SSTIC 2017

solution

http://communaute.sstic.org/ChallengeSSTIC2017

Voici la solution du challenge SSTIC 2017, par le candidat 'kerial'.

Le défi consiste à extraire les données d'une trace d'analyseur logique, puis à résoudre les épreuves suivantes. L'objectif est d'y retrouver une adresse e-mail (...@sstic.org).

Une nouveauté du concours de cette année, à notre connaissance, est l'introduction du système de LUM, une sorte de borne de validation intermédiaire, qui en quelque sorte divise un challenge en plusieurs parties. Ce système à été extrêmement positif pour notre motivation. Sans ces LUMs, les épreuves peuvent s'apparenter à une traversée du désert, dans laquelle nous ne savons pas si la direction est la bonne ni à quelle distance se trouve l'oasis. Nous avons collectés nos impressions dans la dernière partie de ce document, mais il se peux que la frustration passagère qu'on causée certaines épreuves apparaissent entre certaines lignes du texte.

Le challenge commence par le téléchargement d'un fichier archive au format zip (Bittendo_email_leak_by_shadowbrokers.zip) que nous commençons naturellement par explorer.

1) Bittendo_email_leak_by_shadowbrokers.zip

Fichiers attachés: 0_do.py

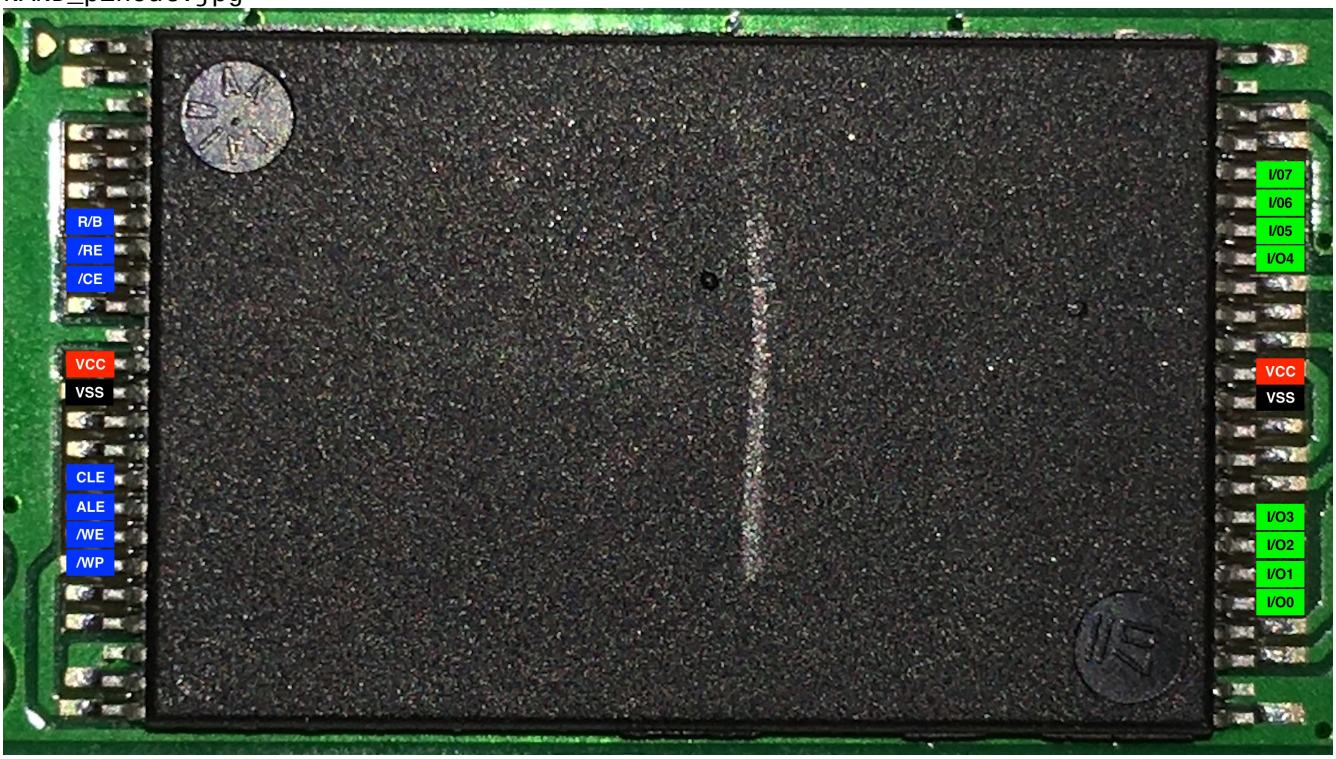
Bittendo_email_leak_by_shadowbrokers.zip contient un seul fichier: firmware.eml

Firmware.eml commence par ce texte: un courriel sauvegardé avec ses fichiers attachés: NAND_pinout.jpg et NAND_FLASH_writes_no_OOB_5MHz.sr

```
X-Mozilla-Status: 0001
X-Mozilla-Status2: 00000000
X-Mozilla-Keys:
FCC: mailbox://john.smith@localhost/Sent
X-Identity-Key: id1
X-Account-Key: account2
To: valerian.bedecho@bittendo.com
From: John Smith <john.smith@china-electronic-corp.com>
Subject: Validation du firmware
Message-ID: <614cd63d-36ce-7a59-4edd-bc9b42d0e2f3@china-electronic-corp.com>
Date: Fri, 17 Feb 2017 06:40:41 -0500
X-Mozilla-Draft-Info: internal/draft; vcard=0; receipt=0; DSN=0; uuencode=0;
 attachmentreminder=0
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100101
 Thunderbird/45.7.1
MIME-Version: 1.0
Content-Type: multipart/mixed;
 boundary="------------35AAF8A13FC1378B41D9ABFE"

This is a multi-part message in MIME format.
--------------35AAF8A13FC1378B41D9ABFE
Content-Type: text/plain; charset=utf-8; format=flowed
Content-Transfer-Encoding: 8bit

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 bindingspython et une
interface CLI, qui vous permettront certainement d'automatiser l'extraction.

Dans l'attente de votre validation.

Cordialement,
John
```

```
--------------35AAF8A13FC1378B41D9ABFE
Content-Type: image/jpeg;
 name="NAND_pinout.jpg"
Content-Transfer-Encoding: base64
Content-Disposition: attachment;
 filename="NAND_pinout.jpg"

/9j/4AAQSkZJRgABAQIAHAAcAAD/2wBDAAMCAgICAgMCAgIDAwMDBAYEBAQEBAgGBgUGCQgK
CgkICQkKDA8MCgsOCwkJDRENDg8QEBEQCgwSExIQEw8QEBD/2wBDAQMDAwQDBAgEBAgQCwkL
EBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBD/wAAR
```

NAND_pinout.jpg



L'attachement .srr est encodé en base64. Nous le sauvegardons sur disque et utilisons 'base64' pour le convertir en binaire. Le fichier “.sr” est un enregistrement des signaux électroniques. Nous utilisons 'sigrok-cli' pour extraire le signal et le passer à un script python pour le décoder.

```
sigrok-cli -i NAND_FLASH_writes_no_OOB_5MHz.sr | python do.py
```

Exemple de sortie de 'sigrok-cli':

```
I/O-0:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-1:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-2:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-3:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-4:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-5:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-6:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-7:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
CLE:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
ALE:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
WE:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

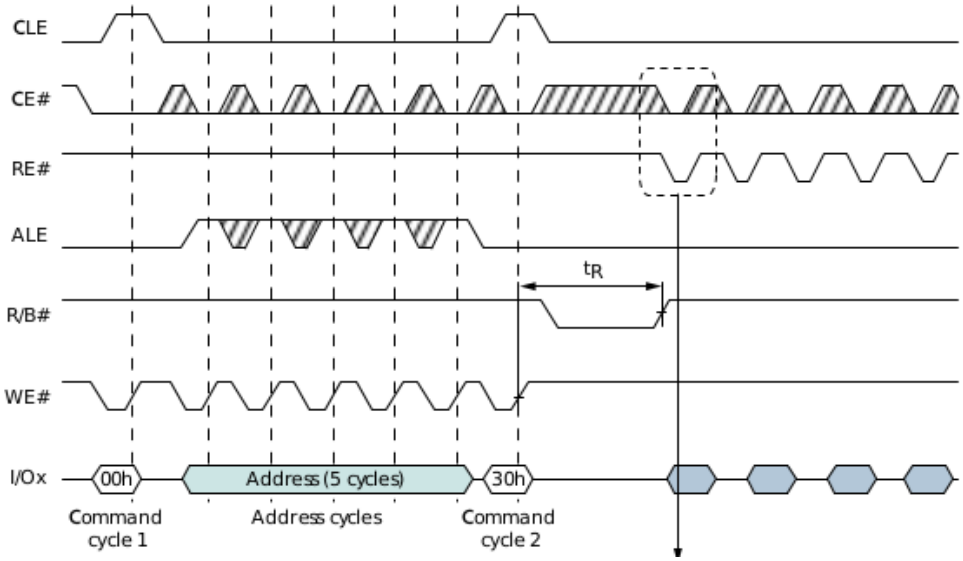
RE:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

I/O-N : 8 bits parallel line
CLE: Command Latch Enable
WE: Write-enable
RE: Read-enable
ALE: Adress Latch Enable

La trace enregistrée semble être un enregistrement de la programmation (au sens d'écriture) d'une mémoire flash. Nous nous sommes documentés sur la programmation des mémoires NAND en utilisant la documentation d'un fabricant (Micron). https://www.ece.umd.edu/~blj/CS-590.26/micron-tn2919.pdf

Quand le signal CLE s'élève (passe de basse tension à haute tension), les huit signaux de données (I/O-0 → I/O-7) sont lus et représentent le code de la commande.

Figure 4: Command Cycles for NAND Flash Operations



Quelques exemples de commandes tirés de la documentation fabricant:

Table 5: Command Cycles and Address Cycles

Command	Command Cycle 1	Number of Address Cycles	Data Cycles Required[1]	Command Cycle 2	Valid During Busy
READ PAGE	00h	5	No	30h	No
READ PAGE CACHE SEQUENTIAL	31h	–	No	–	No
READ PAGE CACHE SEQUENTIAL LAST	3Fh	–	No	–	No
READ for INTERNAL DATA MOVE	00h	5	No	35h	No
RANDOM DATA READ	05h	2	No	E0h	No
READ ID	90h	1	No	–	No
READ STATUS	70h	–	No	–	Yes
PROGRAM PAGE	80h	5	Yes	10h	No
PROGRAM PAGE CACHE	80h	5	Yes	15h	No
PROGRAM for INTERNAL DATA MOVE	85h	5	Optional	10h	No
RANDOM DATA INPUT	85h	2	Yes	–	No
ERASE BLOCK	60h	3	No	D0h	No
RESET	FFh	–	No	–	Yes

(Table reproduite ici sans aucune espèce d'autorisation, et de plein gré).

Certaines commandes nécessitent une adresse mémoire pour opérer, par exemple, ProgramPage (0x80). Le code ProgramPage est suivi de 5 octets d'adresse, capable d'adresser 1TO d'espace. Chaque octet d'adresse peut être lu quand le signal ALE passe de bas à haut.

Certaines commandes produisent des données à lire sur le bus. Pour ce faire, le lecteur attends que la ligne RE passe de bas à haut et, à ce moment, lis les états des lignes de données (IO-xxx).

Un script python a été écrit pour analyser et interpréter les données de sortie de sigrok-cli, et traduire les signaux électroniques en langage de commande NAND. En quelque sorte un émulateur de mémoire NAND. Chaque commande ProgramPage écrit dans un fichier, à l'adresse spécifiée. Si l'interpréteur fonctionne correctement, à la fin du flux de données, le fichier devrait contenir un listage de la mémoire NAND telle que programmée lors de l'enregistrement de la trace.

Seule quatre commandes NAND sont utilisées dans cet enregistrement: Reset, ProgramPage, ReadId et EraseBlock. Notre programme n'émule donc qu'elles.

La réponse à la commande ReadId nous donne notre premier LUM:
“SSTIC PSEUDO NAND LUM{x.g215WiPCR}”

Une fois la mémoire capturée, son interprétation peut commencer: la commande 'file' de Linux nous renseigne: il s'agit d'un 'x86 boot sector'. Il se laisse monter assez docilement, le système de fichier étant au format vfat.

Une fois montée, ce système révèle contenir deux fichiers: challenges.zip et challenges.zip.md5. Nous vérifions que le md5 correspond. Le fichier zip révèle un fichier README.TXT dont le contenu est reproduit ci-dessous:

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 permissions CAP_NET_RAW et CAP_NET_ADMIN ainsi que le droit d'exécuter la commande "ip" avec ces permissions) 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

Nous suivons les instructions et nous retrouvons dans une machine virtuelle implémentée en JavaScript et fonctionnant dans un navigateur web. Le réseau de la machine virtuelle passe par des “web-sockets”.

2) La machine

Une fois la machine virtuelle démarrée, nous nous retrouvons dans un terminal de type shell bien familier et rassurant. Pas pour longtemps hélas. Un fichier Readme.txt donne le ton:

Bienvenue aventurier !

Le Grand Protoon a été enlevé, l'équilibre du monde est rompu. Les Electroons ont été enfermés dans diverses épreuves, retrouve-les, et libère le Grand Protoon pour que le monde retrouve sa stabilité.

Les épreuves qui t'attendent sont dans le dossier "/challenges", certaines d'entre elles doivent être préalablement déverrouillées.

Lorsque tu auras réussi à libérer un Electroon, ajoute-le à l'aide de la commande "/challenges/tools/add_key". Les épreuves verrouillées se déverrouilleront automatiquement.

Dans chaque niveau, des Lums sont cachés, retrouve-les ! La plupart sont dissimulés sur le chemin vers l'Electroon. Pour les collectionner, ajoute-les avec la commande "/challenges/tools/add_lum". Il n'est pas essentiel de collecter l'ensemble des Lums pour finir ta quête.

Bonne chance !

Plusieurs informations sont extraites de ce document:
- Il y a plusieurs épreuves, une clef doit être trouvé pour chacune d'elle
- Il existe des LUM facultatifs.
- Nous sommes un “aventurier”!

3) Don't let him escape

Fichiers joints: 1_bpf.c, 1_unbpf.py, 1_main.c et 1_brutesrch.c

Le dossier /challenges/dont_let_him_escape/ contiens deux fichiers python et un fichier .so. Le fichier bpf.py offre une librairie de fonctions traitant de BPF (Berkeley Packet Filter), des filtres programmables de paquets réseaux. Les fonctions de bpf.py utilisent “mini-libc.so” via la libraire python 'ctypes' pour faire des appels systèmes.

Le BPF est un programme interprété dans le noyaux (par opposition à l'espace userland). Quand un programme BFP est attaché à une 'socket', il est invoqué à chaque paquet réseau passant par la 'socket'. A chaque invocation, le programme accède aux octets du paquet et, par un jeu de logique, peut rejeter ou accepter le paquet (d’où son nom de filtre). En outre, il est possible de communiquer avec un programme BPF par le biais d'une table associative. Des entrées de la table peuvent être ajoutées, modifiées ou supprimées par le code client (userspace) ainsi que par le programme BPF lui-même.

Les fonctions du fichier bpf.py sont les suivantes:

- bpf_map_create: création d'une table associative pour communiquer avec le programme
- bpf_map_lookup: retourne la valeur de la table associée à la clef '0'
- load_bpf_program: charge un programme BPF dans le noyau, il est en général vérifié à ce moment (libre au noyau de le transcoder, de l'optimiser pourvu que la sémantique reste inchangée).
- bpf_attach_socket: associe un programme BFP à une 'socket', les paquets passeront dès lors par le filtre BPF
- create_raw_socket: crée une 'socket' pour recevoir des paquets au niveau Ethernet.

Le fichier 'server.py' crée une socket en mode RAW, charge un programme BPF puis entre dans une boucle sans fin lisant les arrivées sur la 'socket'. Pour chaque paquet reçu, la valeur associée à la clef 0 de la table associative est consulté. Si la valeur est 1, le paquet contient un LUM: un appel au script 'add_lum' est effectué avec le contenu du paquet en argument. Si la valeur est 2, une clef à été trouvée et un appel est fait au script 'add_key'.
Le challenge consiste donc à trouvé quels paquets pourraient bien être accepté par le filtre BPF. Pour cela nous examinerons le code BPF. Une modification triviale du script server.py permet de sauvegarder le programme BPF dans un fichier.

Linux implémente les versions 1 et 2 du standard BPF (classic et extended). La doc vient avec les sources dur noyau Linux: “Documentation/networking/filter.txt”.
Chaque instruction BPF utilise 8 octets:

- opcode: 8bits
- regs: 8bits
- jump_offset: 16bits
- k: 32bits

'opcode' est le numéro de l'instruction. Pour les instructions qui incluent un saut de programme, jump_offset contient l'adresse de destination du saut. L'interprétation de 'k' dépend de l'instruction ('code').

Exemple:

```
bf 16 00 00 00 00 00 00
opcode=bf → class=opcode&7=7 (BPF_ALU64)
```

```
        src = opcode&0x8=1 (use src register, not k)
        code= opcode >> 4 = 0xb (BPF_MOV)
    regs=16 → src_reg=regs>>4=1 (R1), dst_reg=regs&f=6 (R6)
    jump_offset n'est pas utilisé dans cette instruction, k non plus. Si src
    (dans opcode) avait été mis à 1, k aurait été utilisé comme valeur à
    assigner.
    En 'C', cette instruction s'écrirait: R6 = R1;
```

Nous avons écrit un traducteur en langage python qui lis le binaire BPF et le traduit en un code source de type 'C', lisible par un humain et compilable par gcc. Les instructions non utilisées dans le programme du challenge ont été laissé de coté par souci de réserve.
Le BPF ne pose pas de difficulté à part quelques cas spéciaux:

- 64 bits immediate: les instructions load imm doubleword et load tableid requièrent une valeur de 64bits, or 'k' est de 32bits. Les développeurs de BPF ont décider d'utiliser l'espace de deux instructions dans ces cas la. Notre traducteur BPF vers 'C' traite ce cas correctement.
- Nous avons remarqué qu'une seule table associative était utilisé dans ce programme, nous avons donc omis de traduire l'instruction 'load tableid' correctement. Nous utilisons la valeur 12345 comme pointeur sur la table, mais cette valeur est la uniquement pour que le programme 'C' compile sans erreur.

Le programme BPF est donc traduit en langage 'C' et inclus dans une pilote en 'C' lui aussi qui fournit l'environnement pour l'exécution. Cet environnement est: la table associative initialisée, les fonctions 1, 2 et 3 que le BPF a le loisir d'appeler et une pile d'appel et de valeur référencée par le registre R10. En outre, le paquet réseau se trouve dans le registre BPF R1 au moment de l'appel. Le pilote initialise donc R1 juste avant d'exécuter le BPF traduit.

Cette exécution du BPF nous sert à suivre pas à pas la logique du BPF et ainsi de sculpter notre paquet réseau de sorte que la BPF l'accepte.

Le BPF commence par vérifier que le paquet Ethernet reçu est un paquet IP (0x0800), un paquet UDP (17), que le port destination est (0x539 = 1337 elite!). C'est-à-dire que si un de ces tests se trouve faillir, le programme BPF retourne la valeur 0, qui signifie que ce paquet doit ne pas être transmis.

La suite du programme est retranscrite ci-dessous (copie brute de mes notes durant le challenge).

1. R8 <- udppayload len
2. R9 <- updpayload offset
3. R10-24 : IP headers length (20)
4. R10-8 <- 0
5. R0 = lookup_map(12345, key=0)
6. R1 = *R0
7. if map value != 0 jump to KEY part
8. otherwise continue with LUM part
9. check R8 == 16 (R8 is the udp payload len)
10. check packet[R9+22] == 'L' (R9 is ip options length)
11. check packet[R9+23] == 'U'
12. R10-32 <- R9 + 30 (offset of udppayload[8])
13. R10-56 <- R9 + 26 (offset of udppayload[4])
14. R10-48 <- (32bits)udppayload[4]
15. R10-40 <- (32bits)udppayload[12]
16. R1 <- R10[-32]

```
17.      R8 = packet[R1] = (32bits)udppayload[8]
18.      R9 += 24
19.      R0 = (16bits)packet(R9) (udppayload[2:4])
20.      R8 <<= 32
21.      R1 <- R10[-40] ((32bits)udppayload[12])
22.      R8 |= R1  (donc R8 contains 64bits of payload: udppayload[8:16])
23.      R1 = 0x456443724D66417D ( end of the LUM key string)    "EdCrMfA}"
24.      check R1 == R8
25.      R1 <- R10[-40] ((32bits)udppayload[4])
26.      check R1 == 0x42765751 (first 4 chars of LUM key)       "BvWQ"
27.      check R0 == 19835 = 0x4D7B  "M{"
28.      update_map(12345, key=0, val=1)
29.      return -1
```

En suivant la logique, on remarque que le contenu du message UDP est comparé pièce par pièce à la chaîne de caractère “LUM={BvWQEdCrMfA}”. Voila donc le message que nous devons envoyer: un paquet UDP/IP au port destination 1337, de longueur de contenu 16 octets, ayant pour contenu cette chaîne. Pour vérifier cette découverte, nous créons un petit script python:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
msg = "LUM{BvWQEdCrMfA}"
s.sendto(msg,("10.42.42.2", 1337))
```

Exécuté sur la MV, avec server.py tournant sur un autre terminal, nous recevons le message:

```
/challenges/dont_let_him_escape $ su
/challenges/dont_let_him_escape # python server.py
table_fd = 4
bpf successfully loaded
Good job!, you find a LUM : LUM{BvWQEdCrMfA}
```

Ce qui semble prometteur pour la suite. La deuxième partie du programme BPF a rapport avec un autre paquet contenant une clef. Le programme commence par vérifier que le contenu du paquet UDP est long de 37 octets et que la valeur associée à la clef 0 est 1 (ce qui est le cas si le paquet LUM est déjà passé par là).
Ensuite le BPF vérifie que le contenu commence par “KEY{“ et se termine par “}”, ce qui laisse 32 caractères au milieu (lignes I0122: à I0166:).

De I0177 à I1051:, les 32 caractères de la clef sont interprétés comme caractères hexadécimaux et convertis en entier de 0 à 255. Puis ils sont placés sur la pile aux adresses suivantes par groupes de 4 octets: R10-32, R10-72, R10-80, R10-104 (R10 pointe sur la pile, ces emplacements sont donc des variables locales).

La clef est donc maintenant un nombre de 128 bits, qui est traité en quatre parties de 32bits.

Première partie de la clef ('p1'), lignes 1050 à 1061:

La première partie de 32 bits (appelé p1 ici) subit cette transformation (les valeurs littérales ont été récupérée en utilisant gdb sur le programme 'C'):

```
p1 ← // 4 premiers octets de la clé hex
p1 ^= 0xbd89a8ae;     // Value coming from the hashtable
udp_sz = 0x2d;
unint64_t m = udp_sz * 1886350457 = 0x13c3961145;
```

```
p1 ^= m;
valid = (p1 == 0x53535449); // (0x53535449 == "ssti")
```

Pour passer le test de validité de la dernière ligne, 'p1' doit satisfaire:
p1 = (0x53535449 ^ 0x13c3961145) ^ 0xbd89a8ae = 0x132d4ceda2
soit en ne gardant que les 32bits bas: 0x2d4ceda2.

La vérification en utilisant gdb est triviale et nous pouvons passer à la deuxième partie de la clef ('p2').

Deuxième partie de la clef ('p2'), lignes 1061 à 1083:

Le calcul est un peu différent, mais est résolu de manière similaire, en utilisant gdb.

```
p2;     // Value from the key (hex2i of key[16:32])
p2 ^= 0xba9bbc8d;        // Val from hashtable l.1076
uint64_t l = 1836020326 / 0x2d;
l += p2;
l <<= 32;
l >>= 32;
valid = (l == 0x43204348); // ("C CH")

solution:
p2 = (0x43204348 - (1836020326 / 0x2d)) ^ 0xba9bbc8d = 0xfa2a0e08
```

Troisième partie de la clef ('p3'), lignes 1084 à 1154:

```
p3;
p3 ^= 0xbd89a8ae
r2 = 0x8A46A52D;

r1 = p3
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * p3) % r2
valid = (r1 == 0x204c4c41); // “ LLA”
```

Pour trouver le p3 qui convient ici, un recherche exhaustive sur les entiers de 32bits est effectuée. La recherche trouve deux entiers qui valident le test de fin.! (voir 1_brutesrch.c).

```
solution:
0x41bfa3fb is a solution to r1, so p3 = r1 ^ 0xbd89a8ae = 0xfc360b55
0xcc064928 is a solution to r1, so p3 = r1 ^ 0xbd89a8ae = 0x718fe186
```

Quatrième partie de la clef ('p4'), lignes 1155 à 1234

```
p4;
```

```
p4 ^= 0xba9bbc8d
r2 = 0xe1886cdc

r1 = p4
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * r1) % r2
r1 = (r1 * p4) % r2
valid = (r1 == 0x37313032); // “7102” (2017 inversé)
```

Le problème est le même que pour 'p3' aux constantes près. La solution est recherchée de la même manière. Il n'existe qu'une seule solution.

solution:
0x93865b44 is a solution to r1, so p4 = r1 ^ 0xba9bbc8d = 0x291de7c9

En assemblant chaque partie de la solution p1, p2, p3 et p4, nous trouvons deux clefs possibles:
"KEY{2d4ceda2fa2a0e08fc360b55291de7c9}"
"KEY{2d4ceda2fa2a0e08718fe186291de7c9}"
seule la première passe le test du “/challenge/tools/add_key” (le deuxième erratum sur la page du challenge nous prévenait de ce dilemme).

De même que pour le paquet UDP LUM, nous envoyons un message UDP/IP de contenu "KEY{2d4ceda2fa2a0e08fc360b55291de7c9}" (sans les guillemets, 37 caractères). Le script server.py nous félicite une fois de plus, et nous prouve sa gratitude en débloquant l’énigme suivante, nous proposant ainsi de dépenser quelques jours supplémentaire de notre vie déjà trop courte.

NOTE: Pour que le message portant la KEY soit validé, il faut au préalable que le message portant le LUM ait été vu par le programme BPF (indication d'état dans la table associative).

NOTE: le contenu du message UDP étant interprété comme un nombre sous forme hexa, les caractères majuscules ou minuscules n'ont en principe pas d'importance. Cela est vérifié, le message en majuscule est admis par le programme BPF, mais la clef n'est pas considérée comme valide par la commande 'add_key'. (premier erratum sur la page du challenge).

4) Riscy zones

Fichiers attachés: 2_aes.py, 2_bitmasks.txt, 2_decrypt.c, 2_decrypt_file.py et 2_riscv2c.py.

Le challenge riscy_zones inclus quatre fichiers. Il s'agit de décrypter le contenu de l'un d'eux, ce qui nécessite de trouver un mot-de-passe. L'un des fichier est une démonstration avec un mot-de-passe donné, ce qui permet d'étudier le comportement des deux programmes.

Le programme 'trustzone_decrypt' est utilisé pour décrypter les fichiers. Il transmet au TEE (Trusted Execution Engine) dans le noyau un exécutable signé (TA.elf.signed, TA pour Trusted Application). Nous vérifiions que le TEE valide la signature en modifiant un seul bit de l'exécutable: le chargement est refusé.

Une fois le TA chargé (via CMD_LOAD_TA) on peut envoyer des messages au TA via CMD_TA_MESSAGE. Le format de ces messages est propre à chaque TA, en inspectant le code assembleur de 'trustzone_decrypt' nous obtenons quelques informations.

Nous commençons par décompiler le programme userspace 'trustzone_decrypt'. Readelf nous informe que l'ISA est "OpenRISC 1000". La commande 'strings' ne révèle pas grand chose à-part peut-être 'CMD_GET_TA_VERSION' qui semble être un enum de commande possible, ce qui suggère qu'il en existe d'autre. En effet, il existe dans /challenges/tools/tee_client.py une librairie pour accéder au TEE qui contient ces commandes:

```
class tee_message(ctypes.Structure):
    CMD_GET_VERSION = 0x0001
    CMD_LOAD_TA     = 0x0002
    CMD_TA_MESSAGE  = 0x0003
    CMD_UNLOAD_TA   = 0x0004
    CMD_CHECK_LUM   = 0x0005
    CMD_CHECK_KEY   = 0x0006
    MAX_LEN         = 8192
    _fields_ = [
        ('cmd', ctypes.c_int),
        ('data_in_len', ctypes.c_int),
        ('data_in', ctypes.c_char_p),
        ('data_out_len', ctypes.c_int),
        ('data_out', ctypes.c_char_p),
```

(La librairie tee_client.py est utilisé par les scripts add_lum et add_key pour validation).

Pour comprendre le format des messages envoyés au TA, nous désassemblons trustzone_decrypt grâce à l'exécutable 'objdump' fourni sur la MV.

Désassemblage

Ensuite, nous retrouvons quelques chaînes de caractères intéressantes dans le programme. Par exemple, pour trouver où la chaîne "retreived version : .[1m%s." est utilisée, nous procédons comme suit.

La chaîne se trouve dans l'exe à l'adresse 0x153c.

```
0001530: 5f54 415f 5645 5253 494f 4e00 7265 7472  _TA_VERSION.retr
0001540: 6569 7665 6420 7665 7273 696f 6e20 3a20  eived version :
```

```
0001550: 1b5b 316d 2573 1b5b 306d 0a00 5b1b 5b33  .[1m%s.[0m..[.[3
```

consultons la table des sections de l'ELF révélés par 'readelf -a'

```
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00002034 0x00002034 0x000e0 0x000e0 R E 0x4
  INTERP         0x00171e 0x0000371e 0x0000371e 0x00017 0x00017 R   0x1
      [Requesting program interpreter: /lib/ld-musl-or1k.so.1]
  LOAD           0x000000 0x00002000 0x00002000 0x018b4 0x018b4 R E 0x2000
  LOAD           0x0018b4 0x000058b4 0x000058b4 0x000f4 0x00114 RW  0x2000
  DYNAMIC        0x0018c8 0x000058c8 0x000058c8 0x00098 0x00098 RW  0x4
  GNU_EH_FRAME   0x001738 0x00003738 0x00003738 0x0004c 0x0004c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
```

L'adresse 0x153c tombe dans la première section LOAD, qui commence à l'offset 0 et se trouve être Readonly (flag R) et Exécutable (E). Cette section est chargée à l’adresse 0x2000, par conséquent notre chaîne aura pour adresse mémoire 0x353c.

Nous retournons au code désassemblé et cherchons si une instruction fait référence à l'adresse 0x353c et nous trouvons ceci:

```
              2e04:          18    60    00    00                 l.movhi    r3,0x0
   ←- Place 0 dans les 16bits hauts de r3
    2e08:   d4 01 10 00     l.sw 0(r1),r2    ←- r2 contient la version
    2e0c:   07 ff fd 8e     l.jal 2444         ←- printf
    2e10:   a8 63 35 3c     l.ori r3,r3,0x353c  ←- "retreived version : .[1m%s."
```

La dernière ligne reference notre adresse. De manière similaire, 'readelf' nous donne les adresses mémoires des fonctions 'extern' au programme (par exemple 'printf' est à l'adresse 0x2444):

```
Symbol table '.dynsym' contains 18 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00002430     0 FUNC    GLOBAL DEFAULT  UND ioctl
     2: 00002444     0 FUNC    GLOBAL DEFAULT  UND printf
     3: 00002458     0 FUNC    GLOBAL DEFAULT  UND memcpy
     4: 0000246c     0 FUNC    GLOBAL DEFAULT  UND puts
```

Cela nous permet de savoir que l'instruction 'l.jal 2444' est en fait un appel à printf.

Un mot sur les appels de fonctions dans openrisc (https://openrisc.io/or1k.html#__RefHeading__504737_595890882 pour la doc compléte):
les paramètres sont placés dans les registres r3 à r8 (6 registres de 32 bits). Les arguments de 64bits prennent deux registres. S'il y a plus d'argument que de place dans ces registres, le reste des arguments est placé sur la pile (registre r1 pointe sur le haut de la pile). En cas d'appel à fonction varadique, les arguments optionnels sont aussi placés sur la pile.

Une particularité de certaines architectures processeur et le 'delay slot' visible dans l'exemple: le premier argument 'r3' fourni à printf est modifié à la ligne assembleur *suivante* l'appel à printf. Ceci est du au fait que le processeur décode l'instruction suivante pendant l’exécution de l'instruction courante. L'instruction de saut 'l.jal' s’exécute, mais l'instruction suivante est quand même 'l.ori' car elle a déjà été décodée.

Le code désassemblé plus haut montre que 'r3' contient notre chaîne de format et que la chaîne de version (venu d'un appel précédent) est mis sur la pile (à r1+0). Il peut s'ecrire en 'C': printf("retreived version : .[1m%s.", version);

En procédant de manière similaire pour d'autres fonctions externes et d'autres chaînes de caractères de l'exe, beaucoup d'indices peuvent être collectés. Par exemple le format des messages et de la réponse de la commande CMD_TA_MESSAGE.

Examinons le code assembleur listé ci-dessous. Les deux premières lignes nous informent que ce code donne la commande CMD_DECRYPT_BLOCK au TA. Cette commande est envoyée sous forme d'un message CMD_TA_MESSAGE, envoyé au TEE. Nous avons en quelque sorte une commande TA dans une commande TEE.

La commande TEE est à stack + 52 (voir @3150:), la commande TEE est 3 (CMD_TA_MESSAGE voir @315c:). Le tampon d'entrée a pour adresse r14 (@3174) et est de longueur r24 (@3168:), le tampon de sortie est pointé par r2 (@318c:), et est de longueur r22 (@3180:). R24 a pour valeur 280 (@30dc:), et r22 = 260 (@30f8:). R2 et r22 sont alloués sur le tas mémoire par appel à malloc. (en bleu dans l'extrait ci dessous).
Le format du message lui-même est ce qui nous intéresse. Il est pointé par r14. (en vert dans l'extrait).

- . Les 4 premiers octets sont mis à 0x03000000 (cf @30c4: et @3130:). Ceci est le code de la commande TA, elle est de 3 içi donc CMD_DECRYPT_BLOCK=3.
- . A l'offset 4 (@3154:), est écris le numéro du bloc (contenu par r18), mais en inversant l'ordre des octets (pour passer de big-endian à little-endian, en rose ci-dessous).
- . Aux offset 8, 12, 16 et 20 sont écrits 4 entiers de 32bits qui sont en fait les données lues dans le fichier à décrypter stockées temporairement dans un tableau local sur la pile (le code correspondant est omis par souçi de place).

Le saut du programme vers l'adresse 2acc (@3188:) effectue un appel système de type ioctl sur le fichier '/dev/sttic'). Cet appel transmet au TEE la commande qui supposément transmet au TA la commande incluse.

```
    30d4:   9c 60 01 04     l.addi r3,r0,260
    30d8:   07 ff fc ea     l.jal 2480 malloc
    30dc:   9f 00 01 18     l.addi r24,r0,280
    30e0:   a8 78 00 00     l.ori r3,r24,0x0
    30e4:   07 ff fc e7     l.jal 2480 malloc
    30e8:   a8 4b 00 00     l.ori r2,r11,0x0
    30ec:   a8 b8 00 00     l.ori r5,r24,0x0
    30f0:   9c 80 00 00     l.addi r4,r0,0
    30f4:   a8 6b 00 00     l.ori r3,r11,0x0
    30f8:   9e c0 01 04     l.addi r22,r0,260
    30fc:   07 ff fc fa     l.jal 24e4 memset
    3100:   a9 cb 00 00     l.ori r14,r11,0x0
    3104:   a8 b6 00 00     l.ori r5,r22,0x0
    3108:   9c 80 00 00     l.addi r4,r0,0
    310c:   07 ff fc f6     l.jal 24e4 memset
    3110:   a8 62 00 00     l.ori r3,r2,0x0
    3114:   18 60 00 00     l.movhi r3,0x0
    3118:   07 ff fc d5     l.jal 246c puts
    311c:   a8 63 35 fc     l.ori r3,r3,0x35fc      // cst string "[.[34;1mi.[0m]
Send command to Trusted App CMD_DECRYPT_BLOCK"
    3120:   b8 92 00 58     l.srli r4,r18,0x18
    3124:   b8 b2 00 48     l.srli r5,r18,0x8
    3128:   b8 d2 00 18     l.slli r6,r18,0x18
    312c:   b8 72 00 08     l.slli r3,r18,0x8
    3130:   d4 0e f0 00     l.sw 0(r14),r30
    3134:   e0 c6 20 04     l.or r6,r6,r4
```

```
    3138:   a4 85 ff 00     l.andi r4,r5,0xff00
    313c:   18 a0 00 ff     l.movhi r5,0xff
    3140:   e0 86 20 04     l.or r4,r6,r4
    3144:   e0 63 28 03     l.and r3,r3,r5
    3148:   9c a0 00 03     l.addi r5,r0,3
    314c:   e0 84 18 04     l.or r4,r4,r3
    3150:   9c 61 00 34     l.addi r3,r1,52         // struct TA_cmd is @stack+52
    3154:   d4 0e 20 04     l.sw 4(r14),r4
    3158:   84 81 00 14     l.lwz r4,20(r1)
    315c:   d4 01 28 34     l.sw 52(r1),r5          // TA CMD = 3 (CMD_TA_MESSAGE)
    3160:   d4 0e 20 08     l.sw 8(r14),r4
    3164:   84 81 00 18     l.lwz r4,24(r1)
    3168:   d4 01 c0 38     l.sw 56(r1),r24
    316c:   d4 0e 20 0c     l.sw 12(r14),r4
    3170:   84 81 00 1c     l.lwz r4,28(r1)
    3174:   d4 01 70 3c     l.sw 60(r1),r14
    3178:   d4 0e 20 10     l.sw 16(r14),r4
    317c:   84 81 00 20     l.lwz r4,32(r1)
    3180:   d4 01 b0 40     l.sw 64(r1),r22
    3184:   d4 0e 20 14     l.sw 20(r14),r4
    3188:   07 ff fe 51     l.jal 2acc                  //
do_sstic_ioctl(cmd_struct *r3)
    318c:   d4 01 10 44     l.sw 68(r1),r2              // set out_ptr
```

Pour résumer, le format du message au TA commence par le numéro de commande TA (3 pour décrypter un bloc), puis dans cette commande particulière le numéro du bloc sur 4 octets en little-endian, puis les données du bloc.
En réponse, le tampon 'data_out' (r2 ici) contient le message décrypté.

Le désassemblage de 'trustzone_decrypt' montre que cet exécutable se contente de passer les données au TA qui fait le travail. Nous implémentons les fonctionnalités de ce programme en python pour plus de flexibilité (voir 2_decrypt_file.py).

En utilisant des méthodes similaires, nous récoltons des informations sur la façon dont l'executable communique avec le TA.

Le format d'un fichier crypté est:
112 octets de HMAC, 16 octets de XORKEY et N blocs de 16 octets, les données cryptées. Ces trois composants sont utilisés ci-dessous.

Pour décrypter un fichier, le mot-de-passe est d'abord envoyé au TA pour vérification (command CMD_VERIFY_PASSWORD=2). Dans cette étape, le mot-de-passe et les 0x70 premiers octets du fichier à décrypter sont envoyés au TA (ce qui représente le HMAC du mot-de-passe).

Le tampon d'entrée de la commande est formaté comme suit:
le code commande sur 4 octets en little-endian (0x02000000==2), la longueur du mot-de-passe sur 16bits en LE (0x1000==16), les 16 octets du mot-de-passe, la longueur du HMAC sur 16bits en LE (0x7000==112) suivi des 112 octets du HMAC.
Le HMAC se trouve dans le fichier à décrypter, ce sont les 112 premiers octets.
La commande communique son status en retour en utilisant les quatre premiers octets du tampon data_out.

Si le mot-de-passe est accepté par le TA, il est possible de décrypter le fichier. Pour décrypter le fichier, la commande CMD_ DECRYPT_BLOCK=3 est utilisée répétitivement par blocs de 16 octets jusqu’à la fin du fichier. Le format de la commande est 32LE(0x3) + 32LE(blocindex) + 16*octet(data). Les données décryptées sont reçues dans le tampon 'data_out', mais il reste à les passer par le filtre XOR

en utilisant la XORKEY du fichier.

Reversing du TA (Trusted Application)

Un readelf de 'TA.elf.signed' nous apprends qu'il s'agit d'un exe de machine 0xf3. Une recherche internet nous apprends qu'il s'agit de l'architecture 'risc-v' (https://riscv.org/, ISA docs ici: https://riscv.org/wp-content/uploads/2016/06/riscv-spec-v2.1.pdf). Pour décompiler le binaire, nous allons chercher et compilons les outils 'binutils' standards (objdump): https://github.com/riscv/riscv-tools

Une commande 'strings' sur le binaire révèle une liste (peut-être incomplète) de commandes, dont une en particulier est prometteuse:
[DEBUG] CMD CMD_TA_INIT
[DEBUG] CMD CMD_GET_TA_VERSION
[DEBUG] CMD CMD_GET_TA_LUM
[DEBUG] CMD CMD_CHECK_PASSWORD

Nous utilisons notre programme 2_decrypt_file.py pour tester la commande CMD_GET_TA_LUM,nous recevons LUM{gdN8.D*@+UV} .

Pour trouver le mot-de-passe, nous nous intéressons à la manière dont fonctionne la commande CMD_CHECK_PASSWD. Voici le code asm, avec nos commentaires (chaînes venues de la section des constantes et des fonctions externes) ajoutés:

```
   // CMD_CHECK_PASSWORD arrives here
   11a08:   00010537            lui a0,0x10
   11a0c:   14050513            addi    a0,a0,320 # 10140       // cst str "[DEBUG]
CMD CMD_CHECK_PASSWORD"
   11a10:   b4dff0ef            jal ra,1155c                    // do_printf(a0)
   11a14:   00445903            lhu s2,4(s0)
   11a18:   01000793            li  a5,16
   11a1c:   01645483            lhu s1,22(s0)
   11a20:   0127d663            ble s2,a5,11a2c                 // if
ushort(inbuf[4:5]) > 16:  This is the passwd length
   11a24:   fff00793            li  a5,-1
   11a28:   00f9a023            sw  a5,0(s3)                    //       outbuf[0]
= -1
   11a2c:   10000793            li  a5,256
   11a30:   0097d663            ble s1,a5,11a3c                 // if
ushort(s0[22:23]) > 256:  This is the len of the HMAC passed in msg
   11a34:   fff00793            li  a5,-1
   11a38:   00f9a023            sw  a5,0(s3)                    //       outbuf[0]
= -1
   11a3c:   00010537            lui a0,0x10
   11a40:   01840613            addi    a2,s0,24
   11a44:   04010593            addi    a1,sp,64
   11a48:   00048693            mv  a3,s1
   11a4c:   0c050513            addi    a0,a0,192 # 100c0       // cst str
"SSTIC_AES_KEY"
   11a50:   99dff0ef            jal ra,113ec <TEE_AES_decrypt>  //
TEE_AES_decrypt("SSTIC_AES_KEY", &stack[64], &inbuf[24], hmac_len)
                                                                // this will
probably decrypt the hmac using the named aes key in store
   11a54:   000105b7            lui a1,0x10
   11a58:   01900613            li  a2,25
   11a5c:   16458593            addi    a1,a1,356 # 10164       // cst str "==BEGIN
```

```
PASSWORD HMAC==\x0d\x0a" len(25)
   11a60:   04010513            addi    a0,sp,64                // points to the
decrypted hmac
   11a64:   00640413            addi    s0,s0,6                 // s0 now points to
passwd bytes
   11a68:   a89ff0ef            jal ra,114f0                    // memcmp(void*
a0_mem0, char *a1_mem1, int a2_len)
    11a6c:  24051863            bnez    a0,11cbc                // if memcmp
failed:  print errmsg and return
   11a70:   fe948513            addi    a0,s1,-23
   11a74:   000105b7            lui a1,0x10
   11a78:   04010793            addi    a5,sp,64
   11a7c:   01700613            li  a2,23
   11a80:   19858593            addi    a1,a1,408 # 10198       // cst str
"\x0d\x0a==END PASSWORD HMAC==" (len 23)
   11a84:   00a78533            add a0,a5,a0
   11a88:   a69ff0ef            jal ra,114f0                    // memcmp(void*
a0_mem0, char *a1_mem1, int a2_len)
   11a8c:   22051863            bnez    a0,11cbc                // if memcmp
failed:  print errmsg and return
   11a90:   02000613            li  a2,32
   11a94:   00c105b3            add a1,sp,a2                    // a1 point to
stack+32
   11a98:   05910513            addi    a0,sp,89                // a0 is now 25
(64+25=89) past start of decrypted buffer (see cst str above)
   11a9c:   969ff0ef            jal ra,11404                    // unhexlify(const
void* a0_src, void* a1_dest, int a2_len);
   11aa0:   24050263            beqz    a0,11ce4                // if unhexlify
failed:  print errmsg and return
   11aa4:   00010537            lui a0,0x10
   11aa8:   00090693            mv  a3,s2                       // s2 contains
passwd len
   11aac:   00040613            mv  a2,s0                       // s0 is ptr to
passwd bytes
   11ab0:   00010593            mv  a1,sp
   11ab4:   0e450513            addi    a0,a0,228 # 100e4       // cst str
"SSTIC_PASSWORD_HMAC_KEY"
   11ab8:   941ff0ef            jal ra,113f8 <TEE_HMAC>         // call
TEE_HMAC("SSTIC_PASSWORD_HMAC_KEY", sp, ptr_to_passwd, passwd_len)
   11abc:   02000613            li  a2,32
   11ac0:   00010593            mv  a1,sp
   11ac4:   00c10533            add a0,sp,a2
   11ac8:   a29ff0ef            jal ra,114f0                    // memcmp(a0=sp+32,
a1=sp, a2=32)
   11acc:   00050493            mv  s1,a0
   11ad0:   20051063            bnez    a0,11cd0                // if
TEE_HMAC(passwd) != hmac in msg:  got print "bad passwd" and return
   11ad4:   00010537            lui a0,0x10
   11ad8:   29850513            addi    a0,a0,664 # 10298       // cst str <"Good
password" in many colors>
```

Il est clair que le mot-de-passe est crypté en AES. La clef nommée "SSTIC_AES_KEY" est initializée à la chaîne de 16 caracteres: “___SSTIC_2017___” à l'initialization du TA (lors du traitement du message CMD_TA_INIT). Le code vérifie que une fois décrypté, le HMAC commence par la chaîne “==BEGIN PASSWORD HMAC==\x0d\x0a" et se termine par "\x0d\x0a==END PASSWORD HMAC==". Ensuite, le mot-de-passe est passé à la fonction TEE_HMAC(), qui calcule le HMAC correspondant et celui-ci est comparé au HMAC fournis. S’il compare favorablement, le mot-de-passe est mémorisé dans le TEE sous la clef “SSTIC_CUSTOM_KEY” et sera utilisé lors du décryptage des blocs.

Nous avons écris un décrypteur en python (aes.py) pour vérifier que les bloc HMAC des deux fichiers cryptés contiennent effectivement les marqueurs indiqués. (voir 2_aes.py)

Il n'est malheureusement pas possible de reverser un HMAC (fonction à sens unique). Cette voie est donc sans issue.

Nous nous tournons maintenant vers la command CMD_DECRYPT_BLOCK.

Pour chaque bloc transmis, cette commande calcule une valeur de 16 octets basé uniquement sur le numéro du bloc et sur le mot-de-passe transmis par la commande CMD_CHECK_PASSWORD. Puis calcule le xor de cette valeur et des 16 octets du bloc.
Ce résultat est transmis dans le tampon de retour 'data_out'.

Nous avons écrit un traducteur de l'assembleur risc-v vers du langage 'C' pour mieux apprécier les détails de la fonction. Le traducteur est implémenté dans 2_riscv2c.py).
Seule la partie du TA qui calcule la valeur XOR est traduite, nous n'avons donc pas besoin de supporter des instructions risc-v qui ne s'y trouvent pas (par exemple un 'call' ou 'jmp').
Un pilote écrit en 'C' est ensuite écrit pour préparer à l'appel de la fonction traduite (voir 2_decrypt.c).

En inspectant la construction de la valeur_xor, il apparaît que les 128 bits du mot-de-passe n'affectent pas les 128 bits de la valeur_xor de manière uniforme. En fait, il est facile de constater que le dernier octet de la valeur_xor est calculée en ne tenant compte que de certains bits du mot-de-passe.

Nous nous penchons sur ce détail. Voici le code asm de TA.elf.signed qui a trait au calcul du dernier octet de la valeur_xor (8 bits bas du registre t1) et plus bas sa traduction en 'C' en gardant le nom des registres comme nom de variables.

```
contexte: s1==bloc_num, a3=passwd[4:8], t5=passwd[12:16], t0 = 0x52555655, s2 =
0xadaaa9aa
   11680:   01748713            addi    a4,s1,23                // a4 <-
bloc_number + 23
   11684:   012f7633            and a2,t5,s2
   11688:   0056f7b3            and a5,a3,t0
   1168c:   01f77713            andi    a4,a4,31
   11690:   00c7e7b3            or  a5,a5,a2                    // a5 <- (key12 &
0xadaaa9aa) | (key4 & 0x52555655)
   11694:   40e00633            neg a2,a4                       // a2 <- -23 <-
fffffe8 (approx)!!! (for first blocK)
   11698:   00e7d733            srl a4,a5,a4
   1169c:   00c797b3            sll a5,a5,a2
   116a0:   00f76733            or  a4,a4,a5                    // a4 <- rotate
bits of a5 in a circular manner to the right
   116a4:   01875313            srli    t1,a4,0x18              //
   116a8:   04130313            addi    t1,t1,65                // t1 <- 65 +
(a4>>24)

a4 = s1 + 23;
a2 = t5 & s2;
a5 = a3 & t0;
a4 = a4 & 31;
a5 = a5 | a2;
```

```
a2 = -a4;
a4 = a5 >> a4;
a5 = a5 << a2;
a4 = a4 | a5;
t1 = a4 >> 0x18;
t1 = t1 + 65;
```

On peut tout à fait imaginer ici que nous regardons le premier bloc et donc poser bloc_num==0, pour simplifier la première analyse. Remarquons que les constantes t0 et s2 sont opposées (NOT binaire) l'une de l'autre.

On peut réécrire cela:

```
a4 = 23
a2 = -23 = 0xFFFFFFE9 = 9 (seul les 5 bits bas sont utilisés dans un shift)
a5 = (passwd[12:16] & s2) | (passwd[4:8] & not s2)
// A ce moment, a5 n'est basé que sur 32bits du mot-de-passe (pris parmi les
64bits mentionnés).
a4 = (a5>>23) | (a5<<9) = rotation_droite(a5, 23)
t1 = ((a4 >> 24) + 65) & 255
```

a5 est une copie de 32bits du mot-de-passe, a4 est une simple rotation des bits de a5. Ensuite seuls les 8bits hauts de a4 sont utilisés. Donc on voit clairement que seuls 8 bits du mot-de-passe sont utilisé pour l'octet de poids fort de value_xor.

Si l'on connaissait la valeur du 16ème octet du premier bloc en clair ainsi que le 16ème octet du premier bloc crypté, nous saurions quelle valeur_xor a été appliquée et nous pourrions ainsi remonter jusqu'aux 8 bits du mot-de-passe utilisés.
Deux questions se posent:

- comment faire pour le reste des bits du mot-de-passe, est-ce que les autres octets du valeur_xor sont aussi basés sur un petit nombre de bits du mot-de-passe?
- comment savoir quelles sont les données du bloc décrypté? Et ainsi pouvoir remonter jusqu’à la valeur_xor.

comment faire pour le reste des bits du mot-de-passe?

Commençons par la première question. Ayant à notre disposition la routine qui calcule la valeur_xor en 'C', nous décidons de choisir un mot-de-passe au hasard, de calculer la valeur_xor correspondante disons pour le bloc 0 (appelons la vx1 pour valeur_xor 1), puis de modifier un seul bit du mot-de-passe et d'observer quels bits de la valeur_xor ont changé. Pour être sûr de notre affaire, nous répétons ce test 1000 fois (avec donc un mot-de-passe pris au hasard 1000 fois). Cette procédure se trouve dans le fichier 2_decrypt.c.

Nous effectuons ce calcul pour chacun des 128 bits du mot-de-passe et observons comment chacun des 128bits de la valeur_xor est affecté. Le résultat est représenté dans la table suivante:

```
 0:  17529 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 00 00 00 00 00 00
 1:   5730 f0 f0 f0 00 00 00 00 00 00 00 00 00 00 00 00 00
 2:  10000 80 80 80 80 80 80 80 80 80 80 00 00 00 00 00 00
 3:   4491 c0 c0 c0 00 00 00 00 00 00 00 00 00 00 00 00 00
 4:  22025 fe fe fe fe fe fe fe fe fe fe fe 00 00 00 00 00
 5:   8032 ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00
 6:  21163 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 00 00 00 00 00
 7:   7905 fc fc fc fc 00 00 00 00 00 00 00 00 00 00 00 00
 8:   7743 f8 f8 f8 f8 00 00 00 00 00 00 00 00 00 00 00 00
 9:  16499 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 c0 00 00 00 00 00
10:  11000 80 80 80 80 80 80 80 80 80 80 80 00 00 00 00 00
```

```
 11:   5970 c0 c0 c0 c0 00 00 00 00 00 00 00 00 00 00 00 00
 12:  23620 fe fe fe fe fe fe fe fe fe fe fe fe 00 00 00 00
 13:   1999 ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 14:  23202 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 00 00 00 00
 15:   1976 fc 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 16:  21012 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 00 00 00 00
 17:   1882 f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 18:  12000 80 80 80 80 80 80 80 80 80 80 80 80 00 00 00 00
 19:   1491 c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 20:  17832 fe fe fe fe fe fe fe fe fe 00 00 00 00 00 00 00
 21:   3975 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 22:  17420 f8 f8 f8 f8 f8 f8 f8 f8 f8 00 00 00 00 00 00 00
 23:   3946 fc fc 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 24:   3902 f8 f8 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 25:  13577 c0 c0 c0 c0 c0 c0 c0 c0 c0 00 00 00 00 00 00 00
 26:   3503 e0 e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 27:   3021 c0 c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 28:  19872 fe fe fe fe fe fe fe fe fe fe 00 00 00 00 00 00
 29:   5887 ff ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00
 30:  19305 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 00 00 00 00 00 00
 31:   5957 fc fc fc 00 00 00 00 00 00 00 00 00 00 00 00 00
 32:  27479 fe fe fe fe fe fe fe fe fe fe fe fe fe fe 00 00
 33:  13827 ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00 00
 34:  26965 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 00 00
 35:  13791 fc fc fc fc fc fc fc 00 00 00 00 00 00 00 00 00
 36:  24446 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 00 00
 37:  13171 f0 f0 f0 f0 f0 f0 f0 00 00 00 00 00 00 00 00 00
 38:  14000 80 80 80 80 80 80 80 80 80 80 80 80 80 80 00 00
 39:  10433 c0 c0 c0 c0 c0 c0 c0 00 00 00 00 00 00 00 00 00
 40:   7000 80 80 80 80 80 80 80 00 00 00 00 00 00 00 00 00
 41:  29867 fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc 00
 42:  28540 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 00
 43:  15890 fc fc fc fc fc fc fc fc 00 00 00 00 00 00 00 00
 44:  26284 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 00
 45:  14979 f0 f0 f0 f0 f0 f0 f0 f0 00 00 00 00 00 00 00 00
 46:  15000 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 00
 47:  11999 c0 c0 c0 c0 c0 c0 c0 c0 00 00 00 00 00 00 00 00
 48:  31691 fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe fe
 49:   9940 ff ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00
 50:  30313 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8 f8
 51:   9930 fc fc fc fc fc 00 00 00 00 00 00 00 00 00 00 00
 52:  27309 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0
 53:   9359 f0 f0 f0 f0 f0 00 00 00 00 00 00 00 00 00 00 00
 54:  16000 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80 80
 55:   7524 c0 c0 c0 c0 c0 00 00 00 00 00 00 00 00 00 00 00
 56:   5000 80 80 80 80 80 00 00 00 00 00 00 00 00 00 00 00
 57:  25429 fc fc fc fc fc fc fc fc fc fc fc fc fc 00 00 00
 58:  12089 fe fe fe fe fe fe 00 00 00 00 00 00 00 00 00 00
 59:  11997 fc fc fc fc fc fc 00 00 00 00 00 00 00 00 00 00
 60:  22832 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 e0 00 00 00
 61:  11242 f0 f0 f0 f0 f0 f0 00 00 00 00 00 00 00 00 00 00
 62:  13000 80 80 80 80 80 80 80 80 80 80 80 80 80 00 00 00
 63:   8989 c0 c0 c0 c0 c0 c0 00 00 00 00 00 00 00 00 00 00
 64:   5798 f8 f8 f8 00 00 00 00 00 00 00 00 00 00 00 00 00
 (table tronquée par soucis des arbres de la forêt)
  table complète en appendice dans 2_bitmasks.txt
```

chaque ligne represente le test sur un bit du mot-de-passe (par exemple la ligne 48: n'a modifié que le bit 48 du mot-de-passe). La valeur de 128 sur la ligne est

le masque de tous les bits de la valeur_xor qui on changé au moins une fois lors des 1000 tests de ce bit du mot-de-passe. Donc sur la ligne 48, 7 bits sur 8 ont été modifiés, soit 112 bits sur 128).

Il est très clair que le dernier octet de la valeur_xor (le numéro 15, le groupe le plus à droite dans la table n'est affecté que par 8 bits du mot-de-passe. Il serait donc possible de déterminer ces 8 octets en connaissant le message en clair et en crypté.
L'octet 14 (avant dernier sur chaque ligne) est affecté par 16 bits du mot-de-passe, mais 8 d'entre eux sont ceux qui affectent l'octet 15 et qui seraient donc déjà déterminés.
Ainsi, de proche en proche, il semble que chacun des 16 octets de la valeur_xor ne dépendent que d'un groupe réduit de 8 bits du mot-de-passe.
L'on pourrait ainsi, en connaissant le premier bloc de 16 octets d'un message et de sa valeur crypté en déduire les 128 bits du mot-de-passe. Ce qui nous ramène à notre seconde question:

comment savoir qu'elle sont les données du bloc décrypte?

Un indice nous est donné dans le nom du fichier: secret.lzma.encrypted. Il s'agirait d'un fichier compressé au format LZMA, peut-être qu'un en-tête de fichier pourrait être deviné?
En utilisant la commande 'lzma' de mon système sur plusieurs fichiers, et en observant les premiers octets, il semble que 14 sur les 16 premiers octets soient constant. Avec un peu de chance c'est le cas du fichier secret.lzma et l'on pourrait toujours essayer toutes les valeurs possibles des 2 derniers octets.

Quelques fichiers .lzma listés avec 'xxd':
```
0000000: 5d 00 00 80 00 ff ff ff ff ff ff ff ff 00 3a 1a  ].............:.
0000000: 5d 00 00 04 00 ff ff ff ff ff ff ff ff 00 05 14  ].............
0000000: 5d 00 00 10 00 ff ff ff ff ff ff ff ff 00 2e 7f  ].............
```

Bien entendu, s'il y a un en-tête, c'est que les octets doivent pouvoir prendre des valeurs différentes (la principale fonction d'un format de compression est de réduire l'espace, après tout!). Mais on peux toujours espérer naïvement et suivre sa bonne étoile.

La procédure “static int searchPasswd()”, dans decrypt.c implémente la dérivation du mot-de-passe. Elle se divise naturellement en deux parties: la collection des données, et la recherche du mot-de-passe.

Collection des données

Pour commencer, la correspondance des bits du mot-de-passe au bits de la valeur_xor est capturé par la procédure “detect()”. Comme expliqué précédemment, pour chaque bit des 128 du mot-de-passe, le masque des bits de la valeur_xor affectés est trouvé. Le masque est naturellement de 128bits.
La variable locale 'in2OutChangeMask' capture cette correspondance, et est utilisée pour présenter la table montrée plus haut (dans le code, 'in' fait référence au mot-de-passe, la donnée d'entrée et 'out' à la valeur_xor, le résultat du calcul).

La variable 'Out2InChangeMask' capture la correspondance inverse: pour chaque bits de la valeur_xor le masque des bits du mot-de-passe qui les affectent. Elle est construite par simple inversion de 'in2OutChangeMask'. Finalement, à partir de 'Out2InChangeMask', nous dérivons pour chaque octet de valeur_xor le masque des bits du mot-de-passe correspondant. Cette structure est un tableau de 16 * 128 bits appelé BitGroup.
Chaque BitGroup contient:

- outMask: Un masque indiquant quels bits de valeur_xor sont concernés
- numPasswdBits: le nombre des bits du mot-de-passe qui influent sur les bits concernés
- passwdBitPos[]: la liste des bits du mot-de-passe qui influent sur les bits concernés

Recherche du mot-de-passe

La recherche du mot-de-passe se décompose en 16 recherches de 8bits, comme nous l'avons expliqué précédemment. La recherche des bits du mot-de-passe aboutissant à un octet de valeur donnée de la valeur_xor s'effectue comme suit. D'abord, le BitGroup correspondant à l'octet recherche est chargé (ctx->bg[j] ou j est le numéro de l'octet, de 0 à 16). Pour chaque bits du mot-de-passe correspondant (bg->numPasswdBits, bg->passwdBitPos[n]), une recherche exhaustive est menée: toutes les valeurs possibles des bits du password de ce groupe sont explorées, la valeur_xor correspondante est calculée et si la valeur de l'octet j a la valeur recherchée, nous passons de manière récursive à l'octet j-1. Si aucune valeur ne satisfait la recherche, la procédure retourne -1. Il est important que cette procédure soit récursive et que l'avancement de recherche de chaque BitGroup soit mémorisé dans des variables locales: cela permet de continuer la recherche quand un BitGroup ne trouve pas de solution.

Une complication est que certains octets de la valeur_xor ne sont pas connus, parce que l'en-tête d'un fichier lzma ne compte que 14 octets, les octets 14 et 15 ne sont pas à priori fixés. Ils devront pouvoir prendre n'importe quelle valeur. Du point de vue de l'implémentation, la façon d'arriver à cela est de simplement mettre à zéro leur outMask: cela signifie que le test qui vérifie que la valeur_xor est correct sera vrai pour chaque valeur du mot-de-passe.

Au final, nous obtenons 65536 mots-de-passe possibles. Chacun des ces candidats est testé individuellement via notre implémentation en python de l'appel à CMD_CHECK_PASSWD sous la MV.

```
'gunzip -c candidates.gz | python decrypt_file.py 0 infile destfile'
('Foundone: ', '5921cd9fd3a82bd9244ece5328c6c95f')
```

Nous vérifions que le mot-de-passe est en plus la clef du challenge:

```
/challenges/tools/add_key 5921cd9fd3a82bd9244ece5328c6c95f
ok!
```

Résumé:

```
      Set password to 16x0
      Set expected_value
      for octectNum 15 downto 0:
            bg = bitgroup of octectNum
            for all possible value of bg->passwdBits:
                  decrypted_bloc = decrypt( password, crypted_bloc )
                  if (decrypted_bloc & bg->bitMask) == (expected_bloc & bg-
>bitMask)
                        // move on to next bitGroup
```

Enfin nous utilisons le mot-de-passe pour décrypter le fichier secret.lzma.encrypted. Nous obtenons un fichier lzma, qui décompressé nous donne un fichier .jpg que voici:

Nous trouvons dans les données parallèles EXIF du fichier jpg cette chaîne :
: Congratulations : YHZ{+g%Yi.vzG8Z}

Nous la décodons grâce à ce site: http://decode.org/?q=YHZ%7B%2Bg%25Yi.vzG8Z%7D
et obtenons: LUM{+t%Lv.imT8M}

5) Unstable machines

Fichiers attachés: 3_checkboxes.png, 3_img_678x306_24bpp_BGR.data, 3_interp.py, 3_mem_datasection.bin, 3_password.png, 3_trace.txt, 3_waterdis.py, 3_waterdis.txt, 3_watermark.bin, 3_patch_disasm.py et 3_waterscript.c.

La challenge précédent débloque l'accès à cette nouvelle épreuve: un simple fichier: /challenges/unstable_machines/unstable.machines.exe
Le nom suggère un fichier exécutable pour système Microsoft Windows. Nous le transférons sur notre machine principale tournant sous Linux et lançons donc la commande:

```
wine unstable.machines.exe
```

Première partie:

Voici ce qui apparaît:

Les cases-à-cocher sont cliquables mais se comportent de manière aléatoire, ou tout du moins erratique. Certaines actions marquent les cases d'autres les remettent à zéro, le texte sur le bouton semble changer à chaque action. Il semblerait que nous devrions peut-être jeter un coup d’œil sous le capot de cette drôle de machine vraiment peu cohérente.

C'est parti!

Nous utilisons nos vieux camarades 'objdump' et 'vim' pour essayer de comprendre. Nous processons le fichier désassemblé de objdump pour marquer les chaînes de

caractère là ou elles sont utilisés et pour marquer les appels de fonctions externes par leur nom plutot que par leur adresse (voir 3_patchdisasm.py).
Nous trouvons sans mal les textes du bouton dans l'exécutable, en déduisons leurs adresses mémoire et ainsi remontons jusqu'aux instructions qui changent le texte du bouton:

```
  4011c0:   55                      push   %ebp
  4011c1:   8b ec                   mov    %esp,%ebp
  4011c3:   83 ec 1c                sub    $0x1c,%esp
  4011c6:   a1 00 30 41 00          mov    0x413000,%eax
  4011cb:   33 c5                   xor    %ebp,%eax
  4011cd:   89 45 fc                mov    %eax,-0x4(%ebp)
  4011d0:   56                      push   %esi
  4011d1:   8b f1                   mov    %ecx,%esi
  4011d3:   c7 45 e4 2c 1d 41 00    movl   $0x411d2c,-0x1c(%ebp)            # cst
str @0x1032c: 'No problem!'
  4011da:   c7 45 e8 38 1d 41 00    movl   $0x411d38,-0x18(%ebp)            # cst
str @0x10338: 'YEAH!'
  4011e1:   c7 45 ec 40 1d 41 00    movl   $0x411d40,-0x14(%ebp)            # cst
str @0x10340: 'Excellent!'
  4011e8:   c7 45 f0 4c 1d 41 00    movl   $0x411d4c,-0x10(%ebp)            # cst
str @0x1034c: 'I'm waiting!'
  4011ef:   c7 45 f4 5c 1d 41 00    movl   $0x411d5c,-0xc(%ebp)         # cst str
@0x1035c: 'Yo, dude!'
  4011f6:   c7 45 f8 68 1d 41 00    movl   $0x411d68,-0x8(%ebp)         # cst str
@0x10368: 'Yaoouuh!'
  4011fd:   e8 71 20 00 00          call   0x403273
```

La fonction de l'API Windows 'CheckDlgButton' sert à modifier l'état d'un bouton ou d'une case-à-cocher. Nous trouvons deux sites d'appel de cette fonction, l'appel en 4012f0: remet toutes les cases-à-cocher à zéro (les adresses sont les adresses memoires, pas fichiers). Cette fonction devrait être appelé plusieurs fois, en particulier quand l'utilisateur clique sur les cases.

```
  4012f0:   56                      push   %esi
  4012f1:   8b 35 4c e1 40 00       mov    0x40e14c,%esi            # cst str
@0xc74c: 'Î&^A'           # cst str @0xc74c: 'CheckDlgButton'
  4012f7:   57                      push   %edi
  4012f8:   6a 00                   push   $0x0
  4012fa:   8b f9                   mov    %ecx,%edi
  4012fc:   68 ed 03 00 00          push   $0x3ed
  401301:   57                      push   %edi
  401302:   ff d6                   call   *%esi
  401304:   6a 00                   push   $0x0
  401306:   68 f6 03 00 00          push   $0x3f6
  40130b:   57                      push   %edi
  40130c:   ff d6                   call   *%esi
```

Nous utilisons 'winedbg', le debuggeur intégré à wine pour tracer l'origine de l'appel et l'on se rends vite compte ou arrivent les messages générés par les clics sur les cases-à-cocher (401430:). En fait, tous les messages du dialogue arrivent dans cette fonction, même les mouvements de la souris ce qui rend les point d'arrêts peu pratique. Après quelques sessions sous le deboggeur, nous arrivons aux conclusions suivantes:

La première case-à-cocher a l'identifiant 0x3ed, et un clic dessus arrive à l'adresse 0x0040154e.

Voici le code à cette adresse:

```
  40154e:   a1 24 66 41 00          mov    0x416624,%eax
  401553:   8b 75 08                mov    0x8(%ebp),%esi
  401556:   83 f8 06                cmp    $0x6,%eax
// Lire l'entier à l'adresse  0x416624 (variable globale du programme).
// s'il a la valeur 6 passer à l'adresse 0x401583.
// Sinon continuer
  401559:   74 28                   je     0x401583
// si la variable globale a la valeur 8, passer à l'adresse 0x401583.
// Sinon continuer
  40155b:   83 f8 08                cmp    $0x8,%eax
  40155e:   74 23                   je     0x401583
// La variable n'a ni la valeur 6, ni 8, la mettre à 0, et remettre à zéro
(décocher) toutes les cases-à-cocher t changer le texte du bouton poussoir.
// puis retourner 1
  401560:   8b ce                   mov    %esi,%ecx
  401562:   c7 05 24 66 41 00 00    movl   $0x0,0x416624
  401569:   00 00 00
  40156c:   e8 7f fd ff ff          call   0x4012f0         // Reset the checkboxes
  401571:   8b ce                   mov    %esi,%ecx
  401573:   e8 48 fc ff ff          call   0x4011c0         // set the dialog main
button text
  401578:   5f                      pop    %edi
  401579:   b8 01 00 00 00          mov    $0x1,%eax
  40157e:   5e                      pop    %esi
  40157f:   5d                      pop    %ebp
  401580:   c2 10 00                ret    $0x10

// Si la variable globale (dans eax) est 6 ou 8, on arrive ici.
// incrementer la variable globale
  401583:   40                      inc    %eax
  401584:   a3 24 66 41 00          mov    %eax,0x416624
// Changer une autre variable globale (addr 0x4140f4) d'une manière non élucidée
  401589:   e8 42 07 00 00          call   0x401cd0             # purefunc returns
0xf4b02f4b
  40158e:   01 05 f4 40 41 00       add    %eax,0x4140f4
  401594:   8b ce                   mov    %esi,%ecx
// changer le texte du bouton poussoir.
  401596:   e8 25 fc ff ff          call   0x4011c0         // set the dialog main
button text
  40159b:   5f                      pop    %edi
  40159c:   b8 01 00 00 00          mov    $0x1,%eax
  4015a1:   5e                      pop    %esi
  4015a2:   5d                      pop    %ebp
// Puis retourner 1 (la valeur dans le registre eax)
  4015a3:   c2 10 00                ret    $0x10
```

En résumé, il semble que la variable globale à 0x416624 doit être à 6 ou 8, et dans ce cas elle passe à 7 ou 9, sinon on remet tout à zéro. En procédant de manière similaire pour les sept autres cases-à-cocher, nous trouvons ceci:

Ici GV est la variable globale qui nous intéresse.
1ère case:

```
    if GV == 6 or 8
        GV = GV + 1
```

2ème case:

```
    if GV == 3
        GV = 4
```

```
3ème case:
    if GV == 7
        set GV to 8
    else if GV == 10
        call GOAL!!!!!!!!!!!!!  arrives à 0x4017e0

4ème case:
    if GV == 1 or 5
        GV = GV + 1

5ème case:
    GV = 1

6ème case:
    if GV == 2
        GV = 3

7ème case:
    if GV == 4 or 9
        GV = GV + 1
```

Nous avons omis le cas of le 'if' est faux. Dans ces cas, pour toutes les cases, GV est mis à zéro et l'état de chaque case est mis à non-coché.

On remarque que la case 5 sort du lot, en effet GV est mis à 1 de manière inconditionnelle. De même, la case 3 est spéciale en cela que si GV=10 elle fait un appel différent.
On voit donc que pour arriver GV==10, if faut cliquer les cases dans le bon ordre, que l'on déduit facilement: 5 4 6 2 7 4 1 3 1 7 3

Quand cela est fait, effectivement la fenêtre de dialogue change:

Deuxième partie:

Dans la nouvelle interface les cases-à-cocher ont laissé place à une boite entrée de texte de type mot-de-passe (les caractères entrés sont représentés par des étoiles '*'). Le bouton poussoir se lit maintenant 'Sauver le monde !' et le texte ne change plus quand on l'active. Le bouton met environ 5-6 secondes à retrouver son état par défaut après une activation.

L'utilisation de winedbg (ou mieux de wine –gdb) nous apprends assez vite que l'appui du bouton se trouve toujours arriver à l'adresse 4015c0:. Maintenant que la première épreuve est passée, le bouton agit différemment (la variable globale 0x416610 vaut maintenant 1).

En décodant le code à cette adresse, l'on trouve la logique suivante:

- Changer le texte du bouton à “Vérification...” (SetWindowTextA)
- Aller chercher le texte entré par l'utilisateur en variable globale (adresse 0x4165e8, tableau de 32 caractères) via GetDlgItemTextA(), puis copier ce texte à une adresse obtenue on-ne-sait-trop-comment (pour encore).
- Appel de la fonction 0x402910
  40163a: e8 d1 12 00 00 call 0x402910
- Puis après deux copies de valeurs de 32octets et l'application d'un xor, les deux valeurs de 32bytes sont comparées et si elle sont égales, le texte “Tu as réussi ! Le Grand Protoon est sauvé !” est utilisé, ce qui semble être une chose désirable pour la suite du challenge et le monde en général.

Si cette logique semble plutôt simple, c'est que la complexité se cache dans ce petit appel de fonction anodin, une sorte de boite de pandore version assembleur.

Premier interpréteur (script en .rodata), fonction 0x402910

La ligne suivante suggère un interpréteur de codes scripts. En effet, cette instruction effectue un saut de programme à une adresse trouvée dans une table. Ceci est l'une des manières qu'un compilateur traduit l'idiome 'switch ()... case' du langage 'C' en code machine.

```
402947:   ff 24 85 20 31 40 00    jmp    *0x403120(,%eax,4)    # interpreter dispatch
```

La table à 0x403120 devrait donc contenir un lot de pointeurs de code de 32bits.

Voici cette table:

```
// opcode  addr
// 0x00    0x4030af
// 0x01    0x402bd3
// 0x02    0x403060
// 0x03    0x402c84
// 0x04    0x4030fc
// 0x05    0x402ee0
// 0x06    0x402da5
// 0x07    0x402b43
// 0x08    0x402f42
// 0x09    0x402abb
// 0x0a    0x403017
```

```
// 0x0b    0x402c37
// 0x0c    0x402e6e
// 0x0d    0x402a33
// 0x0e    0x402dc0
// 0x0f    0x402e0d
// 0x10    0x40294e   // jmp ESI_table( esi ) if esi < 4 else next insn
// 0x11    0x4030f4   // call ExitProcess(-1)
```

Comme exemple, étudions l'implémentation de l'opcode 0x05:

```
  402ee0:   e8 9b f2 ff ff          call   0x402180              #
linklist_read_next_byte()
  402ee5:   8b 3d f4 40 41 00       mov    0x4140f4,%edi
  402eeb:   33 db                   xor    %ebx,%ebx
  402eed:   88 45 ff                mov    %al,-0x1(%ebp)
  402ef0:   83 e6 07                and    $0x7,%esi
  // loop start (32 times)
  402ef3:   8a d0                   mov    %al,%dl
  402ef5:   8a cb                   mov    %bl,%cl
  402ef7:   d2 ea                   shr    %cl,%dl
  402ef9:   f6 c2 01                test   $0x1,%dl
  402efc:   74 0d                   je     0x402f0b
  402efe:   72 03                   jb     0x402f03             # basically
'continue'
  402f00:   73 01                   jae    0x402f03
   0x402f03:    mov    0x4140f4,%edi                            !!# disasm
   0x402f09:    jmp    0x402f37
   0x402f0b:    mov    0x416640(,%esi,4),%edx
   0x402f12:    mov    $0x1,%eax
  402f17:   03 d7                   add    %edi,%edx
  402f19:   8b cb                   mov    %ebx,%ecx
  402f1b:   33 d7                   xor    %edi,%edx
  402f1d:   d3 e0                   shl    %cl,%eax
  402f1f:   2b d7                   sub    %edi,%edx
  402f21:   f7 d2                   not    %edx
  402f23:   0b d0                   or     %eax,%edx
  402f25:   8a 45 ff                mov    -0x1(%ebp),%al
  402f28:   f7 d2                   not    %edx
  402f2a:   03 d7                   add    %edi,%edx
  402f2c:   33 d7                   xor    %edi,%edx
  402f2e:   2b d7                   sub    %edi,%edx
  402f30:   89 14 b5 40 66 41 00    mov    %edx,0x416640(,%esi,4)
  402f37:   43                      inc    %ebx
  402f38:   83 fb 20                cmp    $0x20,%ebx
  402f3b:   7c b6                   jl     0x402ef3
    loop end
  402f3d:   e9 98 01 00 00          jmp    0x4030da             # Move to next
instruction
```

Une autre instruction digne d’intérêt est l'instruction 0x03 qui essaye d'ouvrir le fichier C:\Rayman\Rayman.ini, et teste si le numéro du descripteur obtenu est 0x42!. Si c'est le cas, la pile est incémentée de 0x42 et l'instruction s'arrête là. Cela paraît très louche, voire franchement boggué. Mais le clin d’œil nous a plut. Voici le code en question:

```
  402c8e:   68 00 00 00 80          push   $0x80000000
```

```
  402c93:   68 c8 1f 41 00          push   $0x411fc8           # cst str @0x105c8:
'C:\Rayman\Rayman.ini'
  402c98:   ff 15 3c e0 40 00       call   *0x40e03c           # cst str @0xc63c:
'^N&^A'          # cst str @0xc63c: 'CreateFileA'
  402c9e:   83 f8 42                cmp    $0x42,%eax          # file handle == 66
??? should we open N files before to reach that number)
  402ca1:   75 04                   jne    0x402ca7
  402ca3:   83 c4 42                add    $0x42,%esp
  402ca6:   c3                      ret
```

En décodant le binaire nous avons aussi rencontré des fonctions très tordues qui se sont avérée n'etre que des implementaion loufoques de memcpy.

Tout d'abord précisons que certaines instructions décodées par objdump sont erronées et ont dû être décodées par gdb (commande 'x/i addr' par exemple). Les lignes 0x402f03 à 0x402f12 ci dessus ont été remplacées. Elles étaient à l'origine:

```
  402f02:   e8 8b 3d f4 40          call   0x41346c92
  402f07:   41                      inc    %ecx
  402f08:   00 eb                   add    %ch,%bl
  402f0a:   2c 8b                   sub    $0x8b,%al
  402f0c:   14 b5                   adc    $0xb5,%al
  402f0e:   40                      inc    %eax
  402f0f:   66 41                   inc    %cx
  402f11:   00 b8 01 00 00 00       add    %bh,0x1(%eax)
```

Ce qui n'a aucun sens (l'adresse 0x41346c92 est très en dehors de la section exécutable du programme par exemple).

L'execution d'un script implique une sorte d'état de machine virtuelle, des registres virtuels, etc. Voici comment ce présente celle ci. J’omets ici la véritable chronologie de nos découvertes, pour éviter que ce texte n'arrive à temps que comme solution du challenge SSTIC de l'année prochaine.

La mémoire virtuelle consiste en un espace d'adresses linéaire, mappés sur des blocs de mémoire réels non-contigus. D'une manière similaire au systèmes de paging des CPU modernes (bien que dans ces cas là, les pages aient des tailles fixes).
Voici ce que pourrait être le type d'un nœud en 'C':

```
struct _Noeud {
      uint32_t section_type; // seuls l'octet de poids faible semble utilisé
      uint32_t vmem_ptr;
      uint32_t size;
      uint32_t buf;
      void *   next: // pointeur de chaîne
};
```

Dans l'initialisation de cette machine virtuelle, trois blocs mémoires sont créés, dont nous offrons ici un apercu récupéré via le debogueur. Notons que les adresses des nœuds et par conséquent des pointeurs 'next' peuvent varier d'une exécution à l'autre, car ils sont alloués sur le tas via une appel à HeapAlloc(), le reste des champs ne varie pas: la type du bloc, sa taille et son vaddr.

```
00134ee8:
            0x00    00138985    u8 type?; (only lsb used)
            0x04    05571c00    vaddr
            0x08    00000400    int   size;
            0x0c    00134f08    void *buf;
```

```
            0x10    00135f10    Node* next;

00135f10:
            0x00    001101fe
            0x04    05572000
            0x08    00000100    int   size;
            0x0c    00138900    void *buf;
            0x10    00135f30    Node* next;

00135f30:
            0x00    00110161
            0x04    05572100
            0x08    00001100    int   size;
            0x0c    00139908    void *buf;              <-- passwd is copied here
at offset 0x1020
            0x10    ffffffff    Node* next;
```

Remarquons que les vaddrs sont contiguës alors que les tampons ne le sont pas. Le type du premier bloc est 0x85 (octet de poids faible), le second 0xfe et le troisième 0x61. La fin de la chaîne est signifié par un pointeur 'next' de valeur -1 (et non 0!).

La signification de ces blocs se fait plus claire au fur et à mesure que nous progressons dans la compréhension du code.
La première section de mémoire contient le script lui-même (liste d'opcodes). C'est dans cette section que l'interpréteur lit les opcodes à exécuter. La variable globale 0x416660 contient d'ailleurs la position courante dans du code (instruction pointer (IP) de la machine virtuelle). La fonction à 0x402210 par exemple, va lire un entier de 32bits à la position courante de 0x416660 et augmente 0x416660 de 4 unités.

La deuxième section contient la pile: certaines instruction du script empilent ou dépilent des données sur cette pile. La variable globale 0x41665c contient le pointeur vers le haut de la pile.

La troisième section contient le tas. C'est à un certain endroit de cette section que le mot-de-passe fournis par l'utilisateur est copié puis passé à la moulinette du script.

A noter encore que la machine de script contient un nombre de registres de 32bits aux adresses: 0x416640 et suivantes. Et d'ailleurs les pointeur d'instruction et de haut de pile peuvent être vus comme des registres spéciaux (d'index 8 et 7 respectivement).

Le pointeur vers le premier nœud de cette liste est stocké à l'adresse 0x416638 (que nous appelons ll_root, pour linkedlist root node).

La fonction à 0x402380 réinitialise la machine de script.

NOTE: les pointeurs de cette machine de script ne sont jamais stockés en clair mais sont brouillés d'une façon que l'on retrouve partout: add, xor, sub avec une constante.

Nous avons reimplémenté en python l'interpréteur de ce langage de script (voir 3_interp.py). Les noms de fonctions utilisés dans ce programme sont révélateurs de notre compréhension imcomplète à cet instant.

Notons encore que les valeurs lues de la section code ne sont pas des opcodes directement, mais des indices dans la table des opcodes qui se trouve à 0x403168

qui nous reproduisons ici (les adresses, colonne de gauche, sont invalides):

```
0000000: 0011 1111 1111 1111 1111 1111 1111 1111  ................
0000010: 1111 1111 1111 1111 0111 1111 1111 1111  ................
0000020: 1111 1111 1111 1111 0211 1111 1111 1111  ................
0000030: 0311 1111 1111 1111 1111 1111 1111 1111  ................
0000040: 1111 1111 1111 1111 0411 1111 1111 1111  ................
0000050: 0511 1111 1111 1111 0611 1111 1111 1111  ................
0000060: 1111 1111 1111 1111 0711 1111 1111 1111  ................
0000070: 1111 1111 1111 1111 0811 1111 1111 1111  ................
0000080: 0911 1111 1111 1111 1111 1111 1111 1111  ................
0000090: 0a11 1111 1111 1111 1111 1111 1111 1111  ................
00000a0: 0b11 1111 1111 1111 0c11 1111 1111 1111  ................
00000b0: 0d11 1111 1111 1111 1111 1111 1111 1111  ................
00000c0: 1111 1111 1111 1111 0e11 1111 1111 1111  ................
00000d0: 1111 1111 1111 1111 1111 1111 1111 1111  ................
00000e0: 0f11 1111 1111 1111 10                   .........
```

La mémoire des sections a pu être sauvegarder par la commande 'gdb dump', par exemple ceci sauve dans le fichier 'node0x85.bin' 1KO mémoire depuis 0x00134f08:
dump binary memory node0x85.bin 0x00134f08 0x00134f08+0x400

Notre implémentation python nous permet de traduire le script en un langage plus adaptée à notre nature charnelle et neuronale. Nous obtenons un fichier de 212 lignes (voir 3_trace.txt) qui nous montre quelles ont été les actions du script.

En voici les quatre premières lignes:

```
      1 ASM 0x05571c00:   scr0 = 0x5572100
      1 ASM 0x05571c05:   scr0 = scr0 + 0x1000
      1 ASM 0x05571c0a:   call 0x05571c0e
      1 ASM 0x05571c0c:   jmp 0x05571cef
```

La variable scr0 est assignée la valeur 0x5573100 que l'on reconnaît comme une valeur de vaddr dans la section du tas (3ème node de la liste chaînée).

L'appel de fonction, à la troisième ligne, initialise 32 octets de mémoire de 0x5573100 à 0x5573120 en calculant un hash de valeurs constantes. Nous n'irons pas dans les détails ici, les valeurs étant constantes nous les sauvegardons depuis gdb et passons à la suite, le saut à 0x05571cef.

Interpréteur de script en watermark

Les quelques vingt premières instructions sont parlantes:

```
   8040 ASM 0x05571cef:   scr0 = 0x3
   8040 ASM 0x05571cf4:   load #scr0 pixel watermarks into scr1 -> scr[ scr0 ]
   8040 ASM 0x05571cf5:   if scr1 == 0x78: jump 0x5571d3b
   7507 ASM 0x05571cfa:   if scr1 == 0xfe: jump 0x5571d4b
   5435 ASM 0x05571cff:   if scr1 == 0x33: jump 0x5571d69
   5170 ASM 0x05571d04:   if scr1 == 0x12: jump 0x5571d9b
   4910 ASM 0x05571d09:   if scr1 == 0xa1: jump 0x5571da6
   4645 ASM 0x05571d0e:   if scr1 == 0xaf: jump 0x5571dc7
   2577 ASM 0x05571d13:   if scr1 == 0x8e: jump 0x5571de9
   2577 ASM 0x05571d18:   if scr1 == 0x13: jump 0x5571e0b
   2065 ASM 0x05571d1d:   if scr1 == 0xbb: jump 0x5571e2d
   2057 ASM 0x05571d22:   if scr1 == 0x7c: jump 0x5571ed9
```

```
   1801 ASM 0x05571d27:   if scr1 == 0x32: jump 0x5571e77
   1289 ASM 0x05571d2c:   if scr1 == 0x2f: jump 0x5571ea7
    777 ASM 0x05571d31:   if scr1 == 0xdc: jump 0x5571ef8
    776 ASM 0x05571d36:   if scr1 == 0x59: jump 0x5571e52
```

Il s'agit de la boucle de dispatch d'un interpréteur de script (un nouveau), qui prends ses instructions d'une manière un peu particulière.
L'instruction à 0x05571cf4 est une instruction qui va chercher 'scr0' valeurs d'une fonction que nous avons appelé GetNextPixelWatermark() dans 3_interp.py. Nous avons implémenté GetPixel() en python, ayant préalablement sauvegardé l'image dans un fichier (3_img_678x306_24bpp_BGR.data que l'on peut voir dans gimp). L'image est en fait l'image de fond du dialogue.

```
def GetNextPixelWatermark():
    global bm_0x41663c
    code_0x4140f4 = 0xfde7f446

    y = bm_0x41663c
    bm_0x41663c += 1
    x = code_0x4140f4 ^ y
    x += 0x42
    x &= 0xff
    i = 0
    h = 306
    bl = 0
    edi = x % 3
    while i < 8:
        RGB = GetPixel(x, y)
        bl <<= 1
        bl |= (RGB>>((8*edi)&0xff)) & 1
        edi = (edi + 1) % 3
        x += 0x2a
        i += 1
    #print("GetNextPixelWatermark() returns 0x%x" % bl)
    return bl
```

Cette fonction utilise l'API Windows GetPixel(x,y) pour obtenir la valeur RGB du pixel à la position (x,y). Huit pixels sont ainsi cherchés et chacun est utilisé pour dériver un bit de la réponse de la fonction.

Les huit bits ainsi obtenus forment un nombre entre 0 et 255 inclus. Le script demande trois de ces valeurs qui sont placés dans les registres scr1, scr2 et scr3. Le début de l’interpréteur utilise scr1 comme opcode de ce nouveau langage de script, il n'y a que 14 instructions différentes, tous les autres numéros d'instruction exécutent par défaut le même code que l'instruction 0x78.

Notons au passage qu'en allant chercher le code de tous les pixels de l'image, nous trouvons un autre LUM: LUM{C1UAidv_pzJ}

Nous avons sauvegardé le code de ce script dans 3_watermark.bin, en voici les premières lignes:

```
$ xxd -c 3 -g 1 3_watermark.bin
0000000: 78 1c 00  x..
0000003: 33 1c 04  3..
0000006: a1 9c 00  ...
0000009: fe 04 1c  ...
000000c: af 04 04  ...
000000f: bb 56 47  .VG
```

Nous avons ensuite coder un parseur de ce langage de script vers une syntaxe similaire au 'C' pour mieux le comprendre et pour ensuite l'implémenter en python pour pourvoir faire des expériences. Voir 3_waterdis.py. Les six premières lignes montrées ci-dessus sont traduites ci-dessous (la totalité est dans 3_waterdis.txt):

```
(pix_y   0) (OP 0x78):
mem[0x40+0x1c] = 0

(pix_y   3) (OP 0x33):
mem[0x70] = 1
if mem[0x40+0x1c] != 4
    mem[0x70] = 0

(pix_y   6) (OP 0xa1):
if mem[0x70] != 0:
    scr0 = 156 + (0<<8)
    pix_y = 6 + 3 + 156 = 165

(pix_y   9) (OP 0xfe):
mem[0x40+0x4] = mem[0x40+0x1c]

(pix_y  12) (OP 0xaf):
mem[0x40+0x4] += mem[0x40+0x4]

(pix_y  15) (OP 0xbb):
tmp1 = mem[0x44] << 2
tmp2 = mem[0x20 + tmp1]
mem[0x40] = tmp2
```

Finalement, nous traduisons à la main les 212 instructions en 'C', voir la methode scramble_passwd() dans 3_waterscript.c. Nous la reproduisons en simplifié ici en pseudo code. La memoire 'mem' ici est en fait la tas du premier script à l'offset 0x1000 (le troisème nœud de la liste chaînée). Rappelons ici que les 32 caractères du mot-de-passe entré par l'utilisateur ont été copiés aux adresses tas+0x1020 et suivantes. Ils sont donc à mem + 0x20 ici.

```
   *mem(0x0) = 0x9aacc69b;
 // Constantes recupérées grâce à gdb
   *mem(0x4) = 0x89a31d8a;
   *mem(0x8) = 0xa5b3cd5c;
   *mem(0xc) = 0xb0a5ce54;

   int i, j;
   for(i=0; i<4; i++)
   {
        mem_4c = mem(0x20 + i*2*4);
        mem_50 = mem(0x20 + (i*2+1)*4);

        u32 mem_54 = 0;
        for(j=0; j<64; j++) {
            mem_4c += (mem_50 + scramble(mem_50, 4))
                    ^ (mem_54 + mem_cst[mem_54 & 3]);

            mem_54 += mem_0xf4b + 2;

            mem_50 += (mem_4c + scramble(mem_4c, 4))
```

```
                    ^ (mem_54 + mem_cst[(mem_54>>11) & 3]);
        }

        *mem(0x20 + i*2*4) = mem_4c;
        *mem(0x20 + (i*2+1)*4) = mem_50;
   }

   sleep(3000);
```

La fonction scramble est:

```
static inline u32 scramble( u32 v, u32 sh ) {
    u32 v1 = v << (sh&0xff);
    u32 v2 = v >> ((sh&0xff)+1);
    return (v1 ^ v2) & 0xffffffff;
}
```

Voici donc la façon dont le mot-de-passe est utilisé. Il est à noter que le mot-de-passe est traité en quatre blocs de 8 octets indépendants. Il apparaît que la fonction ci-dessus qui mixe le mot-de-passe est complètement inversible. C'est-a-dire que connaissant la valeur de mem(0x20 + i*2*4) et mem(0x20 + (i*2+1)*4) (quelque soit le i), nous pouvons, de proche en proche, remonter toutes les valeurs des 64 itérations de la boucle jusqu’à la valeur initiale. Cette inversion est implémentée par la fonction “static int unscramble_passwd()”.
Notons aussi que ces emplacements mémoires sont précisément ceux qui sont utilisés pour la validation du mot-de-passe en tout dernier lieu.

En utilisant le débogueur, nous vérifions que nous avons une implémentation correcte de la fonction et de son inverse. Il ne reste plus en théorie qu'a inverser la valeur de vérification et nous aurons le mot-de-passe.

Malheureusement, c'est un échec douloureux, qui résulte en un maux de tête bien plus sûrement que sur un mot-de-passe.

En utilisant notre ami 'gdb' (nous sommes passés au tutoiement depuis bien longtemps), l'on se rends compte que notre valeur brouillée du mot-de-passe est modifiée par la fonction Sleep (notez le Sleep(3000) à la toute fin de la routine de brouillage). Grosse surprise et incompréhension. La fonction Sleep() a peut-être été modifié? Est-ce un piège?
Le mot de l'enigme nous est donné en regardant de plus prés l'appel à Sleep(). Notez l'affectation de la valeur -1 à la variable globale 0x41663c.

```
  4030fc:   68 b8 0b 00 00          push   $0xbb8          # = 3000
  403101:   c7 05 3c 66 41 00 ff    movl   $0xffffffff,0x41663c
  403108:   ff ff ff
  40310b:   ff 15 1c e0 40 00       call   *0x40e01c            # cst str @0xc61c:
'z%^A'           # cst str @0xc61c: 'Sleep'
```

Il se trouve qu'un autre fil d'execution (une thread) attends patiemment que la valeur de cette même variable passe à -1 pour entrer en action. La thread principale suppose que pendant ces 3 secondes de sommeil la thread de l'ombre aura commis son méfait.

Cryptage du mot-de-passe 2 (Sleep thread)

Le méfait nous entraîne dans un univers plus étrange encore, que l'on pourrait appeler l'unirop (nous vous avions parlé de boite de Pandore, n'est-ce-pas?)

Voici le code de la thread dormante, le cauchemard qui prend forme quand la thread principale s'endort.

```
  401370:   56                      push   %esi
  401371:   57                      push   %edi
  401372:   8b 3d 1c e0 40 00       mov    0x40e01c,%edi            # cst str
@0xc61c: 'z%^A'           # cst str @0xc61c: 'Sleep'
  401378:   eb 06                   jmp    0x401380
  40137a:   8d 9b 00 00 00 00       lea    0x0(%ebx),%ebx
  401380:   83 3d 3c 66 41 00 ff    cmpl   $0xffffffff,0x41663c
  401387:   75 49                   jne    0x4013d2
  401389:   be 00 41 41 00          mov    $0x414100,%esi
  40138e:   8b ff                   mov    %edi,%edi
  401390:   e8 3b 09 00 00          call   0x401cd0             # purefunc returns
0xf4b02f4b
  401395:   31 06                   xor    %eax,(%esi)
  401397:   83 c6 04                add    $0x4,%esi
  40139a:   81 fe 40 42 41 00       cmp    $0x414240,%esi
  4013a0:   7c ee                   jl     0x401390
  4013a2:   68 00 41 41 00          push   $0x414100
  4013a7:   e8 b4 ff ff ff          call   0x401360
  4013ac:   83 c4 04                add    $0x4,%esp
  4013af:   be 00 41 41 00          mov    $0x414100,%esi
  4013b4:   ff 05 3c 66 41 00       incl   0x41663c
  4013ba:   8d 9b 00 00 00 00       lea    0x0(%ebx),%ebx
  4013c0:   e8 0b 09 00 00          call   0x401cd0             # purefunc returns
0xf4b02f4b
  4013c5:   31 06                   xor    %eax,(%esi)
  4013c7:   83 c6 04                add    $0x4,%esi
  4013ca:   81 fe 40 42 41 00       cmp    $0x414240,%esi
  4013d0:   7c ee                   jl     0x4013c0
  4013d2:   6a 64                   push   $0x64
  4013d4:   ff d7                   call   *%edi
  4013d6:   eb a8                   jmp    0x401380
```

Remarquez l'instruction “cmpl $0xffffffff,0x41663c ” qui teste cette variable globale, puis dort un peut (push $0x64; call   *%edi).

Comment une fonction si petite peu causer tant de douleur? Et bien ce n'est que la partie visible de l'iceberg. L'appel à 0x401360 est plus petit encore:

```
  401360:   55                      push   %ebp
  401361:   8b ec                   mov    %esp,%ebp
  401363:   8b 65 08                mov    0x8(%ebp),%esp
  401366:   c3                      ret
```

Quatre petites instructions de malheur qui ne font strictement rien! …. à première vue. En fait cette fonction manipule le registre esp de manière vicieuse et fait en sorte que le 'ret' ne retourne pas au site d'appel mais aille dans un endroit bien différent. C'est la méthode classique des crackeurs qui profittent d'un dépassement de tampon pour changer l'adresse de retour d'une fonction et de retour et retour executent un code de leur choix (une ROP chaîne en anglais ROP = Return Oriented Programming). La seule différence ici est qu'il n'y a pas de débordement de tampon mais une intention première et, n'ayons pas peur des mots, sadique des organisateurs du concours.

Esp est donc assigné la valeur passée en paramètre, qui est 0x414100 (regarder le code asm juste avant le call, on à “push $0x414100”). L'instruction 'ret' lis la valeur 32bits (ou 64bits) à l'adresse pointée par esp, ajoute 4 à esp, puis met le

registre eip à cette adresse.
Pour connaître la valeur qui se trouve à 0x414100 (section .data), il est necessaire de savoir ce qui se trouve en memoire pour cela nous utilisons gdb pour faire un dump memoire dans un fichier. Nous choisissons de sauvegarder les adresses 0x413000 à 0x413000+0x2800 (les valeurs memoires de la section .data). Voir 3_mem_datasection.bin

L'adresse 0x414100 est à l'offset 0x1100 de ce fichier (0x414100-ox413000 = 0x1100)

```
0001100: d0104000  ..@.
0001104: 8d1c4000  ..@.
0001108: d0104000  ..@.
000110c: 33000000  3...
```

L'adresse lue en little-endian est donc eip=0x4010d0. Et esp=0x414104 le code à cette adresse est:

```
4010d0:   c3                    ret
```

ce ret met eip=0x401c8d et esp= 0x414108. Le code à 0x401c8d est (selon gdb, car objdump n'a pas decompilé au bon endroit):

```
0x00401c8d: cb lret
```

L'instruction 'lret ' est un far return: elle lis l'adresse de retour à l'adresse esp dans eip, et le cs à l'adresse esp+4, ajoute 8 à esp. Si le cs lu vaut 0x33, le processeur passe en mode 64bits, s'il est 0x23 le processeur passe en mode 32 bits.

Nous donc ici eip=0x4010d0 et cs=0x33. Nous revisitons donc l'adresse 0x4010d0, mais cette fois en mode 64bits. Cette subtilité nous a causé une certaine confusion.

En continuant à suivre le fil, suivant les valeurs de la memoire, les changements de eip, de esp et de cs, nous arrivons à retracer le fonctionnement du programme, que nous résumons ici (notez au passage la LUM qui fait son apparition).

Nous avons utilisé le site https://defuse.ca/online-x86-assembler.htm#disassembly2 pour désassembler le code 64bits, les annotations sont nos notes de travail.

Attention, la convention sur l'ordre des registres change par rapport à objdump et gdb. Ici xor r8,rcx se lit r8 = r8 XOR rcx.

```
0:  5a                      pop    edx    ← LUM{+zhV
1:  59                      pop    ecx    ← QqJy03q}
0:  5a                      pop    rdx    ← 0x756af83de1ffe263
1:  59                      pop    rcx    <-0xfa34ae1002547b89
0:  48 f7 d2                not    rdx    <-
0:  48 0f c9                bswap  rcx    <- 0x897b540210ae34fa
20: 58                      pop    rax  <- 0x4140c0
40: 8b 00                   mov    eax,DWORD PTR [rax]  <- 0x13a928 (addr of
scrambled passwd)
50: 4c 8b 00                mov    r8,QWORD PTR [rax]   <- 8bytes of scrambled
passwd
60: 49 31 c8                xor    r8,rcx
70: 49 01 d0                add    r8,rdx
80: 4c 89 00                mov    QWORD PTR [rax],r8
90: 48 83 c0 08             add    rax,0x8
a0: 4c 8b 00                mov    r8,QWORD PTR [rax]
```

```
b0: 49 31 d0                xor    r8,rdx
c0: 49 0f c8                bswap  r8
d0: 49 29 c8                sub    r8,rcx
e0: 4c 89 00                mov    QWORD PTR [rax],r8
150:    48 83 c0 08             add    rax,0x8
160:    4c 8b 00                mov    r8,QWORD PTR [rax]
170:    49 31 c8                xor    r8,rcx
180:    48 c1 ca 15             ror    rdx,0x15
190:    48 87 d1                xchg   rcx,rdx
1a0:    49 d3 c8                ror    r8,cl
1b0:    49 01 d0                add    r8,rdx
1c0:    4c 89 00                mov    QWORD PTR [rax],r8
1d0:    cb                      retf   cs=0x23 back to 32bit mode
```

Voila donc les operations que la chaîne de ROP execute pendant que la thread principale est endormie pour 3 secondes. On note que le resultat de la thread principale (brouillage du mot-de-passe) est poussé plus avant avec un brouillage supplémentaire. On note aussi que ce brouillage est lui aussi parfaitement réversible.

La fonction scramble_passwd() dans 3_waterscript.c, mentionnée plus haut est notre implementation en python de ces deux brouillages. La fonction unscramble_passwd() implemente le débrouillage inverse.

Nous vérifions la validité des ces fonctions en générant des mots-de-passe aléatoires et en nous assurant que le brouillage + débrouillage nous rend le mot-de-passe d'origine. Nous vérifions aussi que le brouillage de chaque mot-de-passe de 32 octets que l'on peut imaginer confirme le brouillage calculé par le programme exe (en utilisant gdb pour inspecter la mémoire).

Protection anti-deboggage

Étant donné que d'une part nous avons implémenté une fonction de débrouillage correcte (de toute évidence), et que d'autre part nous savons quels sont les octets du mot-de-passe brouillé (ceux utilisé dans le programme pour validation), trouver le mot-de-passe est un jeu d'enfant: il suffit de débrouiller les octets de validation.

Nous effectuons cette procédure et obtenons un mot-de-passe. Malheureusement, à notre grande surprise celui-ci contient des caracteres non-ascii ou non imprimable. Clairement des caractères que l'on ne peux pas taper au clavier dans le champ de l'interface utilisateur.
Pourtant, le brouillage du mot-de-passe obtenu donne les octets de validation corrects.

Après maintes vaines recherches, nous arrivons à l'idée que peut-être le programme ne se comporte pas de la même manière quand il tourne sous un deboggeur. Pour le verifier, nous décidons de lancer le programme sans debogueur (wine unstable-machines.exe) d'entrer un mot-de-passe et d'attacher le debogueur au programme pendant le sleep de 3 secondes (commande “gdb -p PID”). Ce faisant, en utilisant un mot-de-passe simple, on remaque que le programme en effet ne brouille plus le mot-de-passe de la meme manière: nous sommes sur la bonne piste.

En inspectant la mémoire avec gdb, nous trouvons que certaines constantes utilisées dans le brouillage sont différentes (voire les 4 premieres lignes de notre implementation en 'C' plus haut).
La memoire à l'adresse de tas script 0x1008 est differente, elle est maintenant (sans debogueur au lancement du programme):

```
   *mem(0x8) = 0xa5b2cd1c,
```

En changeant cette constante, nous pouvons inverser les octets de validation et chaque caratere du mot-de-passe ainsi trouvé est un caractere possible à entrer au clavier et mieux encore un nibble hexadecimal ([0-9a-z]). Le mot-de-passe trouvé est donc possiblement une clef ce que nous verifions dans la machine virtuelle linux.
passwd = "3f691f3d6eb60b343c931c22e0baa92f"

En entrant ce mot-de-passe dans le champ de l'interface utilisateur, nous sommes gratifiés par un dialogue de félicitations. Le grand protoon m'a pris beacoup d'heure de sommeil et surement beaucoup d'électrons.

Nous pensions en avoir fini avec les maux de têtes et autres tirages de cheveux, mais non, un message nous indique qu'une nouvelle épreuve est débloquée: /challenges/final.txt

Pour finir, nous avons remarqué qu'une autre image existe dans l'exe. Il serait intéressant de prendre un watermark de ses pixels pour voir si un autre LUM ne s'y cache pas, mais nous n'en avons pas le temps.

6) LabyQRynth

Le fichier /challenges/final.txt contient ceci:

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
```

L'on a peu de mal à reconnaître les motifs habituels d'un Qrcode. Nous décidons donc de transformer ce texte en image en replacant les espaces par un pixel blanc et les caractères hexa par un pixel noir. Voici le resultat:

En utilisant un telephone de type Android et une app pour lire les Qrcodes, nous obtenons ce texte: “Please use this Nibble ADD key: 5571C2017”. Un nibble est un nombre sur 4 bits qui se code en général par un caractere hexa [0-9a-z]. les caractéres du fichier sont des nibbles, et nous interpretons cette prière comme suit:
Utilisez les neuf nibbles 5571C2017 en les ajoutant un par un aux nibbles du Qrcode (modulo 16) pour debrouiller la réponse.

Avant de continuer, notons qu'il s'agit d'un Qrcode de version 4 (dimension 33x33), avec niveau de correction d'erreur M (niveau 2 sur 4).

Il existe plusieurs facons d'arranger ces nibbles du texte en chaîne. Nous essayons succéssivement:

- prendre les nibbles des 4 lignes marquées (“deQRypt(“ → “);”)
- prendre les nibbles de tout le code QR
- prendre les nibbles des 4 lignes marquées en appliquant le “masque” QR (NOT)
- idem avec tous les nibbles du QR
- arranger les nibbles dans l'ordre de lecture d'un code QR (en partant du coin bas-droit, en remontant en zigzag, en entremélant les deux blocs de données.
- Reparer les erreurs introduites dans le Qrcode en utilisant la librairie python reedsolo (implementation de reed-solomon), et en recréant le QRCode sans erreurs, mais cela ne mène nulle part, il y a trop d'inconnues.

Voici le Qrcode nettoyé de ses erreurs de transmission:

En vain, aucune de ces martingales ne mène au succès.

Nous avons inplémenté une bonne partie d'un decodeur Qrcode, mais en vain. Une recherche internet (duckduckgo.com) nous indique que peut-etre 'labyrinth' fait reference à une methode de chiffrement.

Une autre possibilité est un simple labyrinth (https://cp4space.wordpress.com/2012/09/25/cipher-2-labyrinth/). Il semble en effet qu'il n'existe qu'un seul chemin qui mène de l'entrée, marquée par la chaîne “deQRypt(” à la sortie “);”. En suivant ce chemin, la chaîne des nibbles est:

ace80e6fcca1776558babe2c51d6b657658a8a2cf973d1bb5c938ac9da252fd4c973

Nous appliquons la clef comme indiqué, mais en utilisant '-' au lieu de '+' (on suppose que '+' était utilisé lors du brouillage, que la suggestion était imprécise, voire malveillante). En appliquant la clef "5571C2017" (nibble - clef[i]) % 16; i++ , l'on obtient:

57774c6e575a6b4541654d6a50616f4b457335bb37726c644073737469632e6f7267
soit en appliquant (binascii.unhexlify()):

WwLnWZkEAeMjPaoKEs5\xbb7rld@sstic.org

Nous sommes sur la bonne voie, pas de doute c'est une adresse email finissant en “@sstic.org”!
Maheureusement cette adresse contient un caractere non-imprimable (\xbb). En observant de plus près le chemin suivi, on note qu'il y a certains endroits plusieurs façons de suivre les cases (le point rouge ci dessous est le premier chemin emprunté menant au caratère \xbb).

En essayant une nouvelle façon, nous trouvons:
ace80e6fcca1776558babe2c51d6b657658a8abcf973d1bb5c938ac9da252fd4c973
57774c6e575a6b4541654d6a50616f4b4573354b37726c644073737469632e6f7267

WwLnWZkEAeMjPaoKEs5K7rld@sstic.org

7) Remarques générales et personnelles

L'introduction des LUM et des clefs intermédiaires est très bénéfique pour le candidat car elles lui permettent de mesurer son avancement sans nécessairement finir le concours en trouvant l'adresse courriel finale.

Le challenge complet m'a semblé trop long, un challenge plus court serait plus abordable. Je pense que la sélection, l'élimination progressive des candidats est à attribuer à la longeur plus qu'à la difficulté de l'épreuve.

L'étape riscy_zones est bien conçue, avec un fichier de test 'password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted' qui permet de se familiariser avec l'exécutable 'trustzone_decrypt' et donne les données nécessaires pour tester nos ré-implémentation en python/C. Cet aspect est bien appréciable.

L'étape 'unstable_machines.exe' que j'ai abordé sans conviction, étant moins intéressé par un code x86/Windows a été intéressante mais longue. Je pense que l'impression de longueur viens du fait qu'une nouvelle embûche se dressait alors que l'on pensait que le but était atteint (la thread de l'ombre, la technique anti débogueur, …) En outre, les deux parties sont de tailles très différentes. Après coup, on peux voir la partie des cases-à-cocher comme une mise en jambe ce qui n'est pas opposable car l'assembleur et le système sont différents.

La dernière étape (QRCode), bien que courte au final, m'a semblé plus frustrante que les autres. En introspectant, je vois plusieurs raisons: premièrement, je pensais que "unstable_machines" était la dernière étape, j'ai donc du me remotiver pour aller jusqu'au bout. Deuxièmement, l'idée de regarder l'image comme un labyrinthe avec une entrée et une sortie me semble très peu évidente (et ce malgré le troisième erratum). Pour cette raison, je suis parti dans plusieurs mauvaises directions (recouvrer le QRCode sans erreurs, aligner les nibbles dans le sens de lecture naturel du QR, appliquer le masque QR aux nibbles, …) qui m'ont pris du temps et de l'effort. J'aurais préférer soit une indication plus claire du labyrinthe, soit une solution nécessitant l'écriture d'un décodeur QR partiel. Pour le coté positif, je connais maintenant bien mieux comment est construit un code QR et un peu mieux les corrections d'erreur Reed-Solomon.

Pour l'année prochaine, il serait interessant de devoir désassembler un véritable trojan ou bien devoir profiter d'une veritable faille de sécurité (dans une MV quand même).

Liste des sujets abordés:
- openrisc, Linux vm en js, QRcode, Reed-Solomon, TEE/TA, BPF, NAND, signaux électroniques, interpréteur de scripts (2 ou 3), ROP, AES.
- environ 2000 lignes de code python
- environ 2000 lignes de code 'C'