Challenge SSTIC 2016

David BERARD
contact@davidberard.fr
@polymorf34

Résumé

Ce document présente la démarche choisie par l'auteur pour résoudre le challenge SSTIC 2016. Comme tous les ans, le but de ce challenge est de retrouver une adresse mèl cachée dans le fichier fourni comme énoncé. Cette année le challenge comporte 3 niveaux de 3 ou 4 épreuves chacun. Les épreuves sont accessibles au travers d'un mini jeu RPG réalisé en JavaScript. Pour valider chaque niveau, il faut résoudre deux épreuves à 1 point, ou une épreuve à deux points.
Le challenge débute avec une capture réseau disponible sur le wiki de SSTIC.
Ce challenge est constitué de :

— 8 épreuves de rétro-ingénierie logicielle allant du script pour calculatrice, à l'architecture ia-64
— 2 épreuves d'analyse réseau : GSM et analyse d'échanges d'un CNC avec son client

Les résolutions de 9 des 10 épreuves seront présentées dans ce document, l'épreuve du dernier niveau "ring" ayant résisté à l'auteur.

Ce fichier PDF est également un fichier ZIP, contenant l'ensemble des scripts utilisés pour résoudre les épreuves.

Table des matières

1 Découverte du challenge

1.1 Trace réseau

La trace réseau fournie en énoncé du challenge contient un simple échange HTTP :

```
$ tcpflow -r challenge.pcap -o flows

$ cat flows/010.069.016.064.40586-195.154.171.095.00080
GET /challenge2016/challenge.zip HTTP/1.0
Host: static.sstic.org
User-Agent: Mozilla/5.0 (Wayland; BTTF; Linux x86_128; rv:142.0) Gecko/20100101
    Firefox/142.0
Accept: *

$ head -n 4 flows/195.154.171.095.00080-010.069.016.064.40586
HTTP/1.0 200 OK
Server: CERN/3.0A
Content-type: application/zip
Content-Length: 52331069
```

Listing 1: extraction des flux TCP

On notera le User-Agent exotique, qui aurait été difficile à attaquer en force brute dans le challenge de l'an dernier.

```
$ dd if=flows/195.154.171.095.00080-010.069.016.064.40586 of=challenge.zip bs=95 skip=1
$ unzip challenge.zip -d ../game
Archive:  challenge.zip
  inflating: ../game/rpgjs/core/scene/Scene_Gameover.js
  [...]
  inflating: ../game/plugins/Bonus/Sprite_Bonus.js
```

Listing 2: Extraction de l'archive

L'archive contient un mini jeu RPG en JavaScript.

1.2 Mini jeu RPG

Le jeu extrait de la trace réseau, est un petit jeu RPG développé en JavaScript à l'aide de la bibliothèque RPGJS.

FIGURE 1: Début du jeu

Ce jeu permet d'obtenir les épreuves du challenge en discutant avec certains personnages, d'autres personnages sont présents uniquement pour raconter des "trolls" (voir la section personnages de ce document : 7.2). Chaque niveau est un plateau de jeu, pour passer au plateau suivant un garde demande les résultats des épreuves.
Pour passer au plateau suivant il faut donner 2 clefs au garde, certaines épreuves valent double et permettent de passer directement au niveau suivant.
Chaque clef obtenue permet de déchiffrer un coffre de clef en AES-CBC-128 (l'IV et les données sont stockées dans un dictionnaire JSON). Les coffres de clefs sont indexés dans le dictionnaire JSON par le SHA256 de la clef permettant

de les déverrouiller, ainsi la validité d'une clef est aisément vérifiable.
Le coffre de clef contient une ou deux clefs. En combinant deux clefs issues des coffres, on obtient la clef AES permettant de déchiffrer le dictionnaire JSON du niveau suivant.
Une implémentation en Python est disponible en annexe (`sstic-crypto.py`), les nostalgiques de la version JavaScript apprécieront les options `--enable-slow-crypto` et `--disallow-copy-paste`.

```
$ python2 sstic-crypto.py --dir extract --sham-data plugins/sham-data.js
[*] Extracting files (level 1)
[*]     extract/calc.zip
[*]     extract/SOS-Fant0me.zip
[*]     extract/radio.zip
[*] You have entered 0/2 keys
Enter your key: 1ac3d8c409e656380a06f6f2c6de6b4a
[+] Your key is valid ! :)
[+]     Next level AES key = 66fa047b1f4acae175aa6d85d793f63c
[+] Next LEVEL !
[*] Extracting files (level 2)
[*]     extract/loader.zip
[*]     extract/success.txt
[*]     extract/foo.zip
[*]     extract/huge.zip
[*] You have entered 0/2 keys
Enter your key: E574B514667F6AB2D83047BB871A54F5
[+] Your key is valid ! :)
[*] You have entered 1/2 keys
Enter your key: 347d8c72720d6ec7a501583be0bccc0c
[+] Your key is valid ! :)
[+]     Next level AES key = c467cb6f7f04da43b025aa07d3171cf8
[+] Next LEVEL !
[*] Extracting files (level 3)
[*]     extract/ring.zip
[*]     extract/video.zip
[*]     extract/strange.zip
[*]     extract/success.txt
[*]     extract/usb.zip
[*] You have entered 0/2 keys
Enter your key: 2342503847250828733577208554403 5
[+] Your key is valid ! :)
[+]     Next level AES key = 110d3a0ef73ccb908758c966d50e7b7c
[+] Next LEVEL !
[*] Extracting files (level 4)
[*]     extract/final.txt
```

Listing 3: Déchiffrement des niveaux avec l'implémentation Python

2 Niveau 1

2.1 Calc : Rétro-ingénierie d'un programme TI-Basic pour Texas Instruments TI-83+

Quand j'étais encore jeune, j'ai caché ma clé dans un instrument que j'ai ramené du Texas.
Malheureusement, j'ai oublié comment la récupérer depuis...

L'archive `calc.zip` fournie par un personnage contient le fichier `SSTIC16.8xp`. Ce fichier est un programme pour la calculatrice scientifique Texas Instrument TI-83+.

```
$ bsdtar -xvf ../calc.zip
x SSTIC16.8xp
$ file SSTIC16.8xp
SSTIC16.8xp: TI-83+ Graphing Calculator (program)
```

Listing 4: Extraction de l'archive de l'épreuve

Quelques chaînes sont présentes dans le fichier, mais rien ne permet d'en comprendre le fonctionnement. Pour aller plus loin, le format 8xp doit être décompilé pour obtenir un fichier lisible.
Plusieurs projets sur Github permettent de décompiler ce type de fichier. Le projet parse8xp a été utilisé pour décompiler le fichier :

```
$ git clone https:/github.com/Lekensteyn/parse8xp.git
$ python2 parse8xp/main.py SSTIC16.8xp SSTIC16.txt
Loading SSTIC16...
Program is protected
 =====  SSTIC 2016 ======================
SSTIC16.8xp successfully decompiled as SSTIC16.txt
```

Listing 5: Décompilation du fichier 8xp

Cette décompilation permet d'obtenir le code TI-Basic du programme, donc voici un extrait :

```
...
Lbl 0
If S=5:Goto 5
If S=6:Goto 6
If S=8:Goto 8
If S=9:Goto 9
If S=10:Goto 10
If S=11:Goto 11
If S=13:Goto 13
If S=14:Goto 14
If S=16:Goto 16
If S=17:Goto 17
If S=18:Goto 18
If S=19:Goto 19
If S=21:Goto 21
If S=23:Goto 23
Disp "DISPATCH ERROR"
Stop

Lbl 3
sub(Str1,length(Str1)-((A+1)*8)+1,8)->Str1
Goto 0

Lbl 2
" "->Str1
Repeat not(A
A/2->A
sub("01",1+not(not(fPart(Ans))),1)+Str1->Str1
iPart(A->A
End
sub(Str1,1,length(Str1)-1)->Str1
While length(Str1)<32
"0"+Str1->Str1
End
```

```
Goto 0
...
```

Listing 6: Extrait du code TI-Basic

Ce code contient des "labels" de manipulation de chaînes pour des représentations binaires (décodage, encodage), et des fonctions de calcul.
Le code est traduit en Python pour une analyse plus poussée, voici le même extrait que précédemment traduit en Python (le code complet est disponible en annexe : `emu.py`) :

```
# ...
def Label0():
  global A,B,C,D,S,N,L1,Z,X,Y
  global Str1,Str2,Str3,Str5,Str6,Str7,Str8
  if S == 5: Label5()
  if S == 6: Label6()
  if S == 8: Label8()
  if S == 9: Label9()
  if S == 10: Label10()
  if S == 11: Label11()
  if S == 13: Label13()
  if S == 14: Label14()
  if S == 16: Label16()
  if S == 17: Label17()
  if S == 18: Label18()
  if S == 19: Label19()
  if S == 21: Label21()
  if S == 23: Label23()
  print "DISPATCH ERROR S="+str(S)

def Label3():
  global A,B,C,D,S,N,L1,Z,X,Y
  global Str1,Str2,Str3,Str5,Str6,Str7,Str8
  Str1 = Str1[len(Str1)-(A+1)*8:len(Str1)-(A+1)*8+8]
  Label0()

def Label2():
  global A,B,C,D,S,N,L1,Z,X,Y
  global Str1,Str2,Str3,Str5,Str6,Str7,Str8
  Str1 = "";
  while A != 0:
    A = A/2.0
    if A-int(A) > 0:
      Str1 = "1" +Str1
    else:
      Str1 = "0" + Str1
    A=int(A)
  while len(Str1) < 32:
    Str1 = "0" + Str1
  Label0()
# ...
```

Listing 7: Extrait de la traduction Python

Cette traduction permet de simuler l'exécution du programme :

```
$ pypy emu.py
Entrez le code : 1234
PERDU
```

Listing 8: Exécution de la traduction Python

La vérification de la solution est réalisée par un contrôle entre une variable et le nombre 3298472535. Le programme utilise l'entrée de l'utilisateur comme un entier. Pour trouver la solution, une approche en force brute est choisie.
La traduction Python étant beaucoup trop lente avec les multiples conversions d'entiers vers chaînes binaires et inversement, XOR, Substring etc.., pour réaliser une attaque efficace, le code doit être simplifié.

Après avoir remplacé toutes les manipulations de chaînes de caractères par des manipulations d'entiers, le programme peut être simplifié pour arriver au résultat suivant :

```
L1 = [0,1996959894,3993919788,...,1510334235,755167117]
for i in xrange(0xFFFFFFFF):
    pos = (i & 0xff)^0xFF
    accu = (0xffffff)^L1[pos]

    pos = ((i>>8)&0xFF)^(accu&0xff)
    accu = (accu>>8)^L1[pos]

    pos = ((i>>16)&0xFF)^(accu&0xff)
    accu = (accu>>8)^L1[pos]

    pos = ((i>>24)&0xFF)^(accu&0xff)
    accu = (accu>>8)^L1[pos]

    if accu == 0x3b654da8: # 3298472535 ^ 0xFFFFFFFF
        print "OK -> %d" % (i)
        break
```

Listing 9: Programme simplifié pour l'attaque en force brute

La solution est trouvée en moins d'une seconde sur un seul coeur d'un processeur classique (Core i5 M 520) :

```
$ time pypy bf.py
OK -> 89594902
pypy bf.py  0,72s user 0,02s system 99% cpu 0,736 total
```

Listing 10: Exécution de la traduction Python

Lorsque l'entrée de l'utilisateur est vérifiée par le calcul et la comparaison précédente, la valeur entrée est utilisée pour initialiser la graine du générateur de nombre pseudo aléatoires de la calculatrice (`Z->rand`). La clef solution de l'épreuve est générée à partir de 32 générations de nombre pseudo aléatoires multipliés par 256. Une implémentation du PRNG est trouvée sur stackoverflow.
Après avoir implémenté le PRNG dans la traduction Python, la clef est obtenue :

```
$ pypy emu.py
Entrez le code : 89594902
KEY: 57D9F82B49C1EB3993CB82D26E37F69C
```

Listing 11: Exécution de la traduction Python

2.2 Radio : trames GSM et SMS depuis un dump de SDR

Tu veux voir ma grosse antenne ?
Ou tu veux sentir mon téléphone ?

En plus je suis un bon coup : je compte double !

L'archive `radio.zip` fournie par un des personnages contient le fichier `rtl2832-f9.452000e+08-s1.0 00000e+06.bin`. RTL2832 correspond à une puce Realtek de démodulation DVB-T, couramment utilisée pour faire de la radio logicielle (SDR) à bas prix.
Le nom de l'épreuve et du fichier oriente vers le logiciel de radio logiciel RTL-SDR qui a certainement été utilisé pour enregistrer ce fichier.
Le nom du fichier contient également la fréquence (option "-f" de rtl-sdr) ainsi que la fréquence d'échantillonnage (option "-s" de rtl-sdr).

- Fréquence : 945.2 MHz
- Fréquence d'échantionnage : 1MHz

La fréquence du signal capturé correspond à une bande GSM, l'épreuve consiste donc certainement à trouver des informations dans des trames GSM.
Pour analyser cette capture, le logiciel gr-gsm/airprobe est utilisé. Dans un premier temps il faut convertir la capture radio dans le format utilisé par airprobe.
La conversion est réalisée avec `GNURadio`, le schéma suivant génère le fichier `gsm.cfile`.

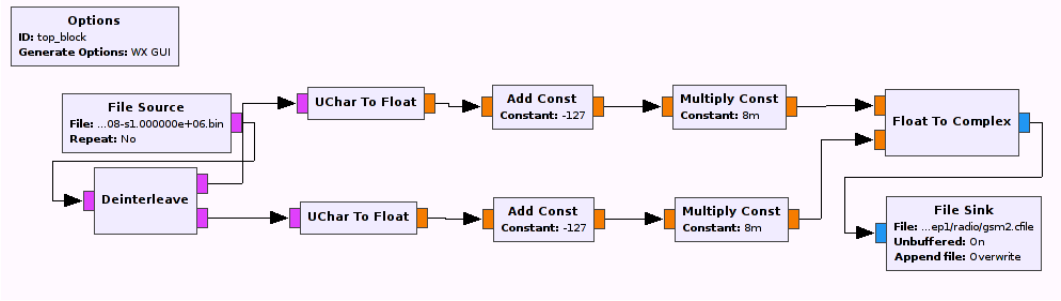


FIGURE 2: Convertion avec gnuradio

Ensuite les trames GSM sont décodées à l'aide de airprobe. Ce logiciel rejoue les trames GSM sur l'interface de loopback.

```
$ airprobe_decode.py --cfile=gsm.cfile --timeslot=0  --mode=BCCH_SDCCH4
```

Listing 12: Décodage des trames GSM

Les trames ainsi rejouées peuvent être analysées avec Wireshark.
Après parcours des différentes trames, un SMS est identifié, celui-ci contient la clef.

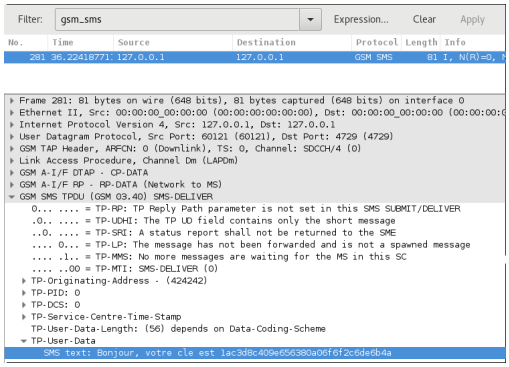


FIGURE 3: Trame GSM dans Wireshark

La clef obtenue permet de passer directement au second niveau, le coffre de clefs correspondant contenant deux clefs.

2.3 SOS-Fant0me : échanges réseau de Gh0st RAT

On suspecte une machine d'être hantée par un fantôme mais cette fois ce n'est pas Casper.
Tiens, voici une capture du champ électromagnétique... en gros une capture réseau. Tu pourras peut-être en tirer quelque chose !

L'épreuve débute avec l'archive `SOS-Fant0me.zip` fournie par un des personnages du jeu.
Cette archive contient une trace réseau au format tcpdump.

```
$ bsdtar -xvf ../SOS-Fant0me.zip
x SOS-Fant0me.pcap
$ file SOS-Fant0me.pcap
SOS-Fant0me.pcap: tcpdump capture file (little-endian) - version 2.4
```

Listing 13: Extraction de l'archive

Dans les flux TCP, la chaîne "Gh0st" se répète régulièrement :

```
Gh0stJ.......x.KS``........@.A....H3..X.r..##..N.........Og..gC..A9...eXZ.Gh0st........
x.c......Gh0st........x.S...$.$Gh0st........x....N.O......./.B.!..HH.U
..<.u.....l7)HH.y...'.1.......4-
%.....|..:........C1..zw..............ul)....3<..>.L:m....}H,..g.zDo".tiWVO.
\..t!....I..%...cK.;:.e../...
...6..-b..u.K.z......v.D....w.{-.#G
[..].Ly......s.........k`*....yZ8.X......Y}.........@.8MX...!~R}1...@NL#.f.-
t.yg.!.A.z........{*&.`.Fti..P.x....
...R.KQ.......L.K......[5#i.....^*..u.....=B48..._.......Gh0st........x..... .
Gh0st........x..f.....|Gh0st........x........Gh0stR...A...x...6204.70.7Zw.U02.24.U.K-
W.M-.NLO.R.....tV..s.H..I.KO.U....*...8...Gh0st........x...N.......Gh0st........x..)-
QPTp.....
```

FIGURE 4: Visualisation des flux TCP dans Wireshark

Après quelques recherches, il semble s'agir d'échange réseau entre le malware Gh0st RAT et son contrôleur. Une analyse du malware par McAfee donne beaucoup d'information sur le malware, en particulier la structure des échanges réseau.
Le protocole peut être décodé comme ceci :

- 5 octets : chaîne "Gh0st"
- 4 octets : taille total du packet
- 4 octets : taille de la charge Zlib decompressé
- octets suivant : charge Zlib

La charge Zlib une fois decompressée peut être décodée comme ceci :

- 1 octet : identifiant de la commande
- octets suivant : argument(s) de la commande ou données

Le décodeur présent en annexe (`ghost-decoder.py`) permet de décoder les échanges réseau et de sauvegarder les informations importantes (fichiers/frappes clavier) dans des fichiers.

```
$ python2 ghost-decoder.py
RSP login
CMD actived
CMD system
RSP pslist
  272 smss.exe    /SystemRoot/System32/smss.exe
    [...]
  1852  notepad.exe   notepad.exe
CMD keyboard
RSP keyboard_start
CMD next
RSP keyboard_data
[...]
CMD down_files : C:/Users/sstic/Documents/Challenge SSTIC 2016/Stage 1/sstic2016-stage1-solution.zip
[...]

sstic2016-stage1-solution.zip saved to files/sstic2016-stage1-solution.zip
visio_stage2.mp4              saved to files/visio_stage2.mp4
how_to_rule_the_world.txt     saved to files/how_to_rule_the_world.txt
Keyboard data                 saved to files/keyboard.txt
```

Listing 14: Décodage des échanges réseau

On observe dans l'échange avec le contrôleur :

- La commande `login` contient les informations du système : Hostname SSTIC-PC1, Windows 7 No service pack - Build : 7600 - Clock : 2500 Mhz - IP : 10.100.96.21 Webcam : yes (décodé avec chopshop)
- La commande `pslist` contient la liste des processus en cours d'exécution sur la machine
- Une suite d'échange de frappe clavier
- Une navigation dans les dossiers de la victime
- Des téléchargements de fichiers

Le script de décodage a sauvegardé les "frappes clavier" dans le fichier `keyboard.txt` ci-dessous :

```
$ cat files/keyboard.txt
[2016/02/27 - 23:14] Newmessage: [SSTIC 2016/Challenge] Stage 1
Salut ! Comme pis voici la clef pour le stage 1 !
Le mot de passe de l archive reste ceui convenu ensemble.
[2016/02/27 - 23:15] sstic2016-stage1-solution.zip - Saisir mot de passe
Cyb3rSSTIC_2016
```

Listing 15: Frappes clavier

Le mot de passe permettant d'extraire le fichier `sstic2016-stage1-solution.zip` est présent dans les "frappes clavier".

Les fichiers téléchargés sont les suivants :

- `how_to_rule_the_world.txt` : Une image en ASCII art
- `sstic2016-stage1-solution.zip` : Une archive ZIP contenant la clef de l'épreuve.
- `visio_stage2.mp4` : Une vidéo de Rick Astley : Never Gonna Give You Up

La clef de l'épreuve est donc obtenue dans l'archive :

```
$ unzip -P Cyb3rSSTIC_2016 sstic2016-stage1-solution.zip
Archive:  sstic2016-stage1-solution.zip
 extracting: solution.txt
$ cat solution.txt
368BE8C1CC7CC70C2245030934301C15
```

Listing 16: extraction de sstic2016-stage1-solution.zip

3 Niveau 2

3.1 Foo : Rétro-ingénierie d'une application EFI

Ce crackme vient d'en bas. Il faudra être EFIcace pour relever ce dEFI, car le débuggueur manque de dEFInition pour ce type de ROM arrangée.

L'archive `foo.zip` fournie par un des personnages contient le fichier `foo.efi`. Les indices donnés par le personnage et le type du fichier concordent, c'est une application EFI.

```
$ bsdtar -xvf ../foo.zip
x foo.efi
$ file foo.efi
foo.efi: PE32 executable (DLL) (EFI application) EFI byte code, for MS Windows
```

Listing 17: Extraction de l'archive

Qemu avec le firmware OVMF peuvent être utilisés pour exécuter l'application. Pour cela il faut mettre l'application `foo.efi` dans une image disque formatée en FAT :

```
$ truncate -s 1G disk.img
$ mkfs.vfat -F 32 disk.img
$ mkdir mountp
$ sudo mount -o loop disk.img ./mountp
$ sudo cp foo.efi ./mountp/
$ sudo umount ./mountp
```

Listing 18: Génération de l'image disque FAT32

```
$ qemu-system-x86_64 -cpu qemu64 -bios ./OVMF.fd -nographic -hda disk.img -enable-kvm
Shell> FS0:
FS0:\> foo.efi
UEFI checker
Missing key ?
FS0:\> foo.efi AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
UEFI checker
Sorry :(
FS0:\>
```

Listing 19: Exécution de l'application EFI dans Qemu

Les programmes EFI sont dans un bytecode dédié : ECB, ce code est exécuté dans une machine virtuelle. Le projet Tianocore propose une version Open-Source d'une machine virtuelle permettant d'exécuter ce Bytecode dans Qemu. Après une rapide lecture du binaire dans IDA pro, certaines instructions couramment utilisées dans les opérations cryptographiques sont ciblées pour être tracées :

- NOT
- OR
- SHL
- XOR

La machine virtuelle est modifiée pour générer une trace du passage par ces instructions (le patch complet est disponible en annexe : `edk2.patch`) :

```
@@ -3848,6 +3851,7 @@ ExecuteXOR (
   IN UINT64        Op2
   )
 {
+  Print(L"ip=0x%08x ExecuteXOR\t( %02X, %02X ) = %02X\n",VmPtr->Ip,Op1,Op2,Op1^Op2);
   return Op1 ^ Op2;
 }
```

Listing 20: Modification du firmware pour Qemu

Les messages ajoutés dans le code permettent d'obtenir la suite des instructions exécutées avec les valeurs des opérandes.

```
ip=0x066A6438 ExecuteSHL ( AA, 08 ) = AA00
ip=0x066A6442 ExecuteOR  ( AA, 00 ) = AA
ip=0x066A6444 ExecuteNOT ( AA ) = FFFFFF55
ip=0x066A6954 ExecuteXOR ( 55, CB ) = 9E
ip=0x066A6956 ExecuteOR  ( 00, 9E ) = 9E
ip=0x066A6438 ExecuteSHL ( AA, 07 ) = 5500
ip=0x066A6442 ExecuteOR  ( 55, 00 ) = 55
ip=0x066A6444 ExecuteNOT ( 55 ) = FFFFFFAA
ip=0x066A6954 ExecuteXOR ( AA, 41 ) = EB
ip=0x066A6956 ExecuteOR  ( 9E, EB ) = FF
```

Listing 21: Extrait de la trace générée lors de l'exécution

À partir de cette trace et en suivant les valeurs, il est aisé d'implémenter un simulateur en Python, le programme ne réalisant que très peu d'opérations. Certaines valeurs, ne dépendant pas de l'entrée, peuvent être utilisées précalculées.

```
input_data=[0xAA]*16
pre_computed_key = map(ord,list("CB41DCB1D89746705A7FE998F11ACCE7".decode("hex")))

accu = 0
for i in range(16):
  shift = 8-(i%8)
  a = input_data[i] << shift
  a = ((a>>8)&0xFF) | (a&0xFF)
  a ^= 0xFF
  accu |= a ^ pre_computed_key[i]

if accu != 0:
  print "Sorry :("
else:
  print "Success!"
```

Listing 22: Simulateur Python

Pour finir ; chaque octet est attaqué par force brute pour trouver la bonne clef :

```
pre_computed_key = map(ord,list("CB41DCB1D89746705A7FE998F11ACCE7".decode("hex")))
key=[0]*16

accu = 0
for i in range(16):
  shift = 8-(i%8)
  for try_key_byte in range(0x100):
    a = try_key_byte << shift
    a = ((a>>8)&0xFF) | (a&0xFF)
    a ^= 0xFF
    if a ^ pre_computed_key[i] == 0:
      key[i]=try_key_byte
      break

key_bin = "".join(map(chr,key))
print key_bin.encode("hex")
```

Listing 23: Brute force de la clef

```
$ time pypy solv.py
347d8c72720d6ec7a501583be0bccc0c
pypy solv.py  0,14s user 0,03s system 99% cpu 0,165 total
```

Listing 24: Extraction de l'archive de l'épreuve

3.2 Huge : Rétro-ingénierie d'un binaire x86-64 de 117To

C'est vraiment très indigeste ce challenge...

L'épreuve commence avec l'archive ZIP `huge.zip` fournie par un des personnages du jeu.

```
$ bsdtar -xvf ../huge.zip
x huge.tar
$ bsdtar -xvf huge.tar
x Huge
$ file Huge
Huge: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, corrupted section
    ↪ header size
$ ls -lah Huge
-rwxr-xr-x 1 david david 117T 31 dec.   1969 Huge
```

Listing 25: Extraction de l'archive

L'archive ZIP contient une archive tar, contenant un binaire x86-64 de 117 To !
Au passage, merci au filesystem ZFS qui permi l'extraction et l'exécution du binaire sans utiliser les 117To nécéssaires :

```
$ du -sh Huge
2,7M  Huge
```

Listing 26: Taille du binaire Huge

A l'exécution, le binaire demande un mot de passe. Lorsque celui ci est entré, l'exécution continue sans s'arrêter.

```
$ ./Huge
Please enter the password: AAA
...
```

Listing 27: Première exécution du binaire Huge

Pour analyser le comportement de ce binaire, qui ne peut pas être ouvert dans un désassembleur classique du fait de sa taille, un module pintool est utilisé (vous trouverez le module complet en annexe : `trace.cpp`). Celui-ci génère une trace contenant les instructions exécutées. Un compteur d'instructions est utilisé pour stopper l'exécution lorsque celui-ci atteint une valeur importante.

```
$ echo 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA' | /opt/pin/pin -t trace.so -- ./Huge
$ wc -l trace.out
200003 trace.out
```

Listing 28: Génération d'une trace avec pintool

L'exécution avec une clef choisie arbitrairement, montre l'exécution de beaucoup d'instructions `add byte ptr [rax], al`, soit 0x0000, dépendant du résultat d'un `cmp` :

```
0x51466a42e713 : mov rbp, rsp
... ; write(Please enter the password:)
; read(rsp,0x400)
0x51466a42e733 : xor rax, rax
0x51466a42e736 : xor rdi, rdi
0x51466a42e739 : mov rsi, rbp
0x51466a42e73c : mov edx, 0x400
0x51466a42e741 : syscall
... ; convertion depuis ASCII
0x43abdb4a0aee : nop
0x43abdb4a0aef : mov rbx, 0x43abdb4a0b0b
0x43abdb4a0af9 : cmp byte ptr [rsp], 0x29
0x43abdb4a0afd : jnz 0x43abdb4a0b09
0x43abdb4a0aff : mov rbx, 0x4a170682000c
0x43abdb4a0b09 : jmp rbx
0x43abdb4a0b09 : jmp rbx
0x43abdb4a0b0b : add byte ptr [rax], al
```

```
... ; instructions 0x0000
```

Listing 29: Trace d'exécution avec une clef arbitraire

Le premier `cmp` compare le premier octet de la clef avec 0x29, la suite de l'exécution dépend de cette comparaison. En générant une nouvelle trace avec le premier octet de la clef fixé à 0x29, on observe un second `cmp` et ainsi de suite. Ce qui permet de fixer certains octets de la clef :

```
input_key=['\x00']*16

#0x43abdb4a0af9 : cmp byte ptr [rsp], 0x29
input_key[0] = '\x29'
#0x4a170682ede1 : cmp word ptr [rsp+0x2], 0xd17e
input_key[0x2] = '\x7E'
input_key[0x3] = '\xD1'
#0x6f4b0e0f370 : cmp byte ptr [rsp+0xb], 0x8c
input_key[0xb] = '\x8C'
```

Listing 30: Comparaisons effectuées sur la clef

La génération d'une trace avec une clef validant les conditions précédentes permet d'obtenir la vérification suivante. La valeur stockée à l'adresse [rax], dépend du nombre d'instructions 0x0000 exécutées, ce nombre d'instructions dépend de la valeur du deuxième octet de la clef. La solution peut être trouvée avec le code suivant :

```
#0x49e7e541be19 : push rax (old rsp = rsp-8)
#0x49e7e541be1a : lea rax, ptr [rsp+0x10b]
# al = 3
#0x49e7e541be22 : movzx rbx, byte ptr [rsp+0x9]
#0x49e7e541be28 : mov byte ptr [rax], 0x0
#0x49e7e541be2b : lea rcx, ptr [rip+0x6]
#0x49e7e541be32 : lea rcx, ptr [rcx+rbx*2]
#0x49e7e541be36 : jmp rcx
#0x49e7e541be38 : add byte ptr [rax], al
#[...]
#0x49e7e541c038 : cmp byte ptr [rax], 0x65
sol=0
for i in range(0x100):
  value = ((0x49e7e541c038-(0x49e7e541be38+i*2))/2)*3
  if value & 0xFF == 0x65:
    sol = i
    break
input_key[0x9-8] = chr(sol)
```

Listing 31: Comparaisons dépendant d'un offset

De nouveau des traces sont générées avec des clefs validant les différentes conditions précédentes, de nouvelles conditions sont observées :

```
# 0x352845ab3bce : lea rbx, ptr [rip]
# 0x352845ab3bd5 : xor ebx, dword ptr [rsp+0xc]
# 0x352845ab3bd9 : cmp ebx, 0xa9b00f5c
result = struct.pack("<I",0x45ab3bd5 ^ 0xa9b00f5c)
pos = 0xC
for i in result:
  input_key[pos]=i
  pos+=1

# 0x000059cb440c4524 :  48 bb 83 0b cc c8 cb    movabs rbx,0x59cbc8cc0b83
# 0x000059cb440c452b :  59 00 00
# 0x000059cb440c452e :  48 8b 0d 19 00 00 00    mov    rcx,QWORD PTR [rip+0x19]
# 0x000059cb440c4535 :  50                      push   rax
# 0x000059cb440c4536 :  8b 44 24 10             mov    eax,DWORD PTR [rsp+0x10]
# 0x000059cb440c453a :  48 31 d8                xor    rax,rbx
# 0x000059cb440c453d :  48 bb 0c 00 4a e2 7e    movabs rbx,0x2a7ee24a000c
# 0x000059cb440c4544 :  2a 00 00
# 0x000059cb440c4547 :  48 0f b1 d9             cmpxchg rcx,rbx
# 0x000059cb440c454b :  58                      pop    rax
# 0x000059cb440c454c :  ff e1                   jmp    rcx
# 0x000059cb440c454e :  56                      push   rsi
# 0x000059cb440c454f :  45 0c 44                rex.RB or al,0x44
# 0x000059cb440c4552 :  cb                      retf
# 0x000059cb440c4553 :  59                      pop    rcx
```

```
rcx = 0x59cb440c4556 # load from addr 0x000059cb440c4535+0x19 = 0x59cb440c454e

rbx = 0x59cbc8cc0b83 # from ip=0x000059cb440c4524
result = struct.pack("<I",rbx ^ rcx)

pos = 0x10-8

for i in result:
  input_key[pos]=i
  pos+=1
```

Listing 32: comparaison suivantes morceaux de clef "XORés"

Les octets restants sont utilisés dans des calculs de flottants :

- Un flottant dans sa représentation binaire est "XORé" avec 4 octets de la clef.
- Le résultat du XOR est chargé dans le registre FPU `st0`
- La valeur du registre `st0` est copiée vers le registre `st1`
- Un cosinus est appliqué sur `st0`
- Les valeurs de `st0` et `st1` sont comparées

Pour obtenir les octets de la clef, il faut donc résoudre $\cos(X) = X$, puis convertir le résultat dans sa forme binaire IEEE 754, et effectuer un XOR de cette valeur avec le flottant présent dans le code (à l'adresse `0x2a7ee24aae74`). Le code suivant permet d'obtenir les octets manquants de la clef :

```
# 0x2a7ee24aae26 : lea rdi, ptr [rsp+0x18]
# 0x2a7ee24aae2b : lea rsi, ptr [rip+0x42]
# 0x2a7ee24aae32 : mov ecx, 0xa
# 0x2a7ee24aae37 : rep movsb byte ptr [rdi], byte ptr [rsi]
# 0x2a7ee24aae37 : rep movsb byte ptr [rdi], byte ptr [rsi]
# 0x2a7ee24aae39 : mov ebx, dword ptr [rsp+0xc]
# 0x2a7ee24aae3d : xor dword ptr [rsp+0x1c], ebx
# 0x2a7ee24aae41 : fwait
# 0x2a7ee24aae42 : fnclex
# 0x2a7ee24aae44 : fld st0, ptr [rsp+0x18]
# 0x2a7ee24aae48 : fld st0, st0
# 0x2a7ee24aae4a : fcos st0
# 0x2a7ee24aae4c : fcompp st0, st1
# 0x2a7ee24aae4e : fwait
# 0x2a7ee24aae4f : fnstsw ax
# 0x2a7ee24aae51 : and ax, 0xffdf
# 0x2a7ee24aae55 : cmp ax, 0x4000
# 0x2a7ee24aae59 : mov rbx, 0x99a3805000c
# 0x2a7ee24aae63 : mov r14, 0x2a7ee24aae7e
# 0x2a7ee24aae6d : cmovnz rbx, r14
# [...]
# 0x2a7ee24aae74 :  cb 6d 71 1e 38 78 b8 24 fe 3f

# COS(X)=X -> solution ~0.739085133215
ieee_solution = numpy.float128(0.739085133215).tostring()[::-1]
search = struct.unpack(">I",ieee_solution[8:8+4])[0]
xor_key = 0x24b87838

key = xor_key ^ search
key_str = struct.pack("<I",key)
pos=0x4
for i in key_str:
  input_key[pos]=i
  pos+=1
```

Listing 33: comparaison sur les flottants

Lorsque la clef fournie est valide, le programme réalise un XOR de cette clef avec une valeur stockée pour obtenir la clef permettant de déchiffrer un des coffres de clef du jeu.

```
# 0x99a380575a7 : movdqa xmm0, xmmword ptr [rsp]
# 0x99a380575ac : movdqa xmm1, xmmword ptr [rip+0x6c]
# 0x99a380575b4 : pxor xmm0, xmm1
# 0x99a380575b8 : mov rdi, rsp
# 0x99a380575bb : mov rbx, 0x2c7affef000c
# 0x99a380575c5 : call rbx

# 0x99a38057620 :   CCFDCBC5B2A9E62B0D7E87370E2E4F19

xor_key = list("CCFDCBC5B2A9E62B0D7E87370E2E4F19".decode("hex"))

out_key=""

for i in range(16):
  out_key+=chr(ord(input_key[i])^ord(xor_key[i]))
```

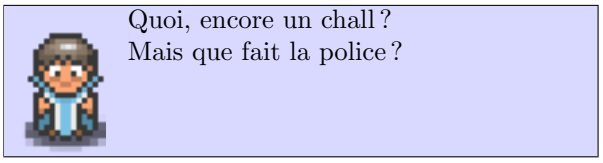
Listing 34: Génération de la solution de l'épreuve

Le code complet est disponible en annexe, voici une trace d'exécution :

```
$ python2 generate_key.py
Input key  : 29897ed1d4d68c99d54ec08c89341bec
Output key : e574b514667f6ab2d83047bb871a54f5
$ ./Huge
Please enter the password: 29897ed1d4d68c99d54ec08c89341bec
The key is: E574B514667F6AB2D83047BB871A54F5
```

Listing 35: Validation de l'épreuve

3.3 Loader : Rétro-ingénierie d'une police TrueType



L'épreuve débute avec l'archive `loader.zip` fournie par un des personnages du jeu. Cette archive contient un binaire PE 32 bits :

```
$ bsdtar -xvf ../loader.zip
x loader.exe
$ file loader.exe
loader.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

Listing 36: Extraction de l'archive

FIGURE 5: exécution de loader.exe

Une analyse rapide du binaire dans IDA pro permet de comprendre le fonctionnement de l'application :

- Elle vérifie que le système d'exploitation est Windows 7, 8 ou 8.1, sinon elle affiche un message d'erreur.
- Elle charge une police TrueType à partir d'une ressource dans le binaire : BizarroSSTIC
- Elle capture les "évènements clavier"
- Pour les "frappes clavier" alphanumériques et `+/=?` les caractères sont ajoutés dans la zone de texte correspondant à la clef
- La zone de texte peut contenir un maximum de 25 caractères

L'analyse du binaire ne montre pas d'autre opération. L'indice fourni en début d'épreuve oriente vers la police.
La ressource est extraite du binaire à l'aide de l'outil ripPE.

```
$ python2 ripPE/ripPE.py --file=loader.exe --section=resources --dump
$ file ripPE-*
ripPE-RESOURCE-[...].rsrc:    TrueType font data
ripPE-RESOURCE-[...].rsrc: data
$ mv mv ripPE-RESOURCE-[...].rsrc BizarroSSTIC.ttf
```

Listing 37: Extraction des ressources du binaire

La police extraite peut ensuite être analysée avec l'outil `FontForge`.

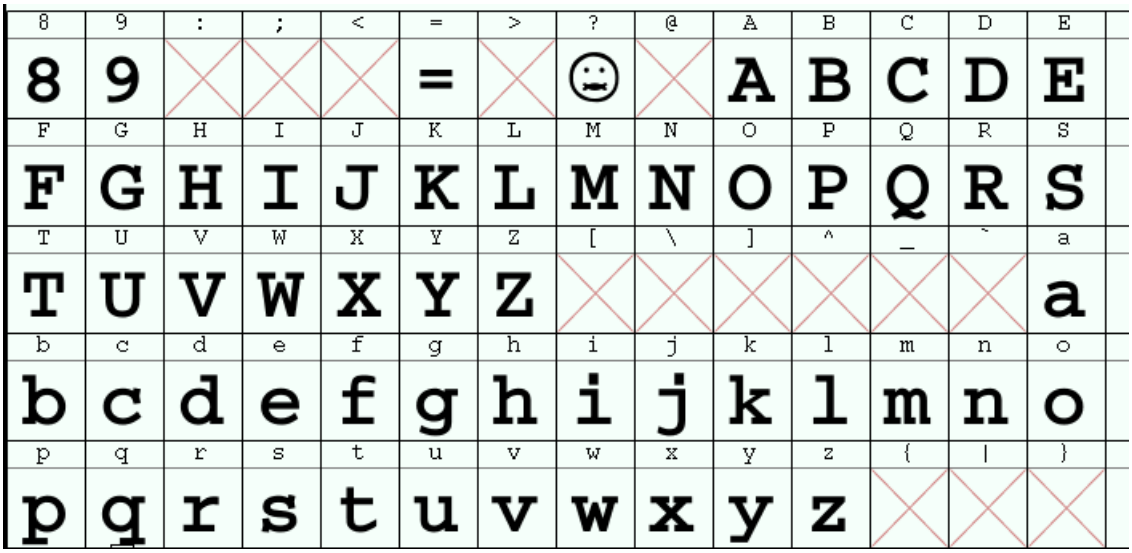


FIGURE 6: Visualisation de la police dans FontForge

La première chose qui saute aux yeux est le motif pour le caractère "?". C'est une frimousse^{TM} avec la bouche dans les deux sens, alors que lors de la frappe celle-ci apparaît uniquement avec une bouche orientée vers le bas.

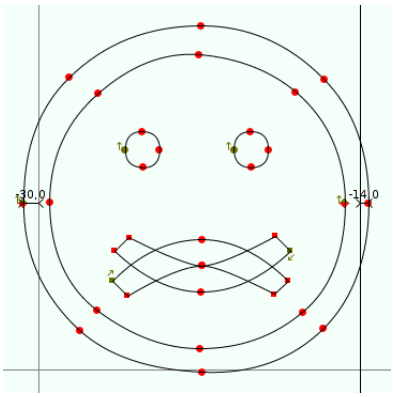


FIGURE 7: Visualisation du caractère dans FontForge

Après quelques recherches il apparaît que les polices TrueType peuvent embarquer des instructions pour un interpréteur interne à TrueType. Une documentation sur le site de Microsoft détaille le fonctionnement de la VM et des instructions disponibles. Le logiciel FontForge permet de désassembler ces instructions :

```
  0 b0   PUSHB_1
  1  00   0
  2 43   RS
  3 b0   PUSHB_1
  4  01   1
  5 43   RS
  6 b0   PUSHB_1
  7  02   2
  8 43   RS
  9 b0   PUSHB_1
 10  03   3
 11 43   RS
 12 b0   PUSHB_1
 13  04   4
 14 43   RS
 15 b0   PUSHB_1
 16  05   5
 17 43   RS
```

FIGURE 8: Visualisation du caratère dans FontForge

La VM TrueType repose sur une stack, et sur une zone de stockage. Des instruction permettent de push/pop sur la stack, et de lire ou écrire à des index de la zone de stockage.
Le caractère "question" contient beaucoup d'instructions, les autres caractères en contiennent eux beaucoup moins. L'analyse débute donc par les caractères "simples".

Chaque caractère réalise les opérations suivantes :
- Lire la valeur à l'index 99 de la zone de stockage.
- Écrire une valeur dépendant de la lettre (voir tableau suivant) à l'index lu précédemment.
- Incrémenter la valeur lue à l'index 99 de la zone de stockage de 1.
- Écrire la valeur incrémentée à l'index 99 de la zone de stockage.

numero	lettre		numero	lettre		numero	lettre
0	a		0	=			
1	b		22	w		43	R
2	c		23	x		44	S
3	d		24	y		45	T
4	e		25	z		46	U
5	f		26	A		47	V
6	g		27	B		48	W
7	h		28	C		49	X
8	i		29	D		50	Y
9	j		30	E		51	Z
10	k		31	F		52	0
11	l		32	G		53	1
12	m		33	H		54	2
13	n		34	I		55	3
14	o		35	J		56	4
15	p		36	K		57	5
16	q		37	L		58	6
17	r		38	M		59	7
18	s		39	N		60	8
19	t		40	O		61	9
20	u		41	P		62	+
21	v		42	Q		63	/

Le caractère "question" réalise des opérations à partir des 22 premiers emplacements de la zone de stockage.
En analysant les opérations réalisées par le caractère question, il est possible d'obtenir le numéro nécessaire pour chaque valeur de la zone de stockage.
En effet plusieurs conditions sont présentes dans le code (instruction "EQUAL").
Pour obtenir toutes ces conditions en fonction des valeurs présentes dans la zone de stockage, un émulateur en Python est développé (consultable en annexe : `emulator-string-vars.py`) pour interpréter uniquement les instructions du caractère "question". Les équations sont obtenues sur les instructions EQUAL.

```
$ python2 emulator-string-vars.py |grep EQUAL
PC = 0x00000051    EQUAL(19,storage[2])
PC = 0x00000074    EQUAL(186,(384*storage[12]/64))
PC = 0x00000082    EQUAL(63,storage[3])
PC = 0x00000094    EQUAL(39,storage[8])
PC = 0x000000a5    EQUAL(35,storage[7])
PC = 0x000000e1    EQUAL(263,((320*(6+storage[13])/64)-7))
PC = 0x000000ef    EQUAL(26,storage[5])
PC = 0x00000103    EQUAL(28,(6+storage[21]))
PC = 0x00000123    EQUAL(21,storage[18])
PC = 0x00000137    EQUAL(53,(storage[0]-3))
PC = 0x00000143    EQUAL(21,(storage[11]-6))
PC = 0x00000169    EQUAL(582,(192*(6+(256*storage[19]/64))/64))
PC = 0x00000177    EQUAL(20,(256*storage[1]/64))
PC = 0x00000185    EQUAL(2,storage[15])
PC = 0x000001a2    EQUAL(1463,(448*(1+(256*storage[4]/64))/64))
PC = 0x000001b0    EQUAL(1415,(320*((320*storage[14]/64)-2)/64))
PC = 0x000001c2    EQUAL(70,(2+(4+(2+storage[17]))))
PC = 0x000001d7    EQUAL(32,(storage[10]-1))
PC = 0x000001e8    EQUAL(5,(storage[16]-2))
PC = 0x000001fb    EQUAL(90,(320*(64*(7+(2+storage[9]))/64)/64))
PC = 0x0000020e    EQUAL(3,(64*((5+(((5+storage[6])-1)-3))-6)/64))
PC = 0x00000222    EQUAL(1970,(320*((256*(3+(256*((storage[20]-7)-1)/64))/64)-2)/64))
```

Listing 38: Génération des équations avec l'émulateur en Python

Les équations peuvent être résolues de la manière suivante :

- storage[0] = 56
- storage[1] = 5
- storage[2] = 19
- storage[3] = 63
- storage[4] = 52
- storage[5] = 26
- storage[6] = 3

— storage[7] = 35
— storage[8] = 39
— storage[9] = 9
— storage[10] = 33
— storage[11] = 27
— storage[12] = 31
— storage[13] = 48
— storage[14] = 57
— storage[15] = 2
— storage[16] = 7
— storage[17] = 62
— storage[18] = 21
— storage[19] = 47
— storage[20] = 32
— storage[21] = 22

En utilisant le tableau de conversion précédent, la chaîne `4ft/0AdJNjHBFW5ch+vVGw`, solution de l'épreuve est obtenue (après ajout du padding base64).

```
$ echo '4ft/0AdJNjHBFW5ch+vVGw==' |base64 -d |xxd
00000000: e1fb 7fd0 0749 3631 c115 6e5c 87eb d51b
```

Listing 39: solution de l'épreuve

Pour vérifier la solution, l'émulateur fourni en annexe (`emulator.py`) permet d'interpréter l'ensemble des instructions (tous les caractères).

```
$ python2 emulator.py |grep EQUAL |grep '^PC'
PC = 0x00000159    EQUAL(19,19)
PC = 0x0000017c    EQUAL(186,186)
PC = 0x0000018a    EQUAL(63,63)
PC = 0x0000019c    EQUAL(39,39)
PC = 0x000001ad    EQUAL(35,35)
PC = 0x000001e9    EQUAL(263,263)
PC = 0x000001f7    EQUAL(26,26)
PC = 0x0000020b    EQUAL(28,28)
PC = 0x0000022b    EQUAL(21,21)
PC = 0x0000023f    EQUAL(53,53)
PC = 0x0000024b    EQUAL(21,21)
PC = 0x00000271    EQUAL(582,582)
PC = 0x0000027f    EQUAL(20,20)
PC = 0x0000028d    EQUAL(2,2)
PC = 0x000002aa    EQUAL(1463,1463)
PC = 0x000002b8    EQUAL(1415,1415)
PC = 0x000002ca    EQUAL(70,70)
PC = 0x000002df    EQUAL(32,32)
PC = 0x000002f0    EQUAL(5,5)
PC = 0x00000303    EQUAL(90,90)
PC = 0x00000316    EQUAL(3,3)
PC = 0x0000032a    EQUAL(1970,1970)
```

Listing 40: solution de l'épreuve dans l'émulateur

La solution peut également être visualisée dans le programme, le caractère "question" génère bien une frimousse™ heureuse.

FIGURE 9: exécution de loader.exe avec la bonne clef

4 Niveau 3

4.1 Strange : Rétro-ingénierie d'un binaire Itanium (ia-64)

4.1.1 Découverte

Ce matin rien n'a changé
Reflétant le soleil,
Les chiffres me regardent.

Attention ! Ce haïku compte double.

L'épreuve débute avec l'archive `strange.zip` fournie par un des personnages du jeu. Cette archive contient un binaire ELF IA-64, et un fichier contenant des données non identifiées :

```
$ bsdtar -xvf ../strange.zip
x a.out
x 196
$ file a.out 196
a.out: ELF 64-bit LSB executable, IA-64, version 1 (SYSV), dynamically linked, interpreter
    ↪ /lib/ld-linux-ia64.so.2, for GNU/Linux 2.4.0, stripped
196:   data
```

Listing 41: Extraction de l'archive

Le premier fichier est un binaire ELF pour l'architecture IA-64, c'est une architecture processeur développé par Intel pour les CPU Itanium et destinée principalement aux serveurs.
Le second fichier n'est pas identifié, il ne contient aucune signature de format connu. La seule observation est que certains octets de ce fichier se répètent :

```
$ hexdump -C 196|head -n5
00000000  21 ea c8 a3 8d b4 49 3f  54 ed 76 38 e7 80 22 3f  |!.....I?T.v8...?|
00000010  95 6a a1 bf 81 3e 42 3f  68 0a e0 1e 3e dc 43 3f  |.j...>B?h...>.C?|
00000020  60 8d 70 33 16 ad 3e 3f  39 15 c0 58 3e 50 36 3f  |‘.p3..>?9..X>P6?|
00000030  23 85 84 64 3b 34 e5 3e  f9 2e d2 bc 63 13 43 3f  |#..d;4.>....c.C?|
00000040  33 af 29 a8 c0 08 4d 3f  3f 17 40 cc 0b f3 4b 3f  |3.)...M??.@...K?|
```

Listing 42: Extraction de l'archive

4.1.2 Analyse statique

Un émulateur pour l'architecture IA-64 développé par HP est disponible, mais il ne compile pas (du premier coup) sur la machine de l'auteur. Une analyse statique du binaire est donc réalisée pour en comprendre son fonctionnement.
L'analyse est effectuée dans IDA Pro qui dispose d'un support pour ce type de processeur.
La lecture de l'assembleur IA-64 est déroutante au début, mais après un temps d'adaptation c'est très lisible.
En effet, le processeur ne dispose pas de registre pour les flags, mais il dispose de plusieurs registres pour la prédiction de branchement (pX).
Pour la notation `(pX) ASM INSTRUCTION`, l'instruction ne sera exécutée que si pX est vrai.
Pour l'assembleur suivant `cmp4.eq p6, p7 = 2, r32`

- `p6 = (2 == r32)`
- `p7 = !p6`

Cette propriété réduit fortement le nombre de branchements, ce qui simplifie la lecture du code.

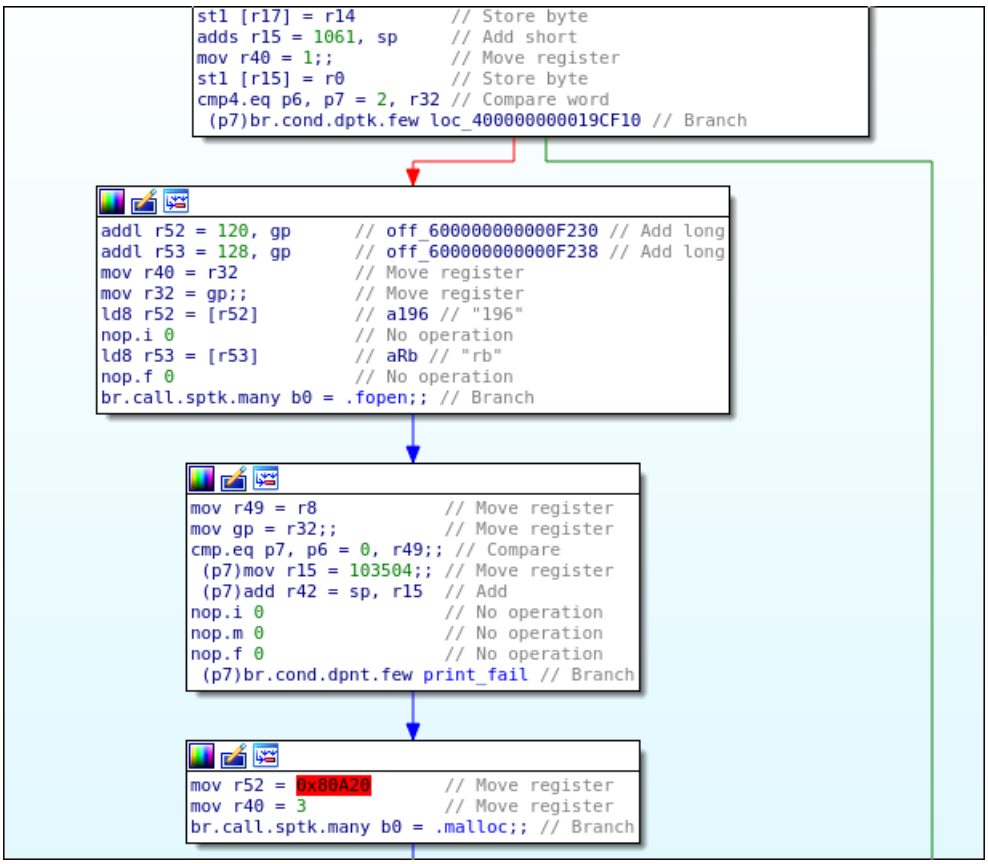


FIGURE 10: Début de la fonction principale

La fonction principale est identifiée en 0x400000000019C700, voici le fonctionnement de sa première partie :

- La valeur 0x3634B0B3 est écrite sur la stack
- Le fichier 196 est ouvert avec `fopen` et le mode "rb"
- En cas d'erreur de `fopen` (et de toutes les fonctions suivantes) une sous fonction est appelée et affiche avec un `printf` la valeur précédemment écrite sur la stack décalé de 1 bit vers la gauche, soit la chaîne "fail".
- Un malloc de 0x80A20 est effectué
- Le nom de fichier fourni en paramètre est ouvert avec `fopen` est le mode "r"
- Si l'ouverture c'est bien déroulée, la première ligne du fichier est lue avec `fgets`.
- Un memcmp compare les 3 premiers octets du fichier avec la chaîne "P2\n"
- La seconde ligne du fichier est lue avec `fgets`, et le premier caractère est comparé à "#".
- Deux valeurs séparées par un espace sont lues sur la ligne suivante avec `sscanf("%d %d")`, et leur multiple comparé à 12800.

Le parsing réalisé par ce binaire sur le fichier permet d'en deviner son type : c'est une image au format Portable Graymap, un format contenant uniquement de l'ASCII.

Ce format se défini ainsi (source Netpbm) :

- La 1ère ligne contient uniquement le magique "P2"
- La 2nd ligne contient un commentaire débutant par '#'
- La 3ème ligne contient la géométrie de l'image en décimal nombre de colonnes, nombre de ligne.
- La 4ème ligne contient le nombre de niveaux de gris.
- Les lignes suivantes contiennent les pixels avec, pour chacun, le niveau de gris en décimales ascii.

En poursuivant la rétro-ingénierie, les vérifications effectuées par le binaire permettent d'obtenir des informations sur l'image :

- Elle doit faire 12800 pixels (XY=12800, mais il n'y a pas de vérification sur X ni Y)
- Le nombre de niveaux de gris doit être 255
- Les pixels ne peuvent être que 0 ou 255

Les différentes boucles dans la fonction principale utilisant les données issues de l'image, laissent penser que la géométrie de l'image est de 20 lines par 640 colones.
La suite de la fonction principale est plus complexe, les pixels sont convertis flottants soit 0 (pixel à 0) soit 1 (pixel à 255). Une boucle réalise 32 tours et effectue les actions suivantes :

- `printf(".\n")`
- exécute la fonction sub_400000000019C410, si son retour (r8) n'est pas 0 le programme fini par l'exécution de la fonction affichant (fail).
- exécute la fonction sub_400000000019AFC0
- Un flottant est chargé depuis la zone des variables globales : 0x3FC3333333333333 soit 0,15
- 4 flottants sont chargés depuis des zones mémoires manipulées par des sous fonctions de sub_400000000019AFC0
- Pour chacun des 4 flottants, la fonction vérifie s'ils sont inférieurs à 0,15. Dans le cas contraire l'exécution est interrompu et le message "fail" est affiché.

Si les flottants sont inférieurs à 0,15 pour les 32 tours de boucle, alors un message issu d'une multiplication de flottant est affiché.
Dans cette boucle la première vérification est effectuée par la fonction sub_400000000019C410. Celle-ci effectue les actions suivantes :

- Vérifie que la première ligne pour les 20 premières colonnes est blanche, si ce n'est pas le cas la fonction retourne 1
- Vérifie que la ligne 20 est blanche, mais cela n'affecte pas le retour de la fonction
- Vérifie que le nombre de colonnes entièrement blanches pour les premières colonnes est inférieur au nombre de colonnes blanches pour les dernières colonnes (jusqu'à la colonne 20).

Les flottants comparés en fin de boucle sont certainement définis par la fonction sub_400000000019AFC0 exécutée juste avant ceux-ci.
Cette fonction est bien plus longue, elle appelle successivement 161 sous-fonctions, sans aucune boucle.
Les sous-fonctions appelées sont toutes similaires. Elles ne comportent aucune instruction conditionnelle, uniquement des lectures et écriture de flottants en mémoire (depuis deux zones distinctes) et des opérations sur les flottants (additions et multiplications). Vers la fin de ces fonctions, une exponentielle est réalisée à l'aide le la fonction `exp` de la libc.
Les opérations sur les flottants sont réalisées avec l'instruction `fma`, réalisant l'opération `C = A*B+C`.
Après avoir suivi les adresses, chacune de ces fonctions réalise les opérations suivantes :

- Lire un flottant dans le fichier 196 `->` C
- Pour chacun des pixels du bloc de 20x20 ciblé (flottant 0 ou 1) : `C = (valeur du pixel) * (valeur lue dans le fichier 196) + C`
- Changement du signe de C
- `C = exp(C)`
- `C = C+1`
- `C = 1/C`
- Stockage en mémoire de la valeur C

Cette fonction est réalisée 160 fois sur le même bloc de 20x20 pixels avec des offsets dans le fichier 196 différents.
Ensuite, la 161ème effectue le même type d'opérations, mais sur les résultats stockés par les 160 fonctions précédentes :

- Lire un flottant dans le fichier 196 `->` C
- Pour chacun des 160 résultats des fonctions précédentes : `C = (valeur stockée) * (valeur lue dans le fichier 196) + C`
- Changement du signe de C
- `C = exp(C)`
- `C = C+1`
- `C = 1/C`
- Stockage en mémoire de la valeur C

Dans la 161ème fonction ces opérations sont réalisées 4 fois, les 4 résultats sont les valeurs flottantes comparées à `<` 0,15 dans la fonction principale.

4.1.3 Simulateur et solution de l'épreuve

Après avoir compris le fonctionnement, une version Python du programme est implémenté. Voici le coeur du programme, les fonctions auxiliaires sont disponibles dans la version complète en annexe (`generate.py`) :

```
def sub_4000000000000B30(round_n,count):
  last_c_float = float(0)
  for line in range(20):
    for column in range(20):
      if line == 0 and column == 0:
        last_c_float = get_float(-1,0)
      fvalue = get_float(column,line)
      c = float(0)
      if get_pixel_value(column+20*count,line) and line > 0:
        c = last_c_float + fvalue
      else:
        c = last_c_float
      last_c_float = c
  # 4000000000008816                fneg f8 = f6
  last_c_float = last_c_float * -1.0
  # call .exp
  exp_value = math.exp(last_c_float)
  # 4000000000008826                fadd.d.s0 f8 = f8, f1
  exp_value = exp_value + 1.0
  # reciprocal approximation
  f8 = 1.0/exp_value
  find_floats[round_n] = f8

# sub_40000000004F12F0
def sub_40000000004F12F0(count):
  start_offset = 0x7f590 + (count*0x80a20)

  offset = start_offset
  for j in range(4):
    init = get_float_at_offset(offset)

    last = init
    for i in range(160):
      offset+=8
      from_file = get_float_at_offset(offset)
      find_before = find_floats[i]
      c = (find_before * from_file) + last
      last=c
    last = last * -1.0
    exp_value = math.exp(last)
    exp_value = exp_value + 1.0
    f8 = 1.0/exp_value
    if f8 <= 0.15:
      print "\t\033[32;1mGOOD FLOAT %f\033[0m" % (f8)
    else:
      print "\t\033[31;1mBAD  FLOAT %f\033[0m" % (f8)
    offset+=40


for count in range(32):
  print "---- pos = %d" % (count)
  find_floats = [0]*160
  for i in range(160):
    sub_4000000000000B30(i,count)
    start_index += 3240
  sub_40000000004F12F0(count)
  start_index = 3208+(0x80a20*(count+1))
```

Listing 43: Implémentation Python issue de la rétro-ingénierie

Pour trouver la solution, la superposition de masques (lus dans le fichier 196) est tentée, sans aucun succès. Il serait bien trop complexe de remonter aux valeurs des pixels, du fait des nombreuses additions et exponentielles.
La lecture des flottants dans le fichier 196 ne débute pas à l'offset 0 mais à l'offset 3200, ce qui est étrange, de plus cela correspond exactement à 20*20 valeurs flottantes stockées sur 8 octets.
L'observation des valeurs flottantes pour les 400 (20x20) premiers offsets, permet d'isoler deux groupes de valeurs :

- Des valeurs proches de 0 dans 89 % des cas
- Des valeurs supérieures à 1 dans le reste des cas

Ces valeurs sont converties en image, le pixel est blanc si la valeur flottante est proche de 0, noir dans l'autre cas. Voici le résultat :

FIGURE 11: image convertie à partir des valeurs flottantes

Ce chiffre, comme l'ensemble de ceux disponibles dans le fichier 196 (aux offsets non utilisés par le programme), permettent de tester chaque possibilité pour chacune des positions dans la clef. Contre toute attente, il n'y a pas de lettre pour constituer un alphabet hexadécimal, la clef doit donc être uniquement numérique. Le script permettant d'extraire l'ensemble des chiffres est disponible en annexe.
Ensuite il ne reste plus qu'à utiliser l'implémentation Python pour tester chaque caractère, pour les 32 positions dans l'image.
Lorsque la comparaison des 4 flottants < 0,15 est validée le caractère est retenu pour la position.

```
$ pypy generate.py
---- pos = 0
  GOOD FLOAT 0.004742
  GOOD FLOAT 0.000048
  GOOD FLOAT 0.009203
  GOOD FLOAT 0.000158
[...]
---- pos = 31
  GOOD FLOAT 0.000295
  GOOD FLOAT 0.001045
  GOOD FLOAT 0.013441
  GOOD FLOAT 0.033773
```

Listing 44: Exécution de l'implémentation Python pour des caractères valides

La comparaison est juste pour les 32 tours de boucle pour l'image suivante :

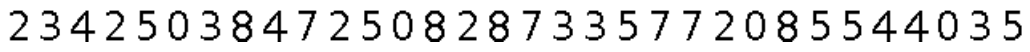
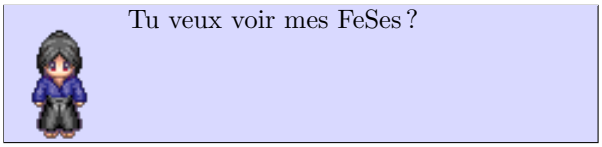


FIGURE 12: Image validant toutes les conditions

La rétro-ingénierie de l'ensemble n'était pas nécessaire avec une version fonctionnelle de SKI. La boucle principale affichant des points à chaque tour, il aurait été simple de tester chaque caractères, après avoir effectué la rétro-ingénierie de la seule fonction principale.

4.2 USB : Rétro-ingénierie d'un driver Windows de chiffrement de fichier



L'épreuve débute avec l'archive `usb.zip` fournie par un personnage. Cette archive contient un binaire Windows x86-64 :

```
$ bsdtar -xvf ../usb.zip
x img.bz2
x userSSTIC.bin
$ file userSSTIC.bin
userSSTIC.bin: PE32+ executable (GUI) x86-64, for MS Windows
$ bunzip img.bz2
$ file img
img: DOS/MBR boot sector; partition 1 : ID=0xb, start-CHS (0x0,0,4), end-CHS (0x88,233,5),
    ↪ startsector 3, 2097152 sectors; partition 2 : ID=0xb, start-CHS (0x89,19,6), end-CHS
    ↪ (0xcd,135,36), startsector 2099201, 1048575 sectors; partition 3 : ID=0xb, start-CHS
    ↪ (0xcd,135,38), end-CHS (0xce,218,57), startsector 3147777, 20480 sectors; partition 4 :
    ↪ ID=0xb, start-CHS (0xce,218,58), end-CHS (0x14,93,50), startsector 3168257, 819199 sectors
```

Listing 45: Extraction de l'archive

Le programme utilisateur `userSSTIC.bin` est rapidement analysé dans un premier temps :

- Le programme charge un driver `drvSSTIC.sys`, issu d'une ressource
- En suivant les chaînes, une boucle listant les fichiers dans `%SystemDrive%/SSTIC/` est identifiée.
- Des évènements permettent de communiquer avec le driver.

```
if ( !ExpandEnvironmentStringsA("%SystemDrive%", Dst, 0x104u) )
  return GetLastError();
v1 = -1i64;
do
  ++v1;
while ( Dst[v1] );
strcpy_s(&FileName, v1 + 1, Dst);
strcat_s(&FileName, 0x104ui64, "\\SSTIC\\*");
v2 = FindFirstFileA(&FileName, &FindFileData);
if ( v2 == (HANDLE)-1i64 )
  return GetLastError();
do
{
  if ( !(FindFileData.dwFileAttributes & 0x10) )
  {
    WaitForSingleObject(*(HANDLE *)qword_14001CB28, 0xFFFFFFFF);
    ResetEvent(*(HANDLE *)qword_14001CB28);
    if ( !(unsigned int)sub_140001450(&FileName, FindFileData.cFileName) )
      SetEvent(*(HANDLE *)(qword_14001CB28 + 8));
  }
  result = FindNextFileA(v2, &FindFileData);
}
while ( result );
return result;
```

FIGURE 13: Liste des fichiers dans C :/SSTIC

Le but est donc de retrouver les fichiers initialement dans le dossier SSTIC, à l'intérieur de l'image disque `img`.
Le driver est obtenu à l'aide de l'outil ripPE permettant d'extraire les ressources.
Pour comprendre le déroulement des opérations, le driver est étudié dans WinDBG. Des fichiers sont placés dans le répertoire C :/SSTIC, et une clef USB vierge est connectée. Le driver s'active et il est possible de suivre l'exécution.
Plusieurs fonctions attirent l'attention :

- `sub_12284` utilise des données issues de `RtlRandomEx` avant les écritures de blocs.
- `sub_11E14` utilise les constantes d'initialisation de RC5 / RC6.

Pour comprendre plus en détail et tenter de déchiffrer des données, le driver est modifié pour n'utiliser que des octets zéro lors des appels à RtlRandomEx, et l'image de la clef USB est conservée. Dans l'image de la clef USB, on voit qu'une zone de 16 octets est maintenant à zéro.
La rétro-ingénierie à partir des fonctions identifiées précédemment permet de comprendre plus en détail l'enchaînement des opérations.

- Le fichier est chiffré avec une méthode non identifiée par `sub_122F8`, en utilisant des données issues de `sub_12284`. La fonction `sub_12284` est initialisée avec des données aléatoires,
- Les données aléatoires sont écrites à chaque fichier.

— Les en-têtes des fichiers contiennent : la taille du fichier et les données aléatoires permettant d'initialiser l'algorithme.
— Le résultat est surchiffré en RC6 avec la clef "551C2016B00B5F00" présente dans le binaire.

En revanche, la répartition des blocs sur le disque n'est pas comprise, mais au vu des zones à forte entropie bien délimitées présentes dans l'image disque, il devrait être possible de déchiffrer.
Pour débuter, il faut donc déchiffrer des données avec RC6 et la clef "551C2016B00B5F00". Le premier bloc est facilement identifiable à l'offset 0x240 de l'image disque.
Malheureusement aucune des implémentations RC6 testées, ne donne le même résultats que les traces dans WinDBG. Pour contourner le problème, le code du driver réalisant le déchiffrement RC6 est utilisé. Un module Python natif, permet de charger le binaire du driver en mémoire, de définir la fonction à partir de son offset dans le fichier, et de l'exécuter depuis Python. Le code complet de ce module est disponible en annexe (`sub_12038.c`). La valeur d'état du RC6 après le `rc6_key_setup` est obtenue dans WinDBG.
Les fonctions `sub_12284` et `sub_122F8` sont implémentées en Python, depuis la version décompilée fournie par IDA Pro.
Une fois déchiffré par RC6 on voit bien apparaître le premier champ de longueur : 0x39.
Le premier fichier est déchiffré avec les fonctions `sub_12284` et `sub_122F8` amorcées avec les données aléatoires présente dans le déchiffré RC6, il contient :

```
password for the zip file : !WooYouAreSuchAnAwesomeGuy!
```

Listing 46: Contenu du premier fichier

Le fichier suivant est un "Lorem ipsum", mais il est coupé au milieu. La suite du fichier doit certainement être dans une autre zone à forte entropie de l'image disque. En testant tous les offsets des parties à forte entropie, la suite du fichier est identifiée à l'offset 0x40000600.
8 fichiers sont trouvés, en cherchant plusieurs fois l'offset suivant. Le script `decrypt.py` présent en annexe permet de déchiffrer l'ensemble des fichiers :

```
$ python2 decrypt.py
write file extract/file-00.bin (size:57)
write file extract/file-01.bin (size:3093)
write file extract/file-02.bin (size:214577)
write file extract/file-03.bin (size:111156)
write file extract/file-04.bin (size:64480)
write file extract/file-05.bin (size:47607)
write file extract/file-06.bin (size:33749)
write file extract/file-07.bin (size:1200138)
$ file extract/ *
extract/file-00.bin: ASCII text
extract/file-01.bin: ASCII text, with very long lines
extract/file-02.bin: PDF document, version 1.5
extract/file-03.bin: JPEG image data, Exif standard: ...
extract/file-04.bin: Zip archive data, at least v2.0 to extract
extract/file-05.bin: JPEG image data, JFIF standard 1.01, ...
extract/file-06.bin: JPEG image data, JFIF standard 1.01, ...
extract/file-07.bin: PC bitmap, Windows 98/2000 and newer format, 800 x 500 x 24
```

Listing 47: Déchiffrement des fichiers

Les documents extraits sont les suivants :
— Un document texte contenant un mot de passe,
— Un document texte contenant un "Lorem ipsum",
— Un PDF intitulé "On the reception and detection of pseudo-profound bullshit",
— Une "Troll-face",
— Un fichier ZIP protégé par mot de passe,
— Une image de chat,
— Une autre image de chat,
— Une capture d'écran d'un PC windows XP au format BMP.

La solution de l'épreuve est trouvée dans le fichier ZIP en utilisant le mot de passe fourni dans le premier fichier :

```
$ unzip -P '!WooYouAreSuchAnAwesomeGuy!' extract/file-04.bin
Archive:  extract/file-04.bin
  inflating: 4.jpg
 extracting: key

$  xxd key
00000000: 0928 bde1 e3ed 8969 8632 dbff 4a23 1138  .(.....i.2..J#.8
```

Listing 48: Extraction de l'archive contenant la solution

FIGURE 14: 4.jpg

4.3 Video : Rétro-ingénierie d'un écran de veille Windows exfiltrant des données

Tiens, jeune padawan, voilà ton sujet de stage.
/ ! Attention / ! : ce stage est interdit aux épileptiques !

L'épreuve débute avec l'archive `video.zip` fournie par un personnage. Cette archive contient un binaire PE, une vidéo, et un fichier texte :

```
$ bsdtar -xvf ../video.zip
x Stage_anti_APT_chez_Airlhes/Airlhes_screensaver_setup.exe
x Stage_anti_APT_chez_Airlhes/Airlhes_CYBER_SECRET_possible_exfiltration.mp4
x Stage_anti_APT_chez_Airlhes/
x Stage_anti_APT_chez_Airlhes/mission.txt
$ cat Stage_anti_APT_chez_Airlhes/mission.txt
Bonjour, cher - quel est votre nom deja ? - stagiaire...
...
```

Listing 49: Extraction de l'archive

Le binaire PE est un installeur pour un écran de veille : `Airlhes_screensaver.scr`, une clef de registre est également installée : `Software/Airlhes/Screensaver/config/trajectories` avec une chaîne hexadécimale débutant par l'entête Zlib.
La vidéo est un film du mur opposé à un écran, celui-ci reflète les couleurs affichées à l'écran.
Au vu du texte de mission fournie dans l'archive, le but de l'épreuve semble donc de retrouver les informations de la clef de registre à partir des couleurs de la vidéo.
Les couleurs sont extraites à l'aide de VLC, le filtre "Scenes" permet d'exporter les images de la vidéo, pour ensuite en extraire les composantes principales les plus proches. 8 couleurs différentes sont identifiées.
La rétro-ingénierie permet d'isoler une fonction intéressante, chiffrant les données du registre :

```
else if ( dword_40C940 == 0xBA5571C )
{
  if ( dword_40F120 )
  {
    byte_7931C0 = (unsigned __int8)byte_7931C0 / 7u;
    v46 = (unsigned __int8)byte_7931C0 % 7u + 1;
    byte_7931E0 = v46;
    v45 = ((_BYTE)dword_793200 + v46) & 7;
    dword_793200 = v45;
    dword_40F120 = (dword_40F120 + 1) % 3;
    if ( !dword_40F120 )
    {
      pos = (pos + 1) % (unsigned int)*(&v52 - 6152);
      if ( !pos )
        dword_40C940 = 0xB003BA88;
    }
  }
  else
  {
    v44 = *(_BYTE *)(*(&v52 - 6153) + pos) ^ *(_BYTE *)(*(&v52 - 6177) + (unsigned __int8)pos);
    dword_40F120 = 1;
    byte_7931C0 = v44;
    byte_7931E0 = (unsigned __int8)v44 % 7u + 1;
    v45 = ((_BYTE)dword_793200 + byte_7931E0) & 7;
    dword_793200 = ((_BYTE)dword_793200 + byte_7931E0) & 7;
  }
  *(&v52 - 6172) = dword_405020[v45];
  *(&v52 - 6172) ^= 0x4FB78EB3u;
}
*(&v52 - 6175) = sub_402F70(&v52 - 6166);
```

FIGURE 15: sub_4032D0

Les couleurs semblent donc coder :
- La longueur de la chaîne présente dans la clef de registre
- La clef de registre elle-même

Chaque octet est codé avec une suite de 3 couleurs, pour résoudre l'épreuve ; il faut inverser la fonction et décoder les couleurs de la vidéo, et décompresser avec Zlib le résultat :

```
$ python decode_colors.py
5311371672ba0179fa3e918a83bedeb4
```

Listing 50: Calcul de la solution

5 Final


```
$ cat final.txt
Coucou !

Tu as presque reussi le challenge !
I01p1 y'4qe3553 z41y : 8Y6d5j9Vy88HUGHfGSKsJvqA@ffgvp.bet
```

Listing 51: final.txt

Dans le dernier plateau du jeu un personnage couronné, donne le précieu fichier final.txt. Cette aventure se termine avec une simple ligne de Python :

```
$ python2
>>> "8Y6d5j9Vy88HUGHfGSKsJvqA@ffgvp.bet".decode("rot13")
u'8L6q5w9Il88UHTUsTFXfWidN@sstic.org'
```

Listing 52: Solution du challenge

6 Conclusion

Encore une fois, le challenge SSTIC était passionnant par la diversité de ses épreuves, et les technologies exotiques rencontrées. Il m'a permis de découvrir une architecture qui m'était inconnue (ia-64) ainsi que les instructions TrueType. J'ai en revanche regretté le manque de stéganographie dans les épreuves finales de cette année.

7 Annexes

7.1 Scripts et solutions des épreuves

Ce fichier PDF est également un fichier ZIP, contenant l'ensemble des scripts utilisés pour résoudre les épreuves.

```
$ unzip SSTIC_2016_David_BERARD.pdf
```

Listing 53: Extraction des scripts

7.2 Les personnages et dialogues du jeu

	phrases	fichiers
	J'ai créé un fichier polyglotte au format pdf qui est aussi un tar, qui est aussi un doc, qui est aussi un jpg, qui est aussi un avi, qui est aussi un exe, qui est aussi un nfo, qui est aussi un odt, qui est aussi un tex, qui est aussi un divx ... ... qui est aussi un html, qui est aussi un pdb, qui est aussi un php, qui est aussi un swf, qui est aussi une dll, qui est aussi un htaccess, qui est aussi un ogg, qui est aussi un js ... ... qui est aussi un 3gp, qui est aussi un bat, qui est aussi un ppt, qui est aussi un ico, qui est aussi un mov, qui est aussi un txt, qui est aussi un wmv, qui est aussi un hlp, qui est aussi un wav, qui est aussi un gz, qui est aussi un xls, et aussi un pdf.	
	Hé, je suis docteur ingénieur moi. Tu ne me crois pas ? Tu veux voir ce que je sais faire ? — **CHOIX 1** : Okay... — **CHOIX 2** : Non, ça ira [**CHOIX 1**] Alors ?! Ça t'épate hein ? **une alert("XSS !") est exécutée** [**CHOIX 2**] Tant pis.	
	Mais qu'est-ce que tu fais là ?	
	Bravo !	
	Attention !! J'ai le mot de passe de la boucherie... Si je ne donne pas son nom aux journalistes mais que je donne 1 phrase qui permet de retrouver le nom de la boucherie-charcuterie ? légal ? — **CHOIX 1** : Je m en tape. — **CHOIX 2** : Si tu veux, je connais une experte mexicaine qui pourrait te répondre. [**CHOIX 1**] Comme quoi la vérité dérange ... [**CHOIX 2**] Merci, reviens me voir quand tu lui auras demandé.	
	Tu veux voir ma grosse antenne ? Ou tu veux sentir mon téléphone ? En plus je suis un bon coup : je compte double !	radio.zip
	C'est bon, passe. Par contre, fais attention à ce que tu vas trouver dans la cave... Il s'y passe des choses étranges.	
	Hé... Ça te dirait de développer un AFL pour Windows ? C'est un super stage payé 400€.	

	phrases	fichiers
	Bonjour ! Tu veux que je te parle de Photorec ?	
	Est-ce que tu as réussi l'épreuve Big ELF ? ... J'ai dû acheter 4 disques dur pour la finir ...	
	Est-ce que tu vas aux Assises ? Tu sais, c'est cyber important !	
	T'aurais pas vu mes lunettes et mon portable ?	
	Quoi, encore un chall ? Mais que fait la police ?	loader.zip
	On suspecte une machine d'être hantée par un fantôme mais cette fois ce n'est pas Casper. Tiens, voici une capture du champ électromagnétique... en gros une capture réseau. Tu pourras peut-être en tirer quelque chose !	SOS-Fant0me.zip
	Un dessin animé ! Un petit garçon avec une queue de singe. Bon, j'y vais !	
	Ce matin rien n'a changé Reflétant le soleil, Les chiffres me regardent. Attention ! Ce haïku compte double.	strange.zip
	Tu veux voir mes FeSes ?	usb.zip
	Ce crackme vient d'en bas. Il faudra être EFIcace pour relever ce dEFI, car le débuggueur manque de dEFInition pour ce type de ROM arrangée.	foo.zip

	phrases	fichiers
	Quand j'étais encore jeune, j'ai caché ma clé dans un instrument que j'ai ramené du Texas. Malheureusement, j'ai oublié comment la récupérer depuis...	calc.zip
	Tiens, jeune padawan, voilà ton sujet de stage. /! Attention /! : ce stage est interdit aux épileptiques !	video.zip
	Je me demande quel est le cadre juridique du challenge du SSTIC.	
	C'est moi, petit poussin !	
	C'est bon, tu peux passer.	
	Puthon, c'est de la pisse.	
	Oh ! Des lunettes et un portable. Ça doit appartenir à quelqu'un.	
	Salut Rémi ! Le python, c'est bon ?	
	Oh, un jukebox ! Jouer une musique au hasard ?	
	Ça y'est ... J'ai cassé AES ! Ça y'est ... J'ai cassé TOR ! Mince ... On a cassé Perseus !	
	Ring-3 for the Reversers under the sky, 7 processes for Beer Drinkers in their street of thirst, 16 bytes for the Warriors doomed to sigh, 1 modulus for the Dark Lord on his crypto quest In the Land of Windows where the Control Flows lie. One Ring to rule them all, One Ring to debug them, One Ring to crash them all, and in SSE bind them, In the Land of Windows where the Control Flows lie. Un anneau qui compte double...	ring.zip

	phrases	fichiers
		final.txt
	C’est vraiment très indigeste ce challenge...	huge.zip
	Je noie mon désespoir dans la boisson car mon épreuve a été refusée pour le challenge du SSTIC. Tiens, la voici !	crosswords.zip
	Salut voyageur ! Pour arriver au bout de ce challenge, il te faudra résoudre des épreuves. Tu auras le choix entre plusieurs épreuves à chaque niveau, donc choisis bien celles que tu préfères. Il suffit d’atteindre le nombre de points requis pour passer au niveau suivant. Chaque épreuve te fournira une clé de 16 octets, qu’il faudra donner au garde du niveau sous forme hexadécimale pour la faire prendre en compte. Si tu lui donnes suffisamment de clés, le garde te laissera passer. Tu rencontreras peut-être d’autres personnages bizarres en cours de route (toute ressemblance avec des personnes existantes serait purement fortuite), ne t’y attarde pas trop, sauf si tu aimes troller. Bonne chance !	
	Bar Mexicain : Le Cactus	
	Je m’appelle Nouveaulogiciel. Est-ce que tu vas release le code source ? T’aurais pas vu mon stagiaire ? J’ai un travail à finir.	
	Au nord-ouest du village, un groupe de gens ont besoin de ton aide. Tu devrais aller leur parler ! Ils te donneront des clés en échange. Sinon, il y a une taverne à côté du pont. Si tu veux aller boire un coup, va y faire un tour.	
	Ça ne se fait pas de faire les poches ! En plus il n’y a pas de clé dedans.	