Solution du challenge SSTIC 2017

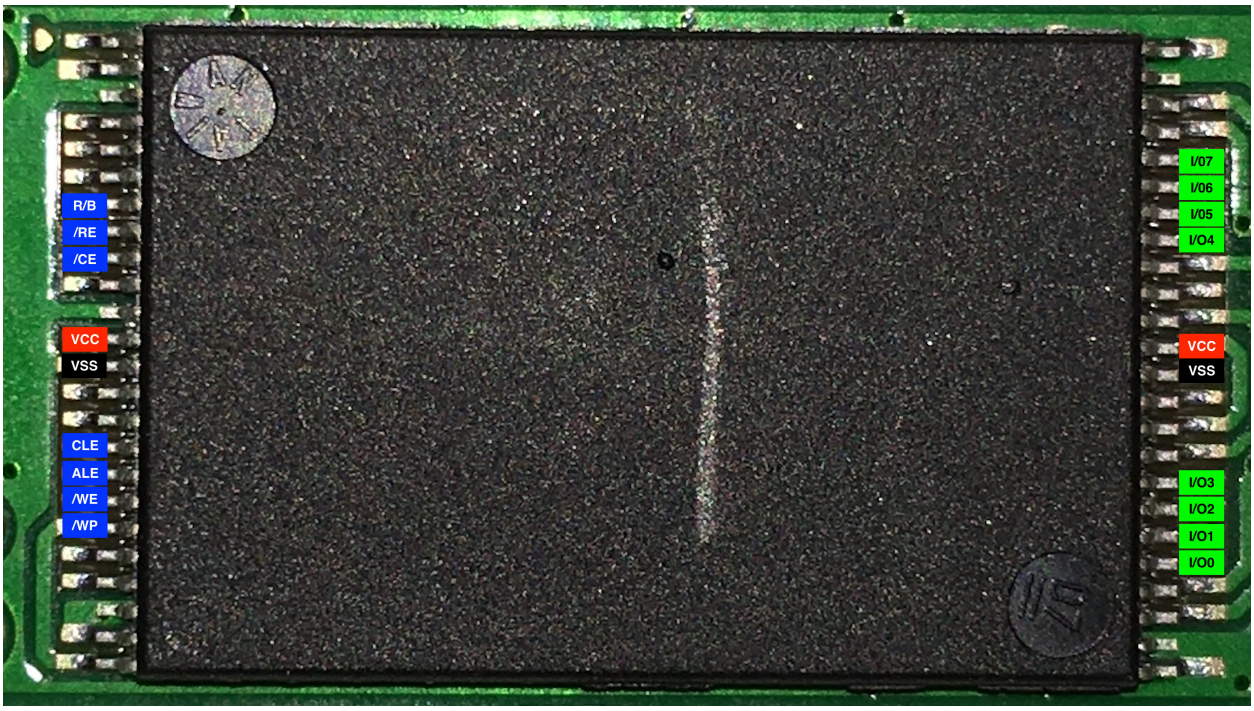
Stratox

Electronic Flash

L'archive ZIP du challenge contient pour unique fichier un courrier électronique (.eml). Ce dernier peut être ouvert avec un client de messagerie, nous permettant de découvrir son contenu :

```
Bonjour,

La chaîne de production est prête pour produire votre nouvelle console
de jeux.

Nous avons programmé un premier modèle avec le firmware que vous nous
avez fourni.
Avant de lancer la production, merci de valider que le firmware écrit
sur ce premier modèle est bien celui que vous souhaitez
déployer sur l'ensemble de vos consoles.
Pour cela, et afin de ne pas faire d'erreur, nous avons enregistré avec
un analyseur logique la programmation de la mémoire flash NAND, vous
trouverez en pièce jointe cet enregistrement. Nous utilisons le logiciel
sigrok, vous pouvez récupérer le firmware avec le décodeur parallèle sur
le bus de données. Nous utilisons uniquement l'interface graphique
"pulseview", mais il me semble qu'il y aussi des bindings python et une
interface CLI, qui vous permettront certainement d'automatiser l'extraction.

Dans l'attente de votre validation.

Cordialement,
John
```

Le message possède deux pièces jointes. La première, `NAND_pinout.jpg`, est une photo montrant la Flash NAND et ses lignes d'entrées/sorties (*pins*).



La seconde pièce jointe, `NAND_FLASH_writes_no_OOB_5MHz.sr`, contient l'enregistrement de l'analyseur logique. Un tel enregistrement permet de connaître l'état des différentes lignes d'entrées/sorties au cours du temps. Il est au format par défaut de Sigrock, qui est un fichier ZIP contenant :

— `version` : fichier texte contenant la version du format de fichier ;
— `metadata` : fichier texte contenant diverses métadonnées, notamment la liste des *pins* dont l'état a été enregistré ;
— `logic-*` : fichiers binaires contenant l'enregistrement.

```
$ cat metadata
[global]
sigrok version=0.4.0

[device 1]
capturefile=logic-1
total probes=16
samplerate=5 MHz
probe1=I/O-0
probe2=I/O-1
probe3=I/O-2
probe4=I/O-3
probe5=I/O-4
probe6=I/O-5
probe7=I/O-6
probe8=I/O-7
probe9=CLE
probe10=ALE
probe11=WE
probe12=RE
unitsize=2
```

Pour extraire de cet enregistrement le *firmware* qui a été écrit, il nous faut connaître le protocole de communication avec la Flash NAND. Le protocole est visiblement standard et est, par exemple, décrit dans ces spécifications : https://www.micron.com/~/media/documents/products/data-sheet/nand-flash/20-series/2gb_nand_m29b.pdf.

Aperçu du protocole de communication avec une Flash NAND

Voici un résumé de l'utilité des *pins* :

- **WE** : *Read enable*, le transfert est dans le sens NAND vers système hôte ;
- **RE** : *Write enable*, le transfert est dans le sens système hôte vers NAND ;
- **I/O-0** à **I/O-7** : entrées/sorties permettant de faire passer des données. Comme il y a 8 pins I/O, chacune représentant un bit, les données sont en général lues octet par octet. Un octet peut être lu (ou écrit) lorsque le pin RE (WE) passe de la position basse à haute (front montant) ;
- **CLE** : *Command Latch Enable*, lorsque ce *pin* est en position haute, des informations de commande sont envoyées vers la Flash par les *pins* I/O-0 à I/O-7 à chaque fois que le *pin* WE est sur front montant ;
- **ALE** : *Address Latch Enable*, lorsque ce *pin* est en position haute, des informations d'adresse sont transférées vers la Flash par les *pins* I/O-0 à I/O-7 à chaque fois que le *pin* WE passe de la position basse à haute (front montant).

Lorsque CLE et ALE sont en position basse et que WE est en front montant, des données sont transférées vers la NAND par les *pins* I/O-*.

Lorsque CLE et ALE sont en position basse et que RE est en front montant, des données sont transférées vers le système hôte par les *pins* I/O-*.

WE	RE	CLE	ALE	I/O-*	Notes
0→1	0	1	0	informations de commandes	Le système hôte envoie une commande
0→1	0	0	1	informations d'adresse	Le système hôte indique une adresse. L'adresse est envoyée en plusieurs cycles (1 cycle permet d'envoyer un octet), le nombre de cycles dépend de la commande
0→1	0	0	0	données	Le système hôte envoie des données vers la NAND
0	0→1	0	0	données	La NAND envoie des données vers le système hôte

Voici les commandes utiles pour la résolution :

- 0xFF : RESET ;
- 0x90 : READ ID. Demande à la NAND des identifiants inscrits lors de sa fabrication ;
- 0x60 : BLOCK ERASE SETUP. Efface un bloc de mémoire dont l'adresse est fournie en 3 cycles ;
- 0xD0 : BLOCK ERASE CONFIRM. Confirme l'effacement d'un bloc ;
- 0x80 : SERIAL DATA INPUT. Demande l'écriture de données. L'adresse est fournie en 5 cycles suivie des données ;
- 0x10 : PROGRAM. Confirme l'écriture de données ;

Décodage de l'enregistrement

Une méthode consiste à utiliser le framework Sigrock : https://sigrok.org/wiki/Main_Page. Sigrock ne possède pas par défaut de décodeur adapté. Il est possible d'en créer un sous forme de plugin python : https://sigrok.org/wiki/Protocol_decoder_HOWTO.

Un LUM (clef optionnelle pour la résolution du challenge) se trouve dans le résultat de la commande READ ID :

```
{
    "cmd": "READ ID"
    "ReadData": b'SSTIC PSEUDO NAND   LUM{x.g215WiPCR}\x00\x00'
}
```

Nous observons ensuite l'écriture de données, le *firmware*, vers la NAND bloc par bloc. Les blocs étant programmés dans le désordre, il est important de tenir compte de leur adresse pour extraire et reconstituer le *firmware*.

Le *firmware* est sous forme d'un système de fichier FAT32, dans lequel se trouve deux fichiers :

```
$ mount firmware.bin ./fw

$ ls -l ./fw
total 14435
-rwxr-xr-x 1 root root 14780383 févr. 17 11:23 challenges.zip
-rwxr-xr-x 1 root root       49 févr. 17 11:23 challenges.zip.md5

$ cat ./fw/challenges.zip.md5
3977e2084331bb1c52abb115a332c6cc  challenges.zip

$ md5sum ./fw/challenges.zip
3977e2084331bb1c52abb115a332c6cc  ./fw/challenges.zip
```

Avant de passer à l'étape suivante du challenge, il reste un second LUM à trouver. Celui-ci se trouve dans un fichier effacé du système de fichier FAT32, qu'il est possible de récupérer avec des outils de la suite Sleuthkit.

```
$ fls -r firmware.bin
r/r 5:  challenges.zip
r/r 8:  challenges.zip.md5
r/r * 10:   lum.txt
v/v 1064195:    $MBR
v/v 1064196:    $FAT1
v/v 1064197:    $FAT2
d/d 1064198:    $OrphanFiles

$ icat firmware.bin 10
LUM{AsPBdVWz95y}
```

Contenu de challenges.zip

L'archive `challenges.zip` contient des fichiers d'une interface Web. Le fichier README.txt se trouvant à la racine détaille son utilisation :

```
Bonjour,

Voici la suite du challenge, les épreuves suivantes se dérouleront dans un
navigateur.  Pour fonctionner, ce dossier doit être propulsé par un serveur
web, par exemple avec python SimpleHTTPServer :

$ python -m SimpleHTTPServer 8000

Pour travailler plus confortablement tu peux te connecter à la machine
virtuelle fonctionnant dans ton navigateur en SSH.
Une passerelle Websocket<->Interface TAP est nécessaire pour fournir du réseau
à la machine virtuelle. Le projet "go-websockproxy" [1] permet de créer cette
passerelle, il remplit également la fonction de serveur web.

Le binaire nécessite des privilèges élevés (les capabilities CAP_NET_RAW et
CAP_NET_ADMIN ainsi que le droit d'exécuter la commande "ip" avec ces
capabilities) pour créer une interface TAP :
sudo go-websockproxy --listen-address=127.0.0.1:8090 --static-directory=$CHALLENGE_DIR
    --tap-ipv4=10.42.42.1/30

Ensuite il ne reste plus qu'à ouvrir ton navigateur à l'adresse
http://127.0.0.1:8090/main.html. Lorsque la machine virtuelle a fini de
démarrer, il est possible de se connecter en SSH :
ssh user@10.42.42.2, le mot de passe est "sstic".

[1] https://github.com/gdm85/go-websockproxy
```

Don't let him escape !

Le script `server.py` attache un filtre eBPF (*extended Berkeley Packet Filter*) à une *socket* réseau. Un filtre eBPF est en réalité un programme sous forme de *bytecode*, exécuté dans une machine virtuelle eBPF, à des fins de filtrage, ici sur des paquets réseau. Pour plus de détails, voir la section Références BPF.

Chaque fois qu'une trame ethernet est envoyée sur l'interface de *loopback*, le programme eBPF inspecte son contenu et indique au script `server.py` lorsque des conditions particulières sont vérifiées en lui envoyant un numéro d'état. Le passage d'information entre le programme eBPF et le `server.py` est effectué en utilisant une *map*, un concept propre au BPF. Les états d'intérêt à atteindre sont :

— 1 : un LUM se trouve dans la trame réseau ;
— 2 : la clé se trouve dans la trame réseau.

Cette partie consiste à désassembler le programme eBPF pour comprendre comment atteindre les états d'intérêt et ainsi trouver la clé ou le LUM.

Le LUM est une simple comparaison de chaîne de caractères.

```
$ /challenges/tools/add_lum LUM{BvWQEdCrMfA}
OK  !
```

La clé est constituée de quatre parties indépendantes de quatre octets qui doivent satisfaire des contraintes.

```
$ /challenges/tools/add_key 2d4ceda2fa2a0e08fc360b55291de7c9
OK  !
```

Références BPF

Voici des liens vers des ressources concernant BPF :

— The BPF system call API, version 14, https://lwn.net/Articles/612878/
— Attaching eBPF programs to sockets, https://lwn.net/Articles/625224/
— Résumé des principaux opcode : https://github.com/iovisor/bpf-docs/blob/master/eBPF.md
— Documentation plus complète : https://www.kernel.org/doc/Documentation/networking/filter.txt
— Désassembleur de bytecode eBPF : https://github.com/iovisor/ubpf/
— Outils de débogage : https://github.com/torvalds/linux/tree/master/tools/net
— Structures et prototypes de fonctions de la machine virtuelle : http://lxr.free-electrons.com/source/tools/include/uapi/linux/bpf.h

RISCy zones

```
/challenges/riscy_zones $ ls -l
total 52
-rw-r--r--    1 user     user          8036 Mar 24 22:03 TA.elf.signed
-rw-r--r--    1 user     user           400 Mar 24 22:03
    password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted
-rw-r--r--    1 user     user         35680 Mar 24 22:03 secret.lzma.encrypted
-rwxr-xr-x    1 user     user          8089 Mar 24 22:03 trustzone_decrypt
```

— **`trustzone_decrypt`** : exécutable qui prend en paramètre un fichier chiffré et sa clé pour le déchiffrer ;
— **`TA.elf.signed`** : exécutable utilisé par trustzone_decrypt pour effectuer le déchiffrement ;
— **`password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted`** : fichier chiffré dont la clé de chiffrement est connue ;
— **`secret.lzma.encrypted`** : fichier chiffré dont la clé est inconnue. Le but de cette étape est de trouver sa clé ;

trustzone_decrypt

`trustzone_decrypt` est un exécutable au format ELF 32-bit LSB, compilé pour l'architecture OpenRISC.

Aperçu de l'architecture OpenRISC

Référence : https://openrisc.io/architecture

Cette architecture possède 32 registres généraux de 32 bits, r0 à r31. Certains sont utilisés suivant une convention :

- r0 : valeur 0 ;
- r1 : SP, *stack pointer* ;
- r2 : FP, *frame pointer* ;
- r3-r8 : paramètres de fonction ;
- r9 : LR, *link address*, adresse de retour ;
- r11 : valeur de retour de fonction, partie basse ;
- r12 : valeur de retour de fonction, partie haute.

Analyse de trustzone_decrypt

```
/challenges/riscy_zones $ ./trustzone_decrypt
usage:
./trustzone_decrypt [password] [encrypted file] [destination file]
```

Nous pouvons l'exécuter pour déchiffrer le message dont la clé est connue :

```
$ ./trustzone_decrypt 00112233445566778899AABBCCDDEEFF
    password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted decrypted
[i] load TA.elf.signed in TrustedOS
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_GET_TA_VERSION
[+] OS return code = 0x00000000, TA return code = 0x00000000
retreived version : SSTIC Trusted APP v0.0.1
[i] check password in TEE
[+] OS return code = 0x00000000, TA return code = 0x00000000
Good password !
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000

<... nombreuses commandes CMD_DECRYPT_BLOCK ...>

[i] Unload TA form TrustedOS
[+] OS return code = 0x00000000, TA return code = 0x00000000
```

Nous pouvons également observer le comportement lorsque le mot de passe est mauvais :

```
$ ./trustzone_decrypt aaaa password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted
    decrypted
[i] load TA.elf.signed in TrustedOS
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_GET_TA_VERSION
[+] OS return code = 0x00000000, TA return code = 0x00000000
retreived version : SSTIC Trusted APP v0.0.1
[i] check password in TEE
[!] OS return code = 0x00000000, TA return code = 0xfffffffe
Bad password :(
[i] Unload TA form TrustedOS
[+] OS return code = 0x00000000, TA return code = 0x00000000
```

Il semble y avoir une méthode pour déterminer si le mot de passe est bon ou mauvais avant de commencer à déchiffrer le contenu du fichier original.

Pour continuer l'analyse, il est possible de le désassembler avec la version d'`objdump` présent dans la machine virtuelle du challenge :

```
objdump -x -d -s trustzone_decrypt > trustzone_decrypt.objdump.txt
```

Opérations effectuées par `trustzone_decrypt` :

1. charge l'application de confiance `TA.elf.signed` dans un environnement sécurisé, génériquement appelé TEE (*Trusted Execution Environment*). Une interface est disponible pour utiliser les services exposés par l'application de confiance ;
2. envoie à l'application de confiance le mot de passe ainsi que les 112 premiers octets du fichier chiffré ;
3. si l'application de confiance indique que le mot de passe est correct, alors le fichier est déchiffré en mode pseudo CBC où :
   (a) l'IV sont les 16 octets à l'offset 112 du fichier chiffré ;
   (b) les blocs chiffrés, de taille 16 octets, commence à l'offset 128 (112 + 16) du fichier chiffré ;
   (c) la valeur de XOR pour chaîner les bloc (l'IV ou un blocs Ci-1) est légèrement différente du CBC standard : l'ordre des octets est inversé ;
   (d) la primitive de déchiffrement par bloc est exécutée dans l'environnement sécurisé par l'application de confiance (commande CMD_DECRYPT_BLOCK).

TA.elf.signed

Exécutable au format ELF 32-bit MSB, compilé pour l'architecture RISC-V.

Aperçu de l'architecture OpenRISC

Spécifications : https://riscv.org/specifications/.

Il est possible de compiler une version d'`objdump` supportant cette architecture : https://github.com/riscv/riscv-binutils-gdb.git.

```
./configure --target=riscv32
make
```

Convention d'utilisation des registres RISC-V :

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller

Register	ABI Name	Description	Saver
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Analyse de TA.elf.signed

Pour son fonctionnement, `TA.elf.signed` utilise des appels système TEE :

- 1 : TEE_write
- 2 : TEE_readkey
- 3 : TEE_writekey
- 4 : TEE_AES_decrypt
- 5 : TEE_HMAC

Un appel système est effectué avec l'instruction `ecall`, par exemple :

```
000113c8 <TEE_write>:
   113c8:   00100893            li  a7,1
   113cc:   00000073            ecall
   113d0:   00008067            ret
```

`TA.elf.signed` expose 5 commandes :

- 0 : CMD_TA_INIT
- 1 : CMD_GET_TA_VERSION
- 2 : CMD_CHECK_PASSWORD
- 3 : CMD_DECRYPT_BLOCK
- 4 : CMD_GET_TA_LUM

Les sections suivantes résument, généralement sous forme de pseudo-code (mi-C, mi-python...), le fonctionnement des commandes.

CMD_TA_INIT

```
TEE_writekey(b'SSTIC_AES_KEY', b'___SSTIC_2017___', 16, 0)
TEE_writekey(b'SSTIC_PASSWORD_HMAC_KEY', b'7FDCB986058767EC47F417EFBE85A10C'.decode(hex)
    , 16, 1)
```

CMD_CHECK_PASSWORD

```
if((uint16)s0_input[4:6] > 16)
{
  *((uint16 *)s3_output) = -1;
}
if((uint16)s0_input[22:24] > 256)
{
  *((uint16 *)s3_output) = -1;
}

TEE_AES_decrypt((char *)b'SSTIC_AES_KEY', (char *)p_aes_decrypted, (char *)(&s0_input
    [24]), (uint16)s0_input[22:24])

if(memcmp(p_aes_decrypted, b'==BEGIN PASSWORD HMAC==\x0d\x0a', 25))
{
    TEE_write_wrapper('[DEBUG] Bad HMAC_BLOB')
    return -1;
}

if(memcmp(((char *)p_aes_decrypted + (uint16)s0_input[22:24] - 23), b'\x0d\x0a==END
    PASSWORD HMAC==', 23))
```

```
{
    TEE_write_wrapper('[DEBUG] Bad HMAC_BLOB')
    return -1;
}

if(decode_hmac(&p_aes_decrypted[25], p_decoded_hmac, 32) == 0)
{
    return -1;
}

TEE_HMAC(b'SSTIC_PASSWORD_HMAC_KEY', p_out, &s0_input[6], (uint16)s0_input[4:6])

if(memcmp(p_out, p_decoded_hmac, 32))
{
    TEE_write_wrapper("Bad password")
    return -2;
}

TEE_write_wrapper(b"Good password !")

TEE_writekey(b'SSTIC_CUSTOM_KEY', &s0_input[6], (uint16)s0_input[4:6], 0)

return 0;
```

CMD_DECRYPT_BLOCK

```
TEE_readkey(b'SSTIC_CUSTOM_KEY', p_key, 16);

// déchiffrement du bloc inliné
```

Le déchiffrement d'un bloc fonctionne globalement de la manière suivante :

```
Pi = F(key, counter) XOR Ci
```

où :

- F(key, counter) est une fonction qui prend en paramètre la clé de déchiffrement, un compteur et renvoie une valeur de 16 octets, que nous appellerons *key stream* ;
- Ci est le bloc chiffré d'indice i ;
- Pi est le bloc déchiffré d'indice i.

CMD_GET_TA_LUM

Cette commande effectue un déchiffrement simple, prenant en paramètre une clé d'un seul octet. En supposant que le clair est un LUM, nous pouvons effectuer une recherche exhaustive.

```
$ /challenges/tools/add_lum LUM{gdN8.D*@+UV}
OK  !
```

Noyau TEE

Nous avons vu que `TA.elf.signed` utilise des appels système TEE. Pour la compréhension globale de l'étape, il est utile de trouver où ils sont implémentés. Ils sont ici simulés en javascript :

```
$ grep -r 'TEE keystore' challenges/*
challenges/jor1k-worker-min.js:x=m("../lib/aes");=
```

Pour rendre le code javascript plus lisible, il est possible d'utiliser http://jsbeautifier.org/.

L'analyse du code permet de trouver que :

- TEE_HMAC effectue un HMAC standard

— TEE_writekey, lorsque son 4ème argument est différent de 0, effectue une opération de XOR sur la clé fournie avant de la charger. La valeur de XOR est :

```
[51, 137, 244, 253, 53, 246, 17, 181, 15, 206, 70, 166, 129, 206, 151, 113]
```

La clé ayant le nom b'SSTIC_PASSWORD_HMAC_KEY', chargée dans la commande CMD_TA_INIT, est donc modifiée avant son utilisation et vaut finalement `LUM{0qvYH:QI?K6}`.

```
$ /challenges/tools/add_lum LUM{0qvYH:QI?K6}
OK  !
```

Résolution

La formule de déchiffrement d'un bloc peut être représentée de la manière suivante :

```
Pi = F(key, counter) XOR Ci XOR reverse(Ci-1)
```

Ce qui implique :

```
F(key, counter) = Pi XOR Ci XOR reverse(Ci-1)
```

En l'ocurrence pour le premier bloc :

```
F(key, 0) = P1 XOR C1 XOR reverse(IV)
```

Nous connaissons la valeur de :

— reverse(IV)
— C1

Il s'avère que nous pouvons également faire des suppositions sur la valeur de P1. En effet, les 14 premiers octets d'un fichier LZMA sont constitués de 13 octets d'en-tête et du premier octet de données. Il sont relativement prévisibles, ce qui permet d'effectuer une attaque à clair connu sur 14 octets du premier bloc. Un bloc faisant 16 octets, il reste 2 octets inconnus.

Exemple de début de fichier compressé en LZMA :

```
$ hd test.lzma | head
00000000  5d 00 00 80 00 ff ff ff  ff ff ff ff ff 00 22 88  |]..............".|
00000010  45 ed 0c 07 e1 8f 7a da  f0 e7 64 40 04 ac e4 68  |E.....z...d@...h|
```

Pour résumer :

— key : valeur recherchée
— reverse(IV) : connu
— C1 : connu
— P1 : partiellement connu (14/16)

Nous pouvons donc connaître 14 octets de `F(key, 0)`. Il est possible d'utiliser un *solver* de contraintes tel que Z3 pour trouver les valeur candidates de `key`. Les deux octets restants peuvent faire l'objet d'une recherche exhaustive grâce au HMAC connu de `key`.

```
$ python z3-resolve.py
 Searching with keystream fbee4d4b7e3b4786162b9e2ceeddc16a
 Expected HMAC value 504861089d10a8d5900cdf3bad962004282e73c20d2d5ab95b67ae6b3356748b
 realHmacKey 'LUM{0qvYH:QI?K6}'

 [!] Key found :  5921cd9fd3a82bd9244ece5328c6c95f

~ $ /challenges/tools/add_key 5921cd9fd3a82bd9244ece5328c6c95f
OK  !
```

Le fichier déchiffré est une image dans laquelle se trouve un LUM :

```
$ strings secret | head
...
Congratulations : YHZ{+g%Yi.vzG8Z}
...

$ python -c "print 'YHZ{+g%Yi.vzG8Z}'.decode('rot13')"
LUM{+t%Lv.imT8M}

~ $ /challenges/tools/add_lum LUM{+t%Lv.imT8M}
OK  !
```

Unstable machines

Quand l'état 10 est atteint, un LUM est déchiffré à l'adresse 0x416614 :

```
$ /challenges/tools/add_lum LUM{2KREDvn3OPf}
OK  !
```

Les données de 0x414100 à 0x414240 déchiffrées par la clé XOR 0xF4B02F4B contiennent, entre autres, un LUM :

```
$ /challenges/tools/add_lum LUM{+zhVQqJy03q}
OK  !
```

Obscurcissement par machine virtuelle

Obscurcissement par machine virtuelle niveau 2

LUM à la fin du bytecode :

```
$ /challenges/tools/add_lum LUM{C1UAidv_pzJ}
OK  !
```

Résolution

```
~ $ /challenges/tools/add_key 3f691f3d6eb60b343c931c22e0baa92f
OK  !
```

LabyQRinth

La dernière étape se présente sous la forme d'un QR code représenté avec des caractères hexadécimaux.

```
$ cat final.txt
Une dernière petite étape!

        cefec06    b   a  e   0   cb519c4
        6     9 434 a4d 12   660e 4     4
        4 d94 f 9 bf f   02 b   a f 287 4
        f 01d 1 2 65   0   0   65 7 183 6
        a fd6 b   7 e7 5  a     5 d d4b c
        f     6   7  a  6   e   6 3     3
        c56b0b4 0 8 3 9 9 0 4 5 c d186a57
                60 3  8c      a 9
        2     6 214 9c b 20   d e80  65f
          422       f e95 b3   6 1 16e  8
        a    49eb 157  2  d  a  96d5  f
        d    9   2d   f4   5  a   d  6d2
        a   b55  cf36d6b657658a8abeca0 fc
          8  a  73   1  5   3 c 2c    7 5
```

```
           0 3 a4 0  2c58   86   1 f 2  56
           6 041 a   e   8  3f   3791 7   4
               6558bab a   e13   d 4 16 0
           f1 27   2 c  90 8829bb1 f8  431
           4b3 7e c    9 26  5e5 70e  c 9
          35 d61   e 9 3 a c2f c     f 7dc
          26   ad 0 4d b f4   a938ac9a7   3
          7   cc e  92    431 6 6   d   4c5
            8 f a  0da9  a0f  23  6 a3  8d
deQRypt(a  e6  8  8b 66   24   6 92 c   e
          ce80 5ba a c   d 2 5bd 81e52f c 3
                   c   8f  f  3d   6   d 2
          228caa6   e2 90   6 8b3ab 5 4c973);
          a     8   d  96f9   b   e   2b7e
          0 294 e  7   0    44 0  2061d2  9
          0 40c 9   fb 44 10  91bcc4  2 44
          3 829 5  1 c0   4   8   6e  f 08d
          e     b     07 2b a6b   0  a8c7
          fd0ca91 26ad 6     9  3    a   9
```

En partant du principe qu'un caractère hexadécimal représente un carré noir, un espace un carré blanc, nous pouvons le transformer en image et le décoder, ce qui donne la phrase suivante :

```
Please use this Nibble ADD key: 5571C2017
```

Un *nibble* est une séquence de 4 bits, soit un demi octet. Encodé en hexadécimal, un nibble est exactement un caractère hexadécimal.

La séquence hexadécimale à déchiffrer est trouvée en parcourant le QRcode à partir de `deQRypt(`, jusqu'à `);`.

```
def nibbleAdd(nibble, key) :
    return '%x' % ((int(nibble, 16) - int(key, 16)) % 0x10)

def deQRrypt(data, key = '5571C2017') :
    res = b''
    for i in range(len(data)) :
        res += nibbleAdd(data[i], key[i%len(key)])
    return res

main()

data = 'ace80e6fcca1776558babe2c51d6b657658a8abcf973d1bb5c938ac9da252fd4c973'
print deQRrypt(data, key = '5571C2017').decode('hex')
```

Résultat de deQRrypt :

```
WwLnWZkEAeMjPaoKEs5K7rld@sstic.org
```

Remerciements

Je tiens à féliciter les concepteurs de ce challenge qui respecte la tradition des challenges du SSTIC. Les problèmes sont bien construits, intéressants et demandent un investissement non négligeable afin de les résoudre.

Je tiens également à préciser que la partie décrivant la solution de l'étape *Unstable machines* n'est pas détaillée par manque de temps, mais son fonctionnement et tous ses détails méritent une longue explication.