `dump memory dumped.bin addr_start addr_end`.

On obtient alors un blob binaire de taille `0x9440`, chargé à l'adresse `0xe00b000`. Ce nouveau binaire semble implémenter la même fonction de déchiffrement à l'adresse `0xe00d07c`. En réitérant notre methode, on parvient à *dumper* 3 nouvelles portions binaires:

- Une portion située à l'adresse `0xe030000` de taille `0x90b0`
- Une portion située à l'adresse `0xe200000` de taille `0x5380`
- Une portion située à l'adresse `0x60000000` de taille `0x2220030`

En utilisant la commande `strings`, on détermine rapidement que la dernière portion *dumpée* correspond au noyau du téléphone. Nous ne nous y intéresserons pas.

Il nous reste 2 portions à analyser, relativement petites. On peut commencer à désassembler les deux portions, que l'on *mappe* dans *ghidra* à leur adresses respectives. Les premières instructions de la portion à l'adresse `0xe030000` définissent le registre `vbar_el3` à l'adresse `0xe036800`. Cela signifie que la portion de code exécutée lors d'un *Secure Monitor Call* depuis le kernel se trouve à l'adresse `0xe036c00`. On peut le vérifier rapidement en placant un *breakpoint* à cette adresse, et en lancant le binaire. Au breakpoint, `x0` vaut `0x83010004`, qui correspond à la valeur passée par le module kernel lors du tout premier *ioctl* du programme. De la même manière, les premières instructions de la portion à l'adresse `0xe200000` définissent le registre `vbar_el1` à la valeur `0xe203000`. On peut supposer que la portion à l'adresse `0xe200000` correspond au code de l'*EL1* exécuté en *Secure state*, et la portion à l'adresse `0xe030000` au code de l'*EL3*. On a maintenant tous les élements nous permettant de résoudre le challenge.

5.3 Des registres et du bytecode chiffrés

S'en suit alors une longue phase de debugging et de *reverse engineering* afin de comprendre les différentes interactions entre les *niveaux d'exceptions* et le fonctionnement du code du *Secure state*.

Voici un petit résumé des différentes découvertes:

- Le code de l'*EL3* implémente une *machine virtuelle* aux registres et aux *opcodes* chiffrés. Toutes les opérations efféctuées par la *machine virtuelle* se trouvent dans la fonction à l'adresse `0xe031034`.
- Les 16 registres sont des blocs de 16 octets chiffrés situés sur la pile de la fonction à l'adresse `0xe031034`.
- Le premier *ioctl* envoyé par le programme permet à l'*EL3* de copier la portion à forte entropie dans l'espace d'adressage de l'*EL3*. Cette portion correspond aux *opcodes* chiffrés de la *machine virtuelle*, ainsi qu'a un tableau utilisé par la *machine virtuelle* pour effectuer ses opérations.
- Le deuxième *ioctl* (`0xc0105301`) copie 4 par 4, les 16 octets de l'entrée utilisateur (`argv[1][12:28]`) dans les registres `fpexc32_el2`, `dacr32_el2`, `ifsr32_el2` et `sder32_el2`. Ces 16 octets vont servir tout au long de l'exécution à chiffrer et déchiffrer les 16 registres de la *machine virtuelle*. La chiffrement est en fait 1 tour d'*AES*, implémenté pour chaque opération de la *VM* via les instructions `aes.e`, `aes.d` et `eor`.

- Les deux autres *ioctls* (respectivement `0xc0105302` et `0xc0105331`) répétés en boucle permettent respectivement de déchiffrer la prochaîne instruction de la *machine virtuelle*, et d'effectuer une ou plusieurs opérations sur ses registres via la fonction à l'adresse `0xe2005a4`.

- La fonction de déchiffrement des *opcodes* se trouve à l'adresse `0xe200e84`. Elle utilise l'adresse de l'opcode à déchiffrer, et 2 mots de 16 octets situés aux adresses `0xe204048` et `0xe204058` pour générer une clé (via l'instruction `sm4ekey`). L'opcode est ensuite déchiffré par 4 tours de l'algorithme SM4, via l'instruction `sm4e` et la clé nouvellement générée.

- La fonction de déchiffrement des *opcodes* est appelée via une exception génerée par l'*EL1* via une tentative d'accès en lecture ou en écriture dans la portion d'opcodes chiffrés.

- Le code de l'*EL1* implémente de l'*anti-debugging*. En particulier, la fonction à l'adresse `0xe201650` vérifie que *qemu* n'a pas été lancé avec l'argument `-s`[12]. Si c'est le cas, la portion correspondant au tableau utilisé par l'algorithme de vérification est mappé à une adresse décalée de `0x1000`.

Afin de comprendre les différentes opérations de la *machine virtuelle*, j'ai écrit un script *gdb* me permettant de récupérer l'état des registres à chaque *opcode* exécuté par la *machine virtuelle*, ainsi que l'*opcode* exécuté.

```
target remote localhost:1234

# Patch anti-debug
b * 0xe2017b4
c
set $x0 = 0
del 1

b * 0xe2005a4
b * 0xe031034
c
c
c
c
set $cpt = 0
while($cpt<10000)
    # Dump opcode
    if $pc == 0xe2005a4
        print $x0
        set $cpt = $cpt+1

    # Dump registers and operation
    else
        if $pc == 0xe031034
            x/32xg 0xe04a5d8
            printf ">> 0x%x 0x%x 0x%x 0x%x", $x0, $x1, $x2, $x3
        else
        end
```

[12] L'option `-s` permet d'attacher un debugger à *qemu*.

```
    end
    c
end
```

Le script génère un fichier de 27M. Voici quelques lignes du fichier géneré:

```
Breakpoint 2, 0x000000000e031034 in ?? ()
0xe04a5d8:      0x8624839042273791      0xd708b2e7ca0c665a
0xe04a5e8:      0x8624839042273791      0xd75bb2e7ca0ca95a
0xe04a5f8:      0x8624839042273791      0xd7e3b2e7ca0ccb5a
0xe04a608:      0x8624839042273791      0xd7ceb2e7ca0c5e5a
0xe04a618:      0x8624839042273791      0xd7a9b2e7ca0cc15a
0xe04a628:      0x8624839042273791      0xd7a9b2e7ca0ca95a
0xe04a638:      0x8624839042273791      0xd7a9b2e7ca0cb95a
0xe04a648:      0x8624839042273791      0xd7a9b2e7ca0c665a
0xe04a658:      0x8624839042273791      0xd713b2e7ca0c615a
0xe04a668:      0x8624839042273791      0xd708b2e7ca0c665a
0xe04a678:      0x86248390b0273791      0xd7b4b2e7ca0c665a
0xe04a688:      0x8624839042273791      0xd74ab2e7ca0c165a
0xe04a698:      0x8624839042273791      0xd7a9b2e7ca0ce65a
0xe04a6a8:      0x862483901e273791      0xd7a9b2e7ca0c545a
0xe04a6b8:      0x8624839042273791      0xd7a9b2e7ca0c875a
0xe04a6c8:      0x8624839042273791      0xd7a9b2e7ca0c205a
>> 0x83010001 0xf 0x0 0x0
Breakpoint 1, 0x000000000e2005a4 in ?? ()
$104 = 0x29c00

Breakpoint 2, 0x000000000e031034 in ?? ()
0xe04a5d8:      0x8624839042273791      0xd708b2e7ca0c665a
0xe04a5e8:      0x8624839042273791      0xd75bb2e7ca0ca95a
0xe04a5f8:      0x8624839042273791      0xd7e3b2e7ca0ccb5a
0xe04a608:      0x8624839042273791      0xd7ceb2e7ca0c5e5a
0xe04a618:      0x8624839042273791      0xd7a9b2e7ca0cc15a
0xe04a628:      0x8624839042273791      0xd7a9b2e7ca0ca95a
0xe04a638:      0x8624839042273791      0xd7a9b2e7ca0cb95a
0xe04a648:      0x8624839042273791      0xd7a9b2e7ca0c665a
0xe04a658:      0x8624839042273791      0xd713b2e7ca0c615a
0xe04a668:      0x8624839042273791      0xd708b2e7ca0c665a
0xe04a678:      0x86248390b0273791      0xd7b4b2e7ca0c665a
0xe04a688:      0x8624839042273791      0xd74ab2e7ca0c165a
0xe04a698:      0x8624839042273791      0xd7a9b2e7ca0ce65a
0xe04a6a8:      0x862483901e273791      0xd7a9b2e7ca0c545a
0xe04a6b8:      0x8624839042273791      0xd7a9b2e7ca0c875a
0xe04a6c8:      0x8624839042273791      0xd7a9b2e7ca0c205a
```

Les registres sont chiffrés. On peut cependant les déchiffrer via 1 tour d'AES, en utilisant la clé passée en paramètre du programme. On obtient le nouveau fichier de log, avec les registres déchiffrés:

```
Breakpoint 2, 0x000000000e031034 in ?? ()
0       c03
1       925
2       506
3       708
4       9
5       25
```

```
6       c
7       3
8       102
9       c03
a       4c503
b       2500
c       4
d       100020
e       1e
f       6f
>> 0x83010001 0xf 0x0 0x0
Breakpoint 1, 0x000000000e2005a4 in ?? ()
$104 = 0x29c00

Breakpoint 2, 0x000000000e031034 in ?? ()
0       c03
1       925
2       506
3       708
4       9
5       25
6       c
7       3
8       102
9       c03
a       4c503
b       2500
c       4
d       100020
e       1e
f       6f
```

A coté de cela, un script permettant de déchiffrer les opcodes est implémenté. On peut maintenant écrire un désassembleur, qui va nous permettre de comprendre le coeur de l'algorithme implémenté par la *machine virtuelle*.

Un opcode à toujours une taille de 3 octets. Deux tableaux permettent de décrire rapidement les differents types d'*opcodes* de la *machine virtuelle*:

bits	23..20	19..18	17..14	13..0
	type	instruction	reg1	imm1

bits	23..20	19..18	17..14	13..10	9..0
	type	instruction	reg1	reg2	non-utilisé

Il existe 4 *types* d'*opcodes*:

- Les sauts.
- les opérations entre un registre et une valeur immédiate.
- les opérations entre deux registres.
- des *opcodes* spécifiques à la *machine virtuelle* (ex: écriture d'une valeur hardcodée à une adresse)

5.4 Algorithme et inversion

La *machine virtuelle* peut effectuer des opérations arithmétiques, et des opérations bit à bit. Elle peut également écrire et lire à des adresses mémoire.

De plus, la *machine virtuelle* semble implémenter une deuxième technique d'*anti-debugging* soit par *contrôle d'intégrité*[13], soit par *contrôle temporel*[14]. L'auteur n'ayant pas su déterminer l'origine et la nature exacte du contrôle. Les opcodes `ec414a` et `ecc14a` semblent effectuer un saut conditionnel, en fonction du résultat du contrôle. Il suffit de ne pas prendre en compte les sauts conditionnels dans notre implémentation python pour obtenir l'algorithme qui serai executé sans la présence d'un *debugger*.

L'algorithme est effectué 4 fois sur notre entrée utilisateur divisé en 4 mots de 64 bits `a, b, c, d`. Une fois les opérations effectuées sur les 4 mots, les octets des 4 mots résultants sont comparés aux octets de la valeur inscrite en dure `pr.a.rfg.cnf.fv.snpvyr@ffgvp.bet`[15] (divisée en 4 mots de 64 bits). Il convient donc de trouver les 4 mots de 64 bits générant la chaîne de caractères désirée en sortie de l'algorithme.

L'algorithme boucle 32 fois pour chaque mot et, comme pour l'étape 3, l'inversion d'un seul tour permet d'inverser entièrement l'algorithme. Plus précisément, le mot de 64 bits en entrée est divisé en 4 mots de 16 bits, et les opérations sont effectuées sur ces 4 mots séparément. L'algorithme est difficilement inversible à cause de l'utilisation des valeurs en entrée comme index d'un tableau de données. Cependant, un tour de l'algorithme ne modifie que 2 des 4 mots de 16 bits, et 2 des 4 mots en entrée ne sont pas utilisés pour la transformation! Voici une représentation très simplifiée d'un tour de l'algorithme:

```
# x0, x1, x2, x3, y2 et y3 sont des mots de 16 bits
def tour(x0, x1, x2, x3):
    y2 = operations_non_inversibles(x1)
    y3 = operations_inversible(x0, x1)
    return x2, x3, y2, y3
```

Pour un tour, il est possible de déterminer `x1` en bruteforcant toutes les valeurs possibles sur 16 bits jusqu'a obtenir `y2`. Une fois la bonne valeur de `x1` determinée, il est aisé de calculer `x0`, les opérations étant inversibles. On réitère l'opération 32 fois pour déterminer 1 des 4 mots de 64 bits en entrée. Et on recommence, pour les 3 autres mots de 64 bits.

Le résultat est obtenu en un temps malheureusement très peu raisonable (14 minutes...) sur la machine de l'auteur.

```
io :: 2019/step_4 => time ./solve.py
[...]
acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e
./solve.py    892,74s user 0,02s system 99% cpu 14:52,77 total
```

La clé obtenue permet de valider l'étape, et de déverrouiller le dernier conteneur.

Flag de validation:
`SSTIC{acadaa8b5b55306fb3c6dfc3b2d1c80770084644225febd71a9189aa26ec740e}`

[13] Les contrôles d'intégrité consistent à vérifier qu'aucun breakpoint n'est présent dans le code.

[14] Les contrôles temporels consistent à calculer le temps d'execution d'une portion de code. L'execution dans un *debugger* est généralement bien plus lente à cause de l'execution *pas-à-pas*.

[15] C'est le ROT13 de la chaîne de caractère `ce.n.est.pas.si.facile@sstic.org`

5.5 Fichiers de l'archive

- `step_4/decrypted_id3_0xe030000` est la portion déchiffrée correspondant au code de l'*El3* dans le *Secure state*.
- `step_4/decrypted_id4_0xe200000` est la portion déchiffrée correspondant au code de l'*El1* dans le *Secure state*.
- `step_4/gdb.sc` est le script *gdb* permettant la génération du log d'exécution de la *machine virtuelle* avec les registres chiffrés.
- `step_4/decrypt_registers.py` déchiffre les registres du fichier de log d'exécution de la *machine virtuelle*.
- `step_4/sm4.py` implémente le chiffrement et déchiffrement de l'algorithme *SM4* sur 4 tours.
- `step_4/aes.py` implémente le chiffrement et déchiffrement de l'algorithme *AES* sur 1 tour.
- `step_4/array.bin` contient les opcodes et le tableau chiffrés de la *machine virtuelle*.
- `step_4/disas.py` permet le désassemblage du code chiffré de la *machine virtuelle*.
- `step_4/algo.py` implémente l'algorithme de transformation de la *machine virtuelle*
- `step_4/solve.py` implémente l'algorithme de transformation inverse par bruteforce de la *machine virtuelle*.

6 Étape 5 - Des sms malveillants

Le dernier conteneur déchiffré ne contient qu'un seul fichier `decrypted_file`.

```
# file decrypted_file
decrypted_file: gzip compressed data, last modified: Wed Mar 27 10:28:25 2019, f
rom Unix, original size 116
```

On télécharge le fichier sur notre machine, puis on le décompresse. On découvre finalement un répertoire `data/`, correspondant au dossier d'un système de fichier d'un téléphone android contenant les données des différentes applications installées.

```
io :: 2019/step_5 => ls -l data/ | head
total 524
drwxr-xr-x  4 stan stan 4096 mars  27 11:24 android
drwxr-xr-x  4 stan stan 4096 mars  27 11:24 com.android.apps.tag
drwxr-xr-x  4 stan stan 4096 mars  27 11:24 com.android.backupconfirm
drwxr-xr-x  5 stan stan 4096 mars  27 11:24 com.android.bluetooth
drwxr-xr-x  4 stan stan 4096 mars  27 11:24 com.android.bluetoothmidiservice
drwxr-xr-x  4 stan stan 4096 mars  27 11:24 com.android.bookmarkprovider
drwxr-xr-x  4 stan stan 4096 mars  27 11:24 com.android.calllogbackup
drwxr-xr-x  6 stan stan 4096 mars  27 11:24 com.android.camera2
drwxr-xr-x  4 stan stan 4096 mars  27 11:24 com.android.captiveportallogin
```

On cherche la chaîne de caractère `sstic` dans les fichiers:

```
io :: step_5/data => grep -iR sstic .
Binary file ./com.google.android.apps.messaging/databases/bugle_db matches
Binary file ./com.google.android.apps.messaging/databases/bugle_db-wal matches
Binary file ./com.android.providers.telephony/databases/mmssms.db matches
Binary file ./com.android.providers.contacts/databases/contacts2.db matches
```

```
io :: step_5/data => strings ./com.google.android.apps.messaging/databases/bugle
_db | grep sstic
 l'adresse 9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org vers votre bo
 l'adresse 9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org vers votre bo
?tablestickersstickers
tablesticker_setssticker_sets
```

On obtient finalement l'adresse tant attendue, le challenge est terminé.

Adresse de validation:
`9e915a63d3c4d57eb3da968570d69e95@challenge.sstic.org`

En lisant les SMS présents sur le téléphone, on découvre qu'une attaque d'ampleur est en train d'être organisée. Les attaquants semblent vouloir réitérer l'opération "Ça coule de source" ayant touché la moitié du public lors de l'édition du SSTIC 2018. Le dernier SMS reçu nous permet de conclure notre investigation: il faudra se méfier du stand des huîtres lors du *social event* de l'édition 2019.

7 Conclusion

Ne m'étant pas essayé aux challenges SSTIC des années précédentes, j'ai été surpris par l'excellente qualité du challenge de cette année, qui a du nécessiter un gros travail de la part des concepteurs. Les epreuves m'ont permis d'apprendre tout un tas de choses intéressantes, notamment sur les processeurs de la famille *ARMv8* et les exceptions *C++*, et surtout de me familiariser avec d'excellents outils comme *miasm* ou *keystone-engine*. Bref, un grand MERCI aux concepteurs!