Solution du challenge SSTIC 2016

Pierre Bienaimé

24 avril 2016

Table des matières

1 Introduction **2**

2 Stage 0 **2**
2.1 Pcap . . . 2
2.2 Rpgjs . . . 2

3 Stage 1 **3**
3.1 calc . . . 4
3.2 SOS-Fant0me . . . 7

4 Stage 2 **10**
4.1 foo . . . 11
4.2 huge . . . 13

5 Stage 3 **17**
5.1 strange . . . 17
5.2 video . . . 22

6 Victoire **28**

7 Conclusion **28**

1 Introduction

J'aime le challenge SSTIC !

Chaque année, c'est une formidable occasion de découvrir qu'on est complètement nul dans beaucoup de domaines. Et chaque année, c'est un motivateur puissant qui nous force à assimiler de nouvelles connaissances afin de devenir un peu moins nul [1].

L'édition 2016 est un jeu de rôle qui renferme 10 épreuves, réparties en 3 niveaux. Pour terminer le challenge, il est nécessaire d'en résoudre environ la moitié.

Cette solution du challenge SSTIC a vocation à être complétée et détaillée davantage lorsque j'aurai résolu chacune des 10 épreuves (s'il s'avère que j'ai le temps et les compétences suffisantes).

2 Stage 0

2.1 Pcap

Le fichier de départ du challenge est un pcap de 53,4 Mo. Une rapide analyse avec Wireshark montre qu'il contient le téléchargement du fichier `http://static.sstic.org/challenge2016/challenge.zip`. Ce fichier n'existe pas sur le serveur du SSTIC, il faut donc l'extraire du pcap. Pour ce faire, pas besoin de s'embêter à le reconstruire à la main, Wireshark sait déjà le faire en un clic : *File–>Export–>Objects–>HTTP*.

On peut s'interroger sur la raison pour laquelle le challenge soit ainsi inclus dans un pcap. Il est probable qu'il s'agisse d'un simple *filtre à noob*, afin d'éviter que le grand public ait directement accès à la foultitude de trolls contenus dans le RPG [2].

Easter egg : le participant attentif aura noté que dans le pcap, le fichier challenge.zip est prétendument téléchargé par la version 142 de Firefox, et qu'il est hébergé sur un serveur CERN httpd [3].

2.2 Rpgjs

Le fichier challenge.zip contient 130,8 Mo de sources d'un RPG codé en Javascript. Le jeu de rôle utilise le framework **rpgjs** [4], on peut y jouer à l'aide d'un navigateur web récent, et il est absolument fantastique :)

Les concepteurs du challenge SSTIC 2016 se sont donnés beaucoup de mal, et le résultat en vaut clairement la peine. Nous sommes ramenés en enfance, dans un RPG style Zelda, accompagné de son lot de musiques 8bit qui saura combler les mélomanes les plus exigeants [5]. Notre personnage croise plusieurs PNJ. Certains sont là pour nous

1. Phrase d'accroche d'occasion. Peu servie. Bon état général.
2. Certains étant assez... osés :)
3. Le tout premier serveur web ! Qui ici est cependant dans la *récente* version 3.0A, datant de 1996
4. `https://github.com/RSamaium/RPG-JS`
5. Hum...

donner des épreuves à résoudre (ils nous font télécharger un fichier). D'autres sont là uniquement pour troller. Tous les *running gag* du SSTIC y passent.

À chaque niveau, un gardien bloque l'accès et nous demande de rentrer des clés, qui sont obtenues en résolvant des épreuves. Le code source du RPG n'a pas été analysé en profondeur, mais de la cryptographie est (naturellement) utilisée pour chiffrer les niveaux suivants, empêchant les petits malins de pouvoir patcher le jeu et brûler les étapes.

Il est à noter que l'intégralité du challenge se déroule hors ligne. Ceci présente l'avantage que dans 10 ans, il sera encore possible à l'aide du simple fichier pcap de départ, de jouer au RPG et de résoudre les 10 épreuves qu'il contient.

La majorité des épreuves sont axées sur du *reverse engineering*, ce qui implique que la méthodologie de résolution n'est pas toujours très facile à retranscrire dans un document. Quand la difficulté et l'intérêt d'une épreuve est de parvenir à comprendre ce que fait un programme, je peine à trouver la meilleure façon d'exposer ma solution. Il y a souvent peu de code à écrire. Pas forcément d'astuce particulière à partager. Et je pense qu'il est vain de remplir ce rapport de pages d'assembleur en tentant de décrire le cheminement intellectuel chaotique qui a permis d'aboutir à la compréhension finale. Je vais donc assez peu me focaliser sur la rétroconception en tant que telle et sur la description minutieuse de ce que fait chaque programme. Je vais plutôt essayer d'expliquer mes choix.

À savoir que lorsque je m'attaque à un challenge, je garde toujours mon credo en tête : ***« Il n'est pas nécessaire de tout comprendre. Juste de comprendre suffisamment pour pouvoir bruteforcer »***. Ce qui aboutit souvent à des solutions plus pragmatiques qu'élégantes :)

3 Stage 1

Le premier niveau propose 3 épreuves à résoudre et pour passer le gardien du niveau 1, il faut récolter deux points.

calc : un programme pour calculatrice TI83+ (1 point)

SOS-Fant0me : le pcap du trafic généré par une machine infectée par un malware (1 point)

radio : la capture d'ondes radio émises par un téléphone portable (2 points)

3.1 calc

```
$ file SSTIC16.8xp
SSTIC16.8xp: TI-83+ Graphing Calculator (program)
```

Le fichier de départ est un programme pour calculatrice TI83+, au format binaire 8xp. Envahi par la nostalgie, je me remémore alors mon tout premier programme informatique, écrit sur une TI82.

Je commence par faire un tour d'horizon des outils capables de manipuler ce format : émulateur, débuggeur, désassembleur, etc. Puis je trouve le projet Github **basically-ti-basic**[6], un programme Python qui décompile des fichiers 8xp afin d'obtenir du TI-Basic, ce qui correspond exactement à ce que je veux faire. L'idée de relire du TI-Basic toutes ces années après me suscite une certaine curiosité.

```
$ basically-ti-basic -d -i SSTIC16.8xp -o SSTIC16.txt
```

On obtient alors les sources d'un programme en TI-Basic d'envrion 200 lignes, dont voici le début :

```
ClrHome
Goto 1

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
```

6. https://github.com/thenaterhood/basically-ti-basic

```
Lbl 3
sub(Str1,,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,,Str1)-1)->Str1
While ,Str1)<32
"0"+Str1->Str1
End
Goto 0

Lbl 4
" "->Str3
For I,1,,Str1)
If "1"=sub(Str1,,Str1)-I+1,1) xor "1"=sub(Str2,,Str1)-I+1,1):Then
"1"+Str3->Str3
Else
"0"+Str3->Str3
End
End
sub(Str3,1,,Str3)-1)->Str3
Goto 0
```

Bon, ok... mon tout premier programme était *un poil* moins compliqué. Mais peu à peu les souvenirs reviennent et l'architecture du programme se dégage. Il commence par un gros switch qui va permettre de gérer le flot d'exécution et de créer des simili-fonctions en TI-Basic. Ensuite, les premiers labels correspondent à des fonctions utilitaires : int2bin, xor, bin2int, int2hex, etc. Enfin, on trouve le corps du programme : il demande à l'utilisateur de rentrer un nombre. Des opérations sont appliquées sur celui-ci. On vérifie que le résultat obtenu est 3298472535. Si oui, la clé est affichée.

On remarquera quand même que la façon qu'a le programme de manipuler des octets est assez horrible et ressemble à du gros bricolage. Il transforme les entiers en binaire, mais les stocke sous la forme de chaînes de caractères. Puis les opérations (xor, shift, masque, etc) sont appliquées sur ces chaînes de caractères. Mais il faut croire que le langage TI-Basic ne permet pas de faire beaucoup plus propre. Quand on fait abstraction de tous les hacks utilisés, il ne reste au final que très peu de code utile. On distingue une boucle qui manipule des données d'entrée (à base de Xor) puis se sert du résultat comme d'un index dans un tableau de 256 entiers. Ça ressemble à un algorithme de cryptographie, et avant d'aller plus loin, il convient de googler quelques une des constantes de ce tableau.

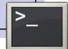
Bingo, il s'agit *exactement* des constantes utilisées par le CRC32. Et quand on compare notre code avec d'autres implémentations de l'algorithme CRC32 [7], on retrouve vite tous nos petits. Et c'est même presque décevant, car il s'agit bien d'une implémentation d'un CRC32 en TI-Basic sans customisation. Je range donc mon code TI-Basic, et c'est l'occasion de ressortir d'un tiroir mon framework **ssticy** qui commençait à sérieusement

7. https://gist.github.com/azat/2762138

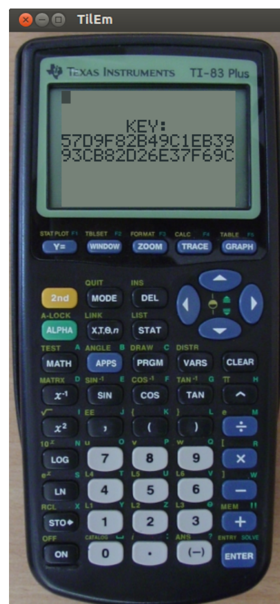
prendre la poussière. On cherche quel entier (de 32 bits maximum) a un CRC32 qui vaut 3298472535.

```python
import zlib
import struct
from ssticy import bruteforce, ALL

def check(p):
    return (zlib.crc32(p) & 0xffffffff) == 3298472535

c = bruteforce(check, length=range(1,5), charset=ALL, multiproc=4, verbose=True)
print "Le code est %i" % struct.unpack("<I", c)[0]
```

```
$ python calc.py
Trying with length = 1
Trying with length = 2
Trying with length = 3
Trying with length = 4
Found : '\x16\x1cW\x05'
Le code est 89594902
```



Lorsque le code demandé par le programme TI83+ est le bon, il sert de graine pour initialiser le générateur de nombres pseudo-aléatoires de la calculatrice. Les 16 premiers octets de random correspondront à la clé de validation de l'épreuve. Ne sachant pas quel algorithme de PRNG est utilisé par la TI83+, le plus simple reste encore d'émuler le programme, de rentrer le code 89594902 et de laisser la clé s'afficher à l'écran. Pour cela, j'ai utilisé **TilEm**.



```
calc : 57 D9 F8 2B 49 C1 EB 39 93 CB 82 D2 6E 37 F6 9C
```

3.2 SOS-Fant0me

Le fichier de départ est le pcap du trafic généré par une machine infectée par un malware. On s'aperçoit rapidement qu'il s'agit d'un vrai malware nommé Gh0st RAT. Il est facile à identifier car il communique avec son C&C sur le port 80 via un protocole custom dont chaque paquet commence par `Gh0st`.

Ce malware a déjà été analysé en détail par de nombreux chercheurs, un papier blanc de McAfee [8] explique même le format du protocole de communication. Pour ce niveau, il n'y a donc pas grand-chose à faire : suivre la documentation et écrire un parseur minimaliste pour le protocole Gh0st.

Pour extraire la charge utile du pcap et parser le protocole, pas besoin de sortir l'artillerie lourde, à base de Construct, de Hachoir ou de dissecteur Scapy. Je me suis contenté de deux lignes de Scapy pour récupérer les données Gh0st brutes, et j'ai fait le reste à la main. Il n'est pas non plus nécessaire de supporter l'ensemble du protocole Gh0st : on code les commandes au fur et à mesure qu'on les rencontre.

Voici le script Python qui parse le pcap et qui liste les communications du client (l'opérateur du RAT) et du serveur (la machine infectée).

```
1  import struct
2  from scapy.all import *
3  from ssticy import strings
4
5  def parse(ghost_data):
6      for d in ghost_data.split("Gh0st"):
7          if len(d) > 8:
8              raw_length, = struct.unpack("<I", d[0:4])
9              payload_length, = struct.unpack("<I", d[4:8])
10             payload = d[8:].decode("zlib")
11             assert raw_length == len(d) + len("Gh0st")
12             assert payload_length == len(payload)
13             yield payload
14
15 def parse_client(client_data):
16     for payload in parse(client_data):
17         cmd = ord(payload[0])
18         if cmd == 0:
19             print "COMMAND_ACTIVATED"
20         elif cmd == 1:
21             print "COMMAND_LIST_DRIVE"
22         elif cmd == 2:
23             print "COMMAND_LIST_FILES", payload[1:-1]
24         elif cmd == 3:
25             print "COMMAND_DOWN_FILES",  payload[1:-1]
26         elif cmd == 7:
27             print "COMMAND_CONTINUE"
28         elif cmd in [0x1E, 0x1F, 0x23]:
29             print "USELESS_COMMAND_0x%X ?" % cmd
30         else:
31             raise Exception("Unsupported command: 0x%X (%r)" % (cmd, payload))
32
33 def parse_server(server_data):
34     last_token = None
35     filename = None
36     keyboard = bytearray()
```

8. http://www.mcafee.com/us/resources/white-papers/foundstone/wp-know-your-digital-enemy.pdf

```
37      file_data = bytearray()
38      for payload in parse(server):
39          token = ord(payload[0])
40          # Keyboard Data
41          if token == 124:
42              keyboard += payload[1:]
43              continue
44          elif keyboard:
45              print "TOKEN_KEYBOARD_DATA", str(keyboard)
46              keyboard = bytearray()
47
48          # File download
49          if filename and size:
50              dwoffset, = struct.unpack("<I", payload[5:9])
51              payload = payload[9:]
52              file_data[dwoffset:] = payload
53              size -= len(payload)
54              first_chunk = False
55              if size <= 0:
56                  with open(filename, "wb") as f:
57                      f.write(str(file_data))
58                  print "--> %s file saved" % filename
59                  filename = None
60              continue
61
62          if token == 102:
63              print "TOKEN_LOGIN", strings(payload)
64          elif token == 103:
65              print "TOKEN_DRIVE_LIST", strings(payload)
66          elif token == 104:
67              print "TOKEN_FILE_LIST", strings(payload)
68          elif token == 105:
69              size, = struct.unpack("<I", payload[5:9])
70              filename = payload[9:-1].rsplit("\\")[-1]
71              first_chunk = True
72              print "TOKEN_FILE_SIZE (%i bytes) : %s" % (size, filename)
73          elif token == 123:
74              print "TOKEN_KEYBOARD_START"
75          elif token == 125:
76              print "TOKEN_PSLIST (%i length)" % len(payload)
77          else:
78              raise Exception("Unsupported token: %i (%r)" % (token, payload))
79
80  if __name__ == '__main__':
81      pkt = PcapReader("SOS-Fant0me.pcap").read_all()
82      client = "".join(p["Raw"].load for p in pkt if p["TCP"].sport == 80 and "Raw" in p)
83      server = "".join(p["Raw"].load for p in pkt if p["TCP"].dport == 80 and "Raw" in p)
84
85      print "parsing Gh0st client data :"
86      parse_client(client)
87
88      print "\nparsing Gh0st server data :"
89      parse_server(server)
```

```
$ python ghost.py
parsing Gh0st client data :
COMMAND_ACTIVATED
USELESS_COMMAND_0x23 ?
USELESS_COMMAND_0x1F ?
USELESS_COMMAND_0x1E ?
COMMAND_LIST_DRIVE
COMMAND_LIST_FILES C:\
COMMAND_LIST_FILES C:\Users\
COMMAND_LIST_FILES C:\Users\sstic\
COMMAND_LIST_FILES C:\Users\sstic\Documents\
COMMAND_LIST_FILES C:\Users\sstic\Documents\Challenge SSTIC 2016\
COMMAND_DOWN_FILES C:\Users\sstic\Documents\Challenge SSTIC 2016\how_to_rule_the_world.txt
COMMAND_CONTINUE
COMMAND_CONTINUE
COMMAND_DOWN_FILES C:\Users\sstic\Documents\Challenge SSTIC 2016\visio_stage2.mp4
COMMAND_CONTINUE
[...]
COMMAND_CONTINUE
COMMAND_LIST_FILES C:\Users\sstic\Documents\Challenge SSTIC 2016\Stage 1\
COMMAND_DOWN_FILES C:\Users\sstic\Documents\Challenge SSTIC 2016\Stage 1\sstic2016-stage1-solution.zip
COMMAND_CONTINUE
COMMAND_CONTINUE
COMMAND_LIST_DRIVE
```

Le client liste des fichiers qui se trouvent sur la machine infectée, et demande le téléchargement de trois d'entre eux : un fichier texte, une vidéo, et une archive zip protégée par un mot de passe.

```
$ python ghost.py
[...]

parsing Gh0st server data :
TOKEN_LOGIN ['SSTIC-PC1']
TOKEN_PSLIST (1172 length)
TOKEN_KEYBOARD_START
TOKEN_KEYBOARD_DATA [2016/02/27 - 23:14] New message: [SSTIC 2016/Challenge] Stage 1Salut !
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 passeCyb3rSSTIC_2016
TOKEN_DRIVE_LIST ['Local Disk', 'CD Drive']
TOKEN_FILE_LIST ['autoexec.bat', 'config.sys', 'PerfLogs', 'Program Files', 'Windows']
TOKEN_FILE_LIST ['Public']
TOKEN_FILE_LIST ['Contacts', 'Desktop', 'Documents', 'Downloads', 'Favorites', 'Pictures',
 'Saved Games', 'Searches', 'Videos']
TOKEN_FILE_LIST ['Challenge SSTIC 2016']
TOKEN_FILE_LIST ['how_to_rule_the_world.txt', 'Stage 1', 'visio_stage2.mp4']
TOKEN_FILE_SIZE (2759 bytes) : how_to_rule_the_world.txt
--> how_to_rule_the_world.txt file saved
TOKEN_FILE_SIZE (193968 bytes) : visio_stage2.mp4
--> visio_stage2.mp4 file saved
TOKEN_FILE_LIST ['sstic2016-stage1-solution.zip']
TOKEN_FILE_SIZE (234 bytes) : sstic2016-stage1-solution.zip
--> sstic2016-stage1-solution.zip file saved
```

La machine infectée répond aux requêtes du client et envoie les fichiers en question. Elle envoie également ce que l'utilisateur tape au clavier. On récupère ainsi le mot de passe de l'archive zip : Cyb3rSSTIC_2016. L'archive contient la clé de validation de l'épreuve.

À noter que la vidéo est plus compliquée à récupérer que les autres fichiers, car elle est découpée en plusieurs morceaux. Pour les reconstituer je me suis retrouvé à lire le code source du malware Gh0st pour comprendre comment fonctionne le `TRANSFER_MODE_JUMP`. Mais ça en valait clairement la peine, puisque cette vidéo est un rickroll :)

`SOS-Fant0me : 36 8B E8 C1 CC 7C C7 0C 22 45 03 09 34 30 1C 15`

4 Stage 2

Un panonceau à l'entrée du niveau 2 nous invite à envoyer un message à l'adresse UkQhxwnHoZIIKw9lPGK5BNLg@sstic.org si l'on souhaite informer les organisateurs de notre avancée.

Lorsqu'on se promène dans ce niveau, on constate que les trolls sont toujours là. On trouve, cachés dans un coin, une paire de lunettes et un portable. Notre personnage les ramasse. Et quand on les rapporte à leur propriétaire (un PNJ situé dans la taverne du niveau 1), il nous remercie de lui avoir évité un aller-retour Paris Rennes. Une quête inspirée de faits réels :)

Le second niveau propose 3 épreuves à résoudre :

foo : une application EFI (1 point)

huge : un binaire ELF de 128 To (1 point)

loader : un binaire Windows que je n'ai pas encore pris le temps de regarder (1 point)

Pour passer le gardien du niveau 2, il faut récolter deux points.

4.1 foo

```
$ file foo.efi
foo.efi: PE32 executable (DLL) (EFI application) EFI byte code, for MS Windows
```

Le fichier de départ est une application EFI, qui est donc prévue pour être chargée lors de la phase DXE d'une séquence de boot UEFI. L'originalité est qu'elle est compilée en EBC (EFI byte code) alors que classiquement, on s'attend plutôt à rencontrer ici du x86_64. Je ne savais même pas qu'il existait un byte code EFI. Cependant, IDA sait le désassembler, et les instructions sont assez explicites. Une fois familiarisé avec l'EBC, il s'avère que cette application EFI est architecturée exactement de la même manière qu'un driver DXE classique, format qui est très bien documenté et structuré.

Le PNJ qui nous donne le challenge nous prévient qu'il va falloir s'accrocher car il existe peu d'outils. En effet, comme l'EBC est assez peu répandu, la plupart des outils d'analyse automatique de BIOS EFI risquent de ne pas fonctionner. De plus, il sera certainement assez compliqué de mettre en place un environnement qui permettra d'exécuter l'application EFI et de la débugger.

Cependant, une application DXE reste facile à analyser de manière statique. Il va simplement falloir le faire *à la main*. L'analyse se fait en deux étapes. On commence par documenter tout ce qui peut l'être, résoudre tous les *call* et renommer les GUID. On essaye ensuite de comprendre ce que fait l'application.

Dans le binaire PE d'un driver ou d'une application DXE, il n'y a pas d'imports. Tous les appels de fonction sont dynamiques. En l'occurrence, en EBC, on aura ici par exemple des `CALL32EXa [R7+4]`. Pour savoir ce qu'est R7, il faut remonter jusqu'au point d'entrée du programme. Chaque application DXE est appelée avec comme 2ème argument

un pointeur vers EFI_SYSTEM_TABLE. Cette table contient, entre autres, des pointeurs vers EFI_BOOT_SERVICES et EFI_RUNTIME_SERVICES, deux structures qui contiennent toutes les fonctions dont on peut avoir besoin. On y trouve des équivalents de malloc, free, memcpy, memset, [...] ainsi que des moyens d'utiliser du code contenu dans d'autres modules qui ont été chargés avant nous (qu'on appelle des protocoles). Les protocoles sont identifiés par des GUID, qui sont eux aussi bien documentés dans la norme EFI.

Une fois la phase de documentation terminée, on constate que l'application foo.efi ne fait pas grand-chose. Elle fait des appels à EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.OutputString pour afficher des chaînes de caractères sur la sortie standard. Elle utilise EFI_LOADED_IMAGE_PROTOCOL.LoadOptions pour récupérer les 32 caractères hexa de clé que l'utilisateur devra passer en paramètre de l'application. Elle manipule le GUID correspondant à EFI_AUTHENTICATION_INFO_PROTOCOL, mais c'est de la poudre aux yeux. Ce GUID est xoré avec des constantes avant d'être utilisé. Le protocole qui sera effectivement chargé sera EFI_DECOMPRESS_PROTOCOL.

L'application décompresse 0x47 octets contenus dans la section *.data* du binaire et elle en extrait un secret. Des opérations sont effectuées sur la clé rentrée par l'utilisateur, et si cette clé est valide, le résultat de ces opérations sera égal au secret. Ainsi, toute cette épreuve a pu être résolue en statique. Dans un premier temps, on extrait les données compressées et on décompresse le secret. J'ai eu du mal à trouver dans la documentation les différents algorithmes de compression utilisés par EFI. Heureusement, j'ai rapidement mis la main sur le projet Github **EfiCompressor** [9] qui fait ça très bien, et qui m'a permis d'éviter de trop réfléchir :)

L'algorithme de modification de la clé a été retranscrit tel quel en Python, sans vraiment faire l'effort d'essayer de le comprendre ou de l'inverser. Comme l'algorithme s'applique à chaque octet de la clé séparément, celle-ci peut être bruteforcée instantanément. Pour casser les 16 octets de la clé, il faudra tester au maximum 16*256 = 4096 possibilités.

Pour minimiser la quantité de code à écrire, c'est l'occasion d'utiliser l'option *incremental* de mon outil **ssticy.bruteforce**. À chaque fois qu'un octet de clé est trouvé, l'outil passe automatiquement à l'octet suivant. Voici le code Python utilisé pour décompresser le secret et bruteforcer la clé :

```
1  import EfiCompressor
2  from ssticy import bruteforce, ALL
3
4  compressed_buffer = "3f0000005c000000003c4c8d823302ed6400b717307da812af1b021dfab86124"+
5      "fe3b24f51fccce8ba7738572726a518f09c1d4b10c7ba3a1b3cf078fcd9d553475ded963832000"
6
7  def key_byte_stuff(key_byte, index):
8      k = ord(key_byte)
9      k_save = k
10     index = index % 8
11     k = k >> index
12     k_save = k_save << (8-index)
13     result = (k | k_save) ^ 0xFFFFFFFF
14     return chr(result & 0xFF)
15
16 def check(key):
17     index = len(key) - 1
```

9. https://github.com/mjg59/python-eficompressor

```
        if index >= 16:
            return False
        return key_byte_stuff(key[-1], index) == secret_data[index]

if __name__ == '__main__':
    data = EfiCompressor.UefiDecompress(compressed_buffer.decode("hex"), 1000)
    secret = str(data).decode("utf16")
    print secret
    secret_data = secret[13:-1].decode("hex")
    print "Bruteforcing key..."
    bruteforce(check, charset=ALL, length=1, incremental=True)
```

```
$ python foo.py
secret data: cb41dcb1d89746705a7fe998f11acce7

Bruteforcing key...
Found partial result: 34
Found partial result: 347d
Found partial result: 347d8c
Found partial result: 347d8c72
Found partial result: 347d8c7272
Found partial result: 347d8c72720d
Found partial result: 347d8c72720d6e
Found partial result: 347d8c72720d6ec7
Found partial result: 347d8c72720d6ec7a5
Found partial result: 347d8c72720d6ec7a501
Found partial result: 347d8c72720d6ec7a50158
Found partial result: 347d8c72720d6ec7a501583b
Found partial result: 347d8c72720d6ec7a501583be0
Found partial result: 347d8c72720d6ec7a501583be0bc
Found partial result: 347d8c72720d6ec7a501583be0bccc
Found partial result: 347d8c72720d6ec7a501583be0bccc0c
No new result. Final result is: 347d8c72720d6ec7a501583be0bccc0c
```

4.2 huge

Le fichier de départ est l'archive huge.tar, qui ne fait que 100Ko mais qui contient un binaire ELF de 128 To ! Cette épreuve était très intéressante, puisqu'elle nous force à trouver des réponses à un problème très concret (mais qu'on n'aurai jamais pensé à se poser) : comment faire pour analyser un binaire qui est trop gros pour pouvoir être stocké sur son système de fichier ? Il parait (initialement) impossible de l'exécuter, encore moins de le débugger.

Les entêtes du fichier tar nous apprennent que le binaire qu'il contient est un fichier de type *sparse*. Il commence par une table qui liste les offsets auxquels se trouveront les données du fichier, une fois celui-ci décompressé [10]. Tout le reste, ce sont des zéros. Pour analyser Huge, je vois donc deux chemins bien différents. Soit tout faire en statique, en faisant en sorte de charger les données aux bonnes adresses. Soit mettre en place un environnement de travail qui supporte les fichiers *sparse* (par exemple en utilisant

10. Je ne suis pas bien sûr du terme exact qu'il faut employer dans ce cas...

le système de fichier btrfs) pour pouvoir l'exécuter, et espérer trouver des outils qui permettront de l'analyser. J'ai préféré commencer par tenter ma chance en statique, en espérant que ça soit suffisant.

J'ai donc écrit un script IDAPython qui parse la *SparseMap*, et qui charge chaque morceau de code à sa bonne adresse virtuelle. Pour ce faire, je pense qu'on est obligé de créer autant de segments qu'il y a de blobs de code (à savoir 23). Or, dans l'ELF, il n'y a que 3 segments. Ainsi, on obtient des effets rigolos quand le programme arrive à la fin d'un segment dans IDA : il exécute des Tera octets de '0', jusqu'à arriver au début du segment suivant :). En effet, en x86_64, "00 00" est une instruction valide et correspond à `add [rax], al`.

Voici le script IDAPython utilisé pour charger Huge :

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import idc
import os

class Segment:
    def __init__(self, offset, virtaddr, size):
        self.offset = offset
        self.virtaddr = virtaddr
        self.size = size

    def __contains__(self, offset):
        return self.offset <= offset < self.offset+self.size

    def off2virt(self, offset):
        o = offset - self.offset
        if o < self.size:
            return self.virtaddr + o
        raise Exception("Bad offset")

segments = [
    Segment(0x1000, 0x2b0000000000, 0x1ef000000000),
    Segment(0x2afffffe1000, 0x49f000000000, 0x161000000000),
    Segment(0x49effffe1000, 0x20000, 0x2afffffe0000)
]

class SparseMap:
    def __init__(self, raw_map, data):
        self.data = data
        self.map = []
        m = raw_map.split("\n")[1:-1]
        while len(m) >= 2:
            offset = int(m.pop(0))
            size = int(m.pop(0))
            self.map.append((offset, size))

def load_files(sparse_map):
    total_size = 0
    for offset, size in sparse_map.map:
        chunk = sparse_map.data[total_size:total_size+size]
        total_size += size
        for s in segments:
            if offset in s:
                virt = s.off2virt(offset)
                fname = "blob_" + hex(virt).strip("L")
                with open(fname, "wb") as f:
                    f.write(chunk)
                print "Writing file %s (%i bytes)" % (fname, len(chunk))
                idc.AddSeg(virt, virt+size, 0, 2, 0, 0)
                idc.LoadFile(fname, 0, virt, size)
```

```
52                 os.remove(fname)
53
54 if __name__ == '__main__':
55     with open("huge.tar", "rb") as f:
56         data = f.read()
57     sparse = data[0x600:0x800]
58     elf = data[0x800:]
59     m = SparseMap(sparse, elf)
60     load_files(m)
```

Ensuite, la rétroconception est assez simple à faire. Le programme saute dans des morceaux de code qui vérifient des sous-parties de la clé passée en paramètre. Si le morceau de clé est valide, l'exécution continue vers d'autres vérifications, sinon le programme fait un exit(42). 7 tests sont ainsi enchaînés. La plupart sont triviaux, le premier vérifie par exemple directement que le 1er octet est 0x29. Le 4ème test est amusant. Le 7ème est sadique.

Dans le 4ème test, le 2ème octet de la clé est utilisé comme index pour sauter dans une tableau d'octets nuls. On a donc la main sur le nombre d'instructions `add [rax], al` qui vont être exécutées, et il faut sortir la calculatrice afin de savoir où sauter. `al` dépend de `rsp` et on peut déterminer statiquement qu'il vaudra toujours 3. `[rax]` est initialisé à 0 et devra valoir 0x65 pour valider le test. On devra donc exécuter 119 fois l'instruction `add [rax], al`, car (119*3)&0xff == 0x65. L'octet de clé est donc (256-119) == 0x89. Enfin... je crois :) Car comme tout se fait en statique, je commence à douter. Une erreur est vite arrivée, et elle sera alors très compliquée à débugger.

Le 7ème et dernier test vérifie les 4 octets de la clé qui nous manquent. Il utilise des instructions assez obscures sur les nombres flottants, et sur leurs flags de statut. À cet instant, j'ai un gros moment d'hésitation. Soit je dois éplucher la documentation x86_64 concernant les nombres flottants, poursuivre mon analyse en statique et croiser les doigts pour ne m'être trompé nulle part. Soit je peux partir du principe que 4 octets, c'est peu, et ça peut se bruteforcer. Seulement, pour réaliser ce bruteforce, il faut auparavant trouver et reverser la partie du RPG en Javascript qui s'occupe de chiffrer les niveaux. Quel choix sera le plus rapide ? Quel choix sera le plus intéressant ? Aller... c'est l'occasion de comprendre comment fonctionnent les nombres flottants.

Je comprends donc que ce 7ème test manipule un nombre flottant codé sur un `tbyte` (10 octets), qui suit le format *Extended Precision* : un bit de signe, 15 bits d'exposant et 64 bits de mantisse, l'exposant étant biaisé de 16383. Voilà qui n'est pas *ultra* simple. Je comprends ensuite que ce flottant doit résoudre l'équation cos(x) = x (à une petite erreur de précision près). Sachant que les 4 octets de poids fort de la mantisse vont être xorés avec la sous-clé, il est possible de résoudre cette équation et d'en déduire la sous-clé.

Tout d'abord, je cherche une approximation du nombre flottant compris entre 0 et 1 qui résout l'équation cos(x) = x. La manière propre serait de faire de la dichotomie, mais je me suis contenté de bruteforcer les 12 chiffres situés après la virgule. On obtient **x** qui vaut environ 0.739085133215.

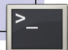
Ensuite, l'exposant du tbyte, sur lequel je n'ai pas le contrôle, vaut 0x3FFE == 16382.

Comme il est biaisé de 16383, la mantisse sera multipliée par 0.5. Il faut donc s'arranger pour qu'elle vaille x*2, soit environ 1.47817026643.

Enfin, il reste à convertir le flottant x*2 en mantisse de 64 bits, afin de déterminer quelle valeur de sous-clé permettra de transformer correctement le tbyte. Voici le script Python utilisé pour réaliser toutes ces étapes.

```
1  import math
2  import struct
3  from ssticy import bruteforce, NUM
4
5  def check(i):
6      if len(i) > 12:
7          return False
8      if i.endswith("9"):
9          return True
10     f = float("0." + str(int(i)+1))
11     return f > math.cos(f)
12
13 def float2significand(f):
14     b=""
15     while f:
16         if f > 1:
17             b += "1"
18             f -= 1
19         else:
20             b += "0"
21         f = f*2
22         if len(b) >= 64:
23             break
24     return int(b, 2)
25
26 if __name__ == '__main__':
27     f = bruteforce(check, charset=NUM, incremental=True, multiproc=1, verbose=False)
28     f = float("0."+f)
29     print "Approx of cos(x) == x :", f
30     m = float2significand(f*2)
31     print "Significand 0x%016x obtained for x*2 (env. %s)" % (m, str(f*2))
32     current_m = 0x24b878381e716dcb
33     print "The current significand is", hex(current_m)
34     key = struct.pack("<I", (current_m ^ m) >> 32)
35     print "So the subkey is 0x" + key.encode("hex")
```

```
$ python float.py
Approx of cos(x) == x : 0.739085133215
Significand 0xbd34aeec1e4437ff obtained for x*2 (env. 1.47817026643)
The current significand is 0x24b878381e716dcb
So the subkey is 0xd4d68c99
```



La clé, maintenant complète, est xorée avec CCFDCBC5B2A9E62B0D7E87370E2E4F19 et le résultat est écrit sur la sortie standard, en précisant qu'il s'agit de la clé finale de l'épreuve. Je donne donc ma clé au gardien, pas complètement rassuré, car la moindre erreur dans une des 7 étapes sera horrible à retrouver, et que je n'ai pas très envie de tout recommencer depuis le début. Ouf ! Du premier coup :)

```
huge : E5 74 B5 14 66 7F 6A B2 D8 30 47 BB 87 1A 54 F5
```

5 Stage 3

Le panonceau à l'entrée du niveau 3 nous indique que l'adresse de validation intermédiaire (et facultative) est RkrjBeyqFzsQApQhUbPvvTmJ@sstic.org.

Le troisième niveau propose 4 épreuves à résoudre :

video : une exfiltration de données via un écran de veille Windows malveillant (1 point)

usb : une épreuve que je n'ai pas encore eu le temps de regarder (1 point)

strange : un programme pour architecture IA-64 (Itanium) (2 points)

ring : 7 processus Windows qui se débugguent entre eux (2 points)

Pour passer le gardien du niveau 3, il faut récolter deux points.

5.1 strange

```
$ file a.out
a.out: ELF 64-bit LSB executable, IA-64, version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.4.0, stripped
```

Le fichier de départ est un ELF de 5Mo compilé pour l'architecture IA-64 des processeurs Itanium. Il est accompagné d'un fichier nommé `196`, qui contient 17 Mo (!) de données non identifiées. Voilà qui n'est pas très encourageant :)

Les processeurs Itanium, conçus par Intel, étaient censés être les processeurs du futur. Mais commercialement, ils ont fait un gros flop. J'aime assez la citation de Donald Knuth à ce propos [11]. D'ailleurs, avant ce challenge, je n'avais jamais entendu parler d'Itanium...

Le gros morceaux de cette épreuve consiste à prendre en main l'architecture IA-64, et à apprendre à la reverser. Car même si IDA sait le désassembler, le résultat est assez déroutant au début. Le plus troublant se passe au niveau de la convention d'appel : les registres utilisés pour passer des arguments à une fonction ne sont pas les mêmes entre l'appelée et l'appelante. Et ils changent pour chaque fonction. J'ai également eu assez peur en voyant toutes les instructions qui manipulent des nombres flottants, surtout qu'en IA-64, les flottants sont stockés sur 82 (!?!) bits. Cependant, une fois familiarisé avec Itanium, cette épreuve n'est pas si indigeste qu'elle en a l'air.

Le programme attend en argument un chemin vers une image au format P2 (Version ASCII du format Portable Graymap). Cette image doit avoir comme dimension 640x20 pixels, et ne contenir que des pixels noirs et blancs. L'image est découpée en 32 vignettes de 20x20 pixels. Pour chacune d'elle, des vérifications sont effectuées en utilisant notamment les données du fichier 196 (qui d'ailleurs s'avère contenir 17Mo de nombres flottants. Cette fois-ci, ce sont des doubles sur 8 octets... il y en a pour tous les goûts).

Tout d'abord, pour être valide, chaque vignette doit répondre à trois critères :
- La 1ère ligne doit être vide (pixels blancs)
- La dernière ligne doit au moins contenir un pixel noir
- Sur chaque ligne, les pixels noirs doivent être centrés

Si ces critères sont respectés, la vignette passe dans 160 fonctions affreuses de plus de 6000 instructions chacune. Ces fonctions se ressemblent beaucoup et manipulent des nombres flottants. Je ne veux pas savoir ce qu'elles font. Vraiment. Je les étudie donc de loin, en boîte noire. Elles semblent utiliser des flottants venant de 196, mais au final, elles génèrent en sortie uniquement 4 nombres flottants. Si ces 4 nombres sont inférieurs à 0.15, la vignette est valide et le programme passe à la suivante. Sinon, il affiche *fail*.

Je comprends qu'il s'agit d'une sorte d'OCR. Le tableau de 4 flottants est probablement un vecteur de caractéristiques. Typiquement, il doit représenter des densités de pixels ou des profils qui vont permettre de différencier des caractères entre eux.

À ce stade, le plan est donc de réussir à exécuter le programme, et de lui passer en entrée des images P2, dans l'espoir de comprendre en boîte noire comment est calculé le vecteur de caractéristiques.

La quête d'un émulateur s'est avérée longue et douloureuse. Je n'ai pas réussi à percer le mystère de `qemu-ia64`. Mythe ou réalité ? Tout ce que j'ai trouvé, ce sont des projets Github qui disent avoir rajouté le support d'IA64. Mais qui en fait ont juste créé un fichier vide et rajouté un TODO :) Je me suis donc plutôt concentré sur l'émulateur `ski`, développé par HP. Cet émulateur semblait être la référence lors des débuts d'Itanium. Mais depuis 2008 il est au point mort. J'ai tenté de le compiler plusieurs fois en vain.

11. *« The Itanium approach...was supposed to be so terrific—until it turned out that the wished-for compilers were basically impossible to write »*

Finalement, alors que j'étais à deux doigts de passer sur une autre épreuve, j'ai trouvé un vieux `.deb` pour Ubuntu Karmic Koala 9.10 [12], qui s'est installé sans aucun problème sur ma Debian Wheezy. L'émulateur `ski` a aussi besoin d'une libc compilée pour IA-64. Et comme je n'ai pas très envie de sortir le cross-compilateur, j'ai trouvé mon bonheur dans un recoin d'internet [13]. Ça y est ! Je peux émuler de l'IA-64. J'ai même un débuggeur (sans documentation et assez folklorique à prendre en main... mais malgré tout extrêmement utile).

J'ai donc créé avec Gimp quelques vignettes, et j'ai observé les 4 flottants que j'obtenais en sortie de fonctions. Puis j'ai fait varier légèrement mes images pour tenter de comprendre d'une part sur quoi sont basées les caractéristiques de l'OCR, et d'autre part pour obtenir une vignette dont les 4 flottants seront inférieurs à 0.15. Bilan, je n'ai pas compris grand-chose, mais j'ai généré une vignette qui a validé le premier tour de tests !

Oui... mais encore ? C'est de l'art abstrait ? Qu'est-ce que je suis censé dessiner ? J'ai commencé à faire le même travail pour tenter de créer à la main une vignette qui passera le 2ème tour de tests mais je me suis vite arrêté en réalisant que je n'allais pas faire ça 32 fois.

Il faut donc poser le problème différemment. Que se passe-t-il si on génère un P2 160x20 qui validera tous les tests ? Où est la clé ? Le programme n'affichera rien, à part des points et une string de 4 octets. La clé est donc forcément dans l'image qu'on doit fournir. Il est donc hautement probable que cette image soit composée de 32 vignettes qui représentent chacune un caractère hexa. Savoir à quel caractère hexa correspondait le faux-positif ci-dessus est laissé en exercice au lecteur.

Cette fois, je sors donc les grands moyens, et j'écris un script PIL qui crée des vignettes de caractères hexa valides, pour chaque police Truetype installée sur mon OS. Je bruteforce. Et je trouve assez rapidement une clé qui passe tous les tests. Le programme m'affiche alors le message *pass*. Je retourne donc dans le RPG et je donne ma clé au gardien, qui m'envoie gentiment balader.

Dans ma clé, je constate qu'il n'y a presque aucune lettre. Il y a par exemple un 'D' majuscule, qu'un OCR pourrait très bien confondre avec un 0. Je fais donc la supposition que la clé ne comporte que des chiffres. Je réécris mon script PIL en conséquence. Je bruteforce. Je trouve une clé. Je retourne voir le gardien du RPG, qui m'envoie à nouveau balader.

12. https://launchpad.net/ubuntu/karmic/amd64/ski/1.3.2-4
13. ftp://rpmfind.net/linux/Mandriva/devel/cooker/ia64/media/main/glibc-2.3.3-12.7.100mdk.ia64.rpm

Voici mon script (un peu cracra) qui génère des vignettes de caractères valides à partir de polices Truetype

```
import os, sys
import ImageFont, ImageDraw, Image, ImageChops

def save_to_p2(img, path):
    w, h = img.size
    p2 = "P2\n#PIL Powered\n%i %i\n255\n" % (w, h)
    for y in range(h):
        for x in range(w):
            p2 += "%i\n" % img.getpixel((x,y))
    with open(path, "wb") as f:
        f.write(p2)

def build_valid_image(c, font_path):
    """ Get the best font size to fit the size required by
    Strange level (20x20 pixels, first line empty,
    columns strictly centered) """
    for size in range(5, 50):
        font = ImageFont.truetype(font_path, size)
        img = Image.new("1", (100,100), "white")
        draw = ImageDraw.Draw(img)
        draw.text((0, 0), c, font=font, fill="black")
        img = remove_white(img)
        if not img:
            continue
        w, h = img.size
        if h == 19 and w <= 20:
            break
    else:
        return None
    valid_img = Image.new("1", (20,20), "white")
    valid_img.paste(img, ((20-w)/2,1))
    if w % 2 != 0:
        # Need to fix width
        w = (20-w)/2 + w - 1
        w = min(w, 19)
        for h in range(20):
            pix = valid_img.getpixel((w, h))
            if pix == 0:
                valid_img.putpixel((w+1, h), 0)
                break
    return valid_img

def remove_white(img):
    bg = Image.new(img.mode, img.size, 255)
    diff = ImageChops.difference(img, bg)
    bbox = diff.getbbox()
    if bbox:
        img = img.crop(bbox)
        return img

if __name__ == '__main__':
    if not os.path.exists("p2"):
        os.mkdir("p2")
    for path, dirs, files in os.walk("/usr/share/fonts/truetype/"):
        for f in files:
            font_path = os.path.join(path, f)
            dirname = os.path.join("p2", f[:-4])
            for c in "0123456789":
                img = build_valid_image(c, font_path)
                if not img:
                    continue
                if not os.path.exists(dirname):
                    os.mkdir(dirname)
                save_to_p2(img, os.path.join(dirname, c) + ".p2")
```

Et voici mon script qui assemble les vignettes en une image P2 et qui utilise `ski` pour bruteforcer la clé. Le bruteforce est d'ailleurs facilité (et encouragé ?) par les concepteurs, puisque plus le programme va loin dans son exécution, plus son code de retour est incrémenté. La vérification du code de retour est donc suffisante pour savoir si une vignette a été reconnue comme valide par l'OCR.

```python
import os
from subprocess import Popen, PIPE

def make_p2(files):
    p2 = "P2\n#Where is the lobster ?\n640 20\n255\n"
    pix = ["255"]*12800
    for idx, p2f in enumerate(files):
        if os.path.exists(p2f):
            with open(p2f) as f:
                pixels = f.read().strip().split("\n")[4:]
            for y in range(20):
                for x in range(20):
                    pix[x + y*640 + idx*20] = pixels[x + y*20]
    for p in pix:
        p2 += p + "\n"
    with open("/tmp/w00t.p2", "wb") as w00t:
        w00t.write(p2)

def bruteforce(key, retcode):
    for path, dirs, files in os.walk("p2"):
        for f in files:
            f = os.path.join(path, f)
            make_p2(key + [f])
            p = Popen(["bskinc", "a.out", "/tmp/w00t.p2"], stdout=PIPE)
            out, err = p.communicate()
            if p.returncode >= retcode + 2 or "fail" not in out:
                key.append(f)
                return p.returncode

if __name__ == '__main__':
    key = []
    retcode = 14
    while len(key) < 32:
        retcode = bruteforce(key, retcode)
        print retcode, key
    print "Win !?"
```

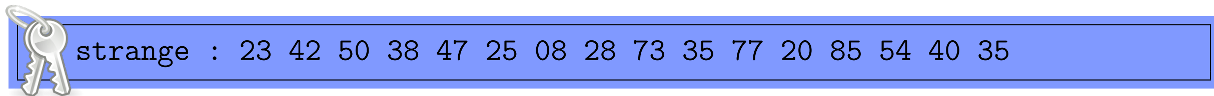
```
$ python strange.py
16 ['p2/TlwgTypewriter-BoldOblique/2.p2']
19 ['p2/TlwgTypewriter-BoldOblique/2.p2', 'p2/Kedage-n/3.p2']
22 ['p2/TlwgTypewriter-BoldOblique/2.p2', 'p2/Kedage-n/3.p2', 'p2/DejaVuSansMono/4.p2']
25 ['p2/TlwgTypewriter-BoldOblique/2.p2', 'p2/Kedage-n/3.p2', 'p2/DejaVuSansMono/4.p2', 'p2/Kedage-n/2.p2']
[...]
Win !?
```

23425038972508287335772085140035

Cette clé contient en réalité 3 erreurs. Mais les trouver n'est pas si facile. À ce moment, j'hésite entre deux possibilités. Patcher le binaire pour baisser la marge d'erreur (actuellement de 0.15) et rendre l'OCR plus strict, en espérant diminuer le nombre de faux positifs. Ou alors, plutôt que de m'arrêter dès que je trouve une vignette valide,

confirmer ou infirmer ce caractère en le testant sur un grand échantillon de polices. La loi des grands nombres voudrait ainsi que j'élimine mes faux positifs.

Je suis parti sur la deuxième piste. Et j'ai fini par trouver la bonne clé. Il s'avère qu'elle ne contient aucun 1, 6 et 9. (Est-ce pour ça que le fichier de données s'appelle 196 ?). Sinon, aux vues des résultats obtenus par mon bruteforce, j'en déduis que l'image de la clé utilisée pour concevoir le challenge est en écriture manuscrite. Ou en tout cas, elle mélange plusieurs polices.



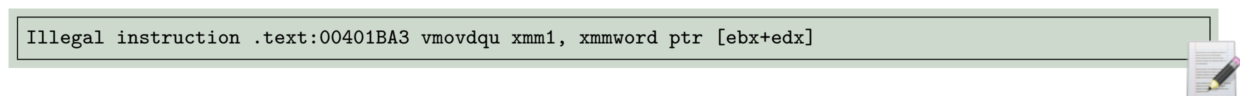
```
strange : 23 42 50 38 47 25 08 28 73 35 77 20 85 54 40 35
```

5.2 video

Attention, âmes sensibles s'abstenir, ma solution pour cette épreuve est élégamment très crade. L'épreuve commence avec une proposition de stage donnée par un PNJ de la société Airlhes : suite à un incident de sécurité, il faut prouver qu'un écran de veille suspicieux n'a pas exfiltré la clé privée maîtresse d'un serveur.

Avec ce document, est fourni l'installeur de l'écran de veille et une vidéo. Elle montre un ordinateur, filmé de dos, dans un bureau plongé dans le noir. On peut observer les couleurs de l'écran de veille qui éclairent la pièce et se reflètent contre le mur blanc. On distingue clairement deux phases dans cette vidéo. D'abord, les couleurs évoluent lentement. Ensuite, il y a une série de flash très rapides, qui doit correspondre à l'exfiltration de données.

L'écran de veille s'avère être un binaire PE tout à fait classique. Après une première analyse en surface, où l'on rencontre (pour changer) pas mal d'opérations sur des nombres flottants, je ressens vite le besoin de pouvoir lancer l'écran de veille pour voir ce qu'il affiche. Seulement, il ne semble pas fonctionner.



```
Illegal instruction .text:00401BA3 vmovdqu xmm1, xmmword ptr [ebx+edx]
```

L'exécutable utilise en effet par endroits le jeu d'instruction AVX, disponible uniquement sur les processeurs très récents. Mon processeur ne les supporte pas, et ma machine virtuelle non plus. Pour pouvoir lancer malgré tout l'écran de veille, j'ai utilisé Intel SDE (Software Development Emulator), afin d'émuler les instructions AVX. Par contre, je n'ai pas pu utiliser de débuggeur.

L'écran de veille est plutôt joli. Trois cercles de couleur se déplacent pseudo-aléatoirement sur l'écran, et éclairent une image de fond sur laquelle on peut lire « *Centre AIRLHES de Cryptographie Appliquée* » (C.A.C.A. pour les intimes). On distingue également une chaîne en hexa, mais qui n'est que partiellement lisible. Cependant, contrairement à la vidéo d'exfiltration, il n'y a aucun flash de lumière.

Afin de récupérer la chaîne hexa, j'ai pu retrouver la trace de l'image de fond utilisée par l'écran de veille. Elle est stockée dans les données du PE, sous la forme d'un buffer compressé avec l'algorithme `lznt1`. Une fois ce buffer décompressé, et ouvert dans Gimp sous forme de tableau de pixels bruts, on obtient cette image un peu déformée :

Cela permet de récupérer la clé 21 35 37 33 67 30 2d 66 52 33 33 2d 50 4b 49 21, ce qui correspond en ASCII à `!573g0-fR33-PKI!`. Par acquis de conscience, j'ai essayé de la donner au gardien du niveau 3. Mais il n'en veut pas. Pour avancer, il va donc falloir explorer d'autres pistes et se poser les bonnes questions.

Pourquoi mon écran de veille ne fait pas de flash ? Il se trouve que le programme fait très fréquemment appel à une fonction qui récupère l'heure de la machine et renvoie 0 ou 1 après y avoir appliqué quelques calculs savants. Dans un cas, l'écran de veille fait des flashs de couleur, dans l'autre, il affiche les cercles normaux. Je n'ai pas pris le temps de comprendre quels sont les opérations faites sur l'heure. À la place, j'ai déterminé de manière empirique, en changeant l'heure de mon poste, que l'écran de veille passe en mode exfiltration entre minuit et 1h du matin. Ce qui est relativement logique, puisque le scénario d'attaque se passe la nuit, lorsque les locaux de l'entreprise sont vides.

D'où proviennent les données qui sont exfiltrées ? Du seul point d'entrée du programme : la clé de registre *Software/Airlhes/Screensaver/config/trajectories*. Quand on installe l'écran de veille, cette clé de registre a une valeur par défaut. Si on la change en mode de fonctionnement normal, on ne perçoit aucune différence. Le fait qu'il s'agisse de trajectoires est donc probablement du bluff. Par contre, après minuit, les flashs de couleur sont modifiés. C'est d'ailleurs l'occasion de se pencher un peu plus sur la valeur par défaut de la clé de registre. Il s'avère qu'il s'agit d'une chaîne de caractères compressée dans un stream zlib.

```
>>> k = '78DAF348CDC9C957482BCA4CCD4B51B0D204002B740510'
>>> k.decode("hex").decode("zlib")
'Hello friend :)'
```

Comment fonctionne l'algorithme qui convertit les octets en couleurs ? L'algorithme n'est pas trivial à reverser. Cependant on trouve assez facilement la palette des 8 couleurs qui est utilisée. On remarque également qu'un tableau d'octets est utilisé par cet algorithme, et que l'offset de départ n'est pas le même si l'écran de veille a été appelé avec ou sans argument (sachant que Windows appelle par défaut ces binaires avec l'option `/s`). Pour le reste, le moins fatiguant est de commencer par une analyse en boîte noire. On modifie la valeur de la clé de registre, et on compare le résultat obtenu. Les flashs qui exfiltrent la valeur de registre "00" sont très différents de ceux pour "0000". Il y a plus de 15 flashs pour juste un octet, ce qui semble beaucoup. Par contre, l'exfiltration entre "00" et "01" est très proche : seul les trois derniers flashs changent. La première conclusion est que la longueur de la donnée à exfiltrer influe beaucoup sur les flashs de couleurs. En effet, cette longueur est stockée sur un entier de 4 octets puis envoyée de la même manière que le reste. La seconde conclusion, c'est qu'il n'est pas nécessaire de comprendre davantage l'algorithme, car il est bruteforçable !

Comment bruteforcer ? Le plus propre est certainement de le retranscrire dans un autre langage. Il est également possible d'extraire les routines assembleur qui nous intéressent, et des les réutiliser dans notre programme. Mais ça reste relativement compliqué, car le flot d'exécution n'est pas complètement trivial, il y a le risque de se tromper lors de la retranscription, ou d'oublier des morceaux. Il n'y a pas de contrôle d'intégrité sur ce programme. J'ai donc fait le choix de patcher directement le PE. Même pas besoin d'un assembleur, car toutes les instructions dont j'avais besoin étaient présentes à d'autres endroits dans le binaire.

Tout d'abord, j'ai remplacé toutes les fonctions utilisant des instructions AVX par un `ret`, après avoir constaté qu'elles étaient uniquement utilisées dans le cas de l'écran de veille légitime (pour faire de la trigonométrie par exemple). Ainsi, on peut lancer l'écran de veille sans SDE, et on peut le débugger. Ensuite, j'ai patché la fonction qui vérifie l'heure de la machine, afin qu'elle renvoie toujours 0. Pour gagner du temps, j'ai patché les boucles pour construire une image d'un seul pixel, au lieu des 1280x720 initiaux. J'ai modifié les imports pour pouvoir appeler directement `printf` (j'ai simplement écrasé le vfprintf qui n'était pas utilisé). Puis enfin, j'ai patché la fonction qui est censée rafraîchir la fenêtre d'affichage, et je l'ai remplacée par un appel à `printf` avec la valeur du pixel courant exfiltré.

C'est dans la poche. Ou presque, car j'ai découvert que comme ce programme est une application graphique (GUI_APP), on ne peut pas écrire (facilement) sur sa sortie standard (puisqu'il n'en a pas !?). Il faut donc d'abord le transformer en application console (CONSOLE_APP). Pour ce faire, il y a deux endroits à patcher : le flag passé à la fonction ___set_app_type, juste après le point d'entrée du programme, ainsi qu'un flag du header PE (OPTIONAL_HEADER.Subsystem).

Ça y est, en modifiant moins de 50 octets, Airlhes.scr a été transformé en une application console qui va lire la clé de registre *trajectories* et réaliser l'exfiltration en affichant les couleurs obtenues sur sa sortie standard. Voici le programme Python qui réalise les patchs en utilisant `pefile`.

```python
import pefile

pe = pefile.PE("Airlhes.scr")

# Remove all AVX-related functions
pe.set_bytes_at_rva(0x1b40, "\xc3") # ret
pe.set_bytes_at_rva(0x16c0, "\xc3")
pe.set_bytes_at_rva(0x2670, "\xc3")
pe.set_bytes_at_rva(0x1fc0, "\xc3")
pe.set_bytes_at_rva(0x20c0, "\xc3")
pe.set_bytes_at_rva(0x21c0, "\xc3")

# Time stuff always returns 0
pe.set_bytes_at_rva(0x3750, "\x31\xC0\xC3") # xor eax, eax ; ret

# Resolution of image is now 1x1 instead of 1280x720
pe.set_bytes_at_rva(0x3d80, "\x01\x00")
pe.set_bytes_at_rva(0x3da0, "\x01\x00")

# Do 1 loop round by color instead of 8
pe.set_bytes_at_rva(0x3dba, "\x08") # add edi, 8

# Change type GUI_APP to CONSOLE_APP
pe.set_bytes_at_rva(0x12a6, "\x01")


# Patch of GUI message stuff, to simply printf the current color instead.
# We keep the prologue.
# At 0x402A2F, esi is a pointer to the pixel map
# Then, we jump to the epilogue at 0x402B85
# For the printf format, we use the substring at 0x40D35D, which is "%d.\n\x00"

# Push printf arguments
# mov dword ptr [esp], offset "%d"  ##  C7 04 24 5D D3 40 00
# mov ebx, [esi]  ## 8B 1E
# mov [esp+4], ebx  ##  89 5C 24 04

# Call printf
# We are at 0x402A2F + 7 (mov format) + 6 (mov esi) + 5 (current opcode) == 0x402A41
# Printf is at 0x404D90
# So relative value for call is 0x404D90 - 0x402A41 = 0x234F
# call printf  ##  E8 4F 23 00 00

# Prepare jump to epilogue
# mov eax, 1  ##  B8 01 00 00 00
# We are at 0x402A41 + 5 + 5 == 0x402A4B
# Relative offset is 0x402B8D - 0x402A4B = 0x142
# jmp epilogue  ##  E9 42 01 00 00

printf_shellcode = "C704245DD340008B1E895C2404E84F230000B801000000E942010000".decode("hex")
pe.set_bytes_at_rva(0x2a2f, printf_shellcode)

# Set WINDOWS CONSOLE type
pe.OPTIONAL_HEADER.Subsystem = 3

# Rebuild PE
pe.write("Airlhes_patched.scr")

# Replace vfprintf by printf
with open("Airlhes_patched.scr", "r+b") as f:
    data = f.read()
    data = data.replace("vfprintf\x00", "printf\x00\x00\x00")
    f.seek(0)
    f.write(data)
```

Ensuite, pour réaliser le bruteforce, un deuxième script Python va instrumenter le programme patché.

```python
from subprocess import *
import _winreg as reg

colors = {
    0xFF0000: "Rouge",
    0x00FF00: "Vert",
    0xFFFF00: "Jaune",
    0x0000FF: "Bleu",
    0xFF00FF: "Violet",
    0x00FFFF: "Cyan",
    0xFFFFFF: "Blanc",
    0x000000: "Noir",
}

magic = ["Blanc"]*8 + ["Noir"]*8 + ["Blanc"]*8

secret = [
    "Cyan", "Violet", "Cyan",
    "Violet", "Bleu", "Violet",
    "Rouge", "Bleu", "Noir",
    "Rouge", "Bleu", "Cyan",
    "Blanc", "Jaune", "Bleu",
    "Rouge", "Noir", "Violet",
    "Blanc", "Vert", "Jaune",
    "Noir", "Rouge", "Jaune",
    "Cyan", "Blanc", "Noir",
    "Cyan", "Noir", "Vert",
    "Blanc", "Jaune", "Bleu",
    "Jaune", "Bleu", "Violet",
    "Bleu", "Violet", "Noir",
    "Blanc", "Bleu", "Cyan",
    "Bleu", "Vert", "Bleu",
    "Blanc", "Bleu", "Blanc",
    "Rouge", "Blanc", "Vert",
    "Cyan", "Rouge", "Jaune",
    "Cyan", "Vert", "Bleu",
    "Rouge", "Vert", "Cyan",
    "Rouge", "Jaune", "Violet",
    "Blanc", "Vert", "Violet",
    "Cyan", "Jaune", "Blanc",
    "Noir", "Rouge", "Cyan",
    "Blanc", "Noir", "Cyan",
    "Noir", "Bleu", "Rouge",
    "Blanc", "Violet", "Noir",
    "Vert", "Rouge", "Violet",
]

def read_color(proc):
    line = proc.stdout.readline().strip("\r\n.")
    c = colors[int(line)]
    return c

def read_until_magic(proc):
    data = []
    while data[-24:] != magic:
        data.append(read_color(proc))
    return data[:-24]

def get_score(data, secret):
    length = min(len(data), len(secret))
    score = 0
    while score < length and data[score] == secret[score]:
        score += 1
    return score
```

```
65
66  def set_reg(value_name, value):
67      hkey = reg.OpenKey(reg.HKEY_CURRENT_USER, "Software\\Airlhes\\Screensaver\\config\\trajectories", 0, reg.KEY_ALL_ACCESS)
68      reg.SetValueEx(hkey, value_name, 0, reg.REG_BINARY, value)
69      hkey.Close()
70
71  def run_scr():
72      p = Popen(["Airlhes_patched.scr", "/lobster"], stdout=PIPE)
73      read_until_magic(p)
74      result = read_until_magic(p)
75      p.terminate()
76      s = get_score(result, secret)
77      return s
78
79  def bruteforce_key_length():
80      length = 1
81      while True:
82          set_reg("spot0", "\x00"*length)
83          score = run_scr()
84          if score >= 12:
85              print "Key length found :", length
86              return length
87          length += 1
88
89  def bruteforce_key(key_length):
90      key = []
91      score = 12 + 3*len(key)
92      c = len(key)
93      while c < key_length:
94          for i in range(0x100):
95              set_reg("spot0", "".join(key) + chr(i) + "\x00"*(key_length-c-1))
96              s = run_scr()
97              if s >= score + 3:
98                  score += 3
99                  key.append(chr(i))
100                 print "New key byte found : 0x%02X" % i
101                 break
102         c += 1
103     return "".join(key)
104
105 if __name__ == '__main__':
106     kl = bruteforce_key_length()
107     k = bruteforce_key(kl)
108     print "Key is", k.encode("hex").upper()
109     print "Flag is", k.decode("zlib").encode("hex").upper()
```

```
C:\sstic\stage3\video>python scr.py
Key length found : 24
New key byte found : 0x78
New key byte found : 0xDA
New key byte found : 0x0B
New key byte found : 0x16
New key byte found : 0x34
New key byte found : 0x17
New key byte found : 0x2B
New key byte found : 0xDA
New key byte found : 0xC5
New key byte found : 0x58
New key byte found : 0xF9
New key byte found : 0xCB
New key byte found : 0x6E
New key byte found : 0x62
New key byte found : 0x57
New key byte found : 0xF3
New key byte found : 0xBE
New key byte found : 0x7B
New key byte found : 0x5B
New key byte found : 0x00
New key byte found : 0x32
New key byte found : 0x50
New key byte found : 0x07
New key byte found : 0x7E
Key is 78DA0B1634172BDAC558F9CB6E6257F3BE7B5B003250077E
Flag is 5311371672BA0179FA3E918A83BEDEB4
```

```
video : 53 11 37 16 72 BA 01 79 FA 3E 91 8A 83 BE DE B4
```

6 Victoire

Une fois suffisamment de clés récoltées, le garde du niveau 3 nous laisse pénétrer dans la salle du trône, où nous sommes accueillis par une musique absolument formidable :).

Le roi nous donne une ultime énigme.

```
Coucou !

Tu as presque reussi le challenge !

I01p1 y'4qe3553 z41y : 8Y6d5j9Vy88HUGHfGSKsJvqA@ffgvp.bet
```

Il s'agit d'un simple rot13. L'adresse email de validation du challenge SSTIC 2016 est donc 8L6q5w9Il88UHTUsTFXfWidN@sstic.org

7 Conclusion

Cette édition 2016 du challenge SSTIC était excellente. J'ai été bluffé par les efforts des concepteurs pour intégrer leurs épreuves dans un RPG. J'ai pu découvrir l'architecture IA-64, le byte-code EFI, ou encore ce qu'est un fichier *sparse*. Et j'ai un peu moins peur des flottants.

Je reste cependant assez mitigé sur la nouveauté de cette année qui consiste à avoir le choix dans les épreuves. Certes, j'y vois certains avantages, car cela peut permettre de ne pas décourager trop vite les participants qui seraient réfractaires à certaines épreuves. C'est un bon appât. Mais je trouve ça assez étrange que deux personnes puissent finir le challenge en ayant au final résolu une seule épreuve en commun. Peut-être qu'une meilleure formule serait de commencer par du choix multiple, mais de réduire le choix au fur et à mesure, et de terminer par un boss de fin qui soit commun à tout le monde (ring ? ... ou pas).

J'ai pris du plaisir avec ce challenge. Mais cette nouvelle formule m'a amené à réfléchir à ce qui me plaisait tant dans les challenges de sécurité (et dans les challenges SSTIC en particulier). En fait, en matière de challenge, je n'aime pas avoir le choix. J'aime être obligé de devoir m'attaquer à une technologie que je ne connais pas ou qui me révulse. D'avoir l'impression que c'est trop dur pour moi et que je n'y arriverai jamais. Je vomis mes tripes pendant plusieurs jours. Mais je persévère car je veux terminer le challenge et que je n'ai pas le choix. Quand finalement j'arrive au bout de l'épreuve, le sentiment d'avoir réussi à faire des choses dont je me sentais incapable quelques jours plus tôt est grisant.

Quand on a le choix, je trouve qu'il est plus compliqué de garder sa motivation face à la difficulté. Il est trop tentant de laisser de côté ce qui nous semble trop dur, pour se diriger vers les épreuves plus familières, qui au final nous apprendront moins de chose et nous apporteront moins de plaisir.

Chers lecteurs, pour toute remarque, commentaire ou insulte, n'hésitez surtout pas à me contacter à l'adresse email encodée en dernière page.

Merci pour tout et à l'année prochaine !