Challenge SSTIC

UNE SOLUTION PERILLEUSE

Eric Dangereux | Conception de Base | 13 avril 2016

Introduction

Ce document présente une brève solution au challenge SSTIC 2016, organisé par l'équipe Zèbre.

Le challenge est composé de multiples épreuves. Seule une partie d'entre elles nécessite d'être résolue pour arriver au bout. Ayant eu malheureusement peu de temps à y consacrer, je ne détaillerai ici que les épreuves sur lesquelles j'ai pu travailler.

Les solutions sont brèves : plutôt que d'expliquer le fonctionnement de chaque niveau, j'ai préféré présenter la méthode et les intuitions qui m'ont permis de résoudre le challenge finalement assez rapidement.

Le challenge est un fichier au format pcap. On observe dans ce fichier une requête GET vers http://static.sstic.org/challenge2016/challenge.zip et la réponse du serveur. Ce fichier n'est plus présent sur le serveur, mais il peut être extrait de la capture avec Wireshark, par exemple.

Le Zip contient un jeu de rôle développé avec RPG JS. On incarne un petit personnage pixellisé. Un manuel indispensable explique au départ comment mouvoir le personnage avec les flèches directionnelles, utile aux personnes ne maniant pas au quotidien les interfaces graphiques.

En sortant de la maison, on rencontre plusieurs individus qui proposent des défis. C'est parti !

Niveau 1

Les discussions sont plus ou moins fructueuses. Je suis personnellement peu enclin à converser avec des gens demandant si je veux sentir leur grosse antenne tout en précisant qu'ils sont un bon coup. J'ai préféré m'éloigner et aller discuter avec d'autres villageois.

Un type avec un chapeau d'aventurier (j'aime bien les chapeaux d'aventuriers) nous donne une capture réseau. Les aventuriers ont bien changé, on dirait, mais l'épreuve a l'air intéressante.

SOS-FANToME

Le PCAP de cette épreuve est une capture réseau entre deux machines ayant des adresses MAC issues d'un dessin animé, d'une chanson de Boney M ou d'une expression swahilie. Cette expression, « Hakuna matatizo », signifie qu'il n'y a pas de problème, probablement pour tromper les analystes.

Ces deux machines communiquent en TCP. Les données échangées commencent toutes par « Ghost ». Une recherche Internet montre qu'il s'agit d'un outil d'administration à distance (OAD). Les sources de l'OAD sont disponibles : https://github.com/sincoder/ghost. Il n'y a pas vraiment besoin des sources pour comprendre le format des paquets : les octets 14 et 15 des paquets començant par « Ghost » valent tous 78 9c, soit l'en-tête de données compressés avec Zlib avec un niveau de compression standard. On déduit le reste du format facilement :

- Un en-tête fixe, « Ghost » ;
- 4 octets spécifiant la taille des données compressées ;
- 4 octets spécifiant la taille des données décompressées ;
- Des données compressées.

Les données peuvent être réassemblées avec Scapy et décompressées. Étant assez fainéant, j'ai choisi de mettre à la suite toutes les données de tous les paquets, les paquets Ghost étant bien délimités, et de les décompresser dans une seconde phase. Ce qui donne un script d'une grande qualité exploitant au maximum les possibilités de Scapy :

```
import struct
import zlib
from scapy.all import *


def main():
    sess = rdpcap("SOS-Fant0me.pcap")

    data = ""
    for pkt in sess:
        if Raw in pkt:
            data += pkt.getlayer(Raw).load
    print(len(data))

    while len(data) > 0:
        packed_len, unpacked_len = struct.unpack('<LL', data[5:13])
        print(zlib.decompress(data[13:13 + packed_len]).encode("hex"))

        data = data[packed_len:]


if __name__ == '__main__':
    main()
```

Toutes ces données ont ensuite été analysées avec un éditeur hexadécimal.

On repère aisément un Zip à la fin des données extraites, ainsi que quelques lignes correspondant probablement à des frappes clavier :

```
7c43
7c79
7c62
7c33
7c72
7c53
7c53
7c54
7c49
7c43
7c5f
7c32
7c30
7c31
7c36
```

Le Zip est protégé par un mot de passe, et les frappes clavier correspondent à ce mot de passe :

```
00000650  35 5D 20 73 73 74 69 63 32 30 31 36 2D 73 74 61  5] sstic2016-sta
00000660  67 65 31 2D 73 6F 6C 75 74 69 6F 6E 2E 7A 69 70  ge1-solution.zip
00000670  20 2D 20 53 61 69 73 69 72 20 6D 6F 74 20 64 65   - Saisir mot de
00000680  20 70 61 73 73 65 7C 43 7C 79 7C 62 7C 33 7C 72   passe|C|y|b|3|r
00000690  7C 53 7C 53 7C 54 7C 49 7C 43 7C 5F 7C 32 7C 30  |S|S|T|I|C|_|2|0
```

```
000006A0  7C 31 7C 36 01 67 43 03 F4 27 00 00 EB 1F 00 00  |1|6.gC.ô'..ë...
```

Le Zip contient un premier drapeau : 368BE8C1CC7CC70C2245030934301C15.

La clé a été donnée à un soldat lourdement armé, qui demande d'autres clés. Il faut donc aller voir l'autre villageois (le Texan sans chapeau).

CALC

Ce Texan nous propose un nouveau défi : un fichier nommé SSTIC16.8xp. Il s'agit d'un programme TI-BASIC pour TI-38 Plus. Le site SourceCoder 3 (https://www.cemetech.net/sc/) permet de décompiler de tels fichiers.

On remarque trois choses :

- Le programme contient plein de directives Goto et un ordonnanceur en tout début de programme ;
- Une table CRC32 est présente ;
- Une comparaison est effectuée ligne 174 entre une variable A et la valeur 3298472535.

Le flot d'exécution a l'air pénible à analyser. Or, il s'agit du premier niveau du challenge, donc l'épreuve doit être rapide à résoudre.

Il faut entrer un code, stocké dans la variable Z. La variable C est ensuite initialisée à 4294967295, soit la valeur de départ du CRC32. La première intuition est d'entrer un code dont le CRC32 vaut 3298472535.

L'algorithme étant inversible, le calcul est immédiat. N'ayant pas de script sous la main, j'ai préféré faire une recherche exhaustive et aller prendre un café. La solution a été trouvée avant que j'aie pu remplir le percolateur :

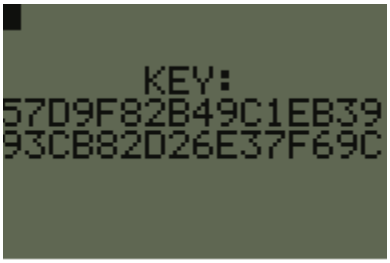
```
import zlib
import time

n = 0
start_time = time.time()
while 1:
    crc = zlib.crc32(struct.pack('<L', n))
    if crc < 0:
        crc += 0x100000000
    if crc == 3298472535:
        print(hex(n))
        break
    n += 1

print("elapsed time: {0}".format(time.time() - start_time))
```

```
0x5571c16
elapsed time: 65.2030000687
```

La valeur obtenue montre que notre intuition était bonne. Reste à récupérer le drapeau sans, si possible, devoir étudier comment il est généré à partir du code.

On peut pour cela utiliser un émulateur. J'ai cherché un émulateur récent, car de mémoire ceux du XXe siècle fonctionnaient assez mal. Wabbitemu (https://wabbit.codeplex.com/) semble être un outil de choix, et il permet d'installer des programmes TI-BASIC.

La solution est affichée au bout de quelques longues secondes :



La recopie dans le jeu est assez pénible, mais ça vaut le coup : le soldat est maintenant content et nous laisse passer.

Niveau 2

De l'autre côté du pont, le climat a complètement changé. Le sol est enneigé, ce qui facilite l'aventure : des traces de pas dans la neige nous indiquent la direction : la bibliothèque. Curieusement, notre héros ne laisse pas de traces.

Une personne au fond mentionne la police[1]. J'ai préféré m'enfuir et regarder les deux autres épreuves, EFI et Huge, épreuves que j'ai trouvé particulièrement intéressantes.

EFI

Il s'agit d'analyser un exécutable PE contenant du bytecode EFI. Il est composé de 5 fonctions : une analyse statique devrait suffire. Un code doit être entré, lequel est traité par la fonction sub_10000530 qui doit retourner une valeur non nulle pour valider le défi.

Dans cette fonction, une table à l'adresse 10001698 est référencée. Il s'agit du GUID EFI_AUTHENTICATION_INFO_PROTOCOL_GUID. Or, ce GUID n'est pas utilisé tel quel : un masque disjonctif avec des données d'une table fixe lui est appliqué :

```
>>> tbl1 = [0x2E, 0xA5, 0x60, 0xAE, 0x7D, 0xC7, 0xA7, 0x50, 0x30, 0x53,
0x23, 0xB7, 0xD5, 0x20, 0xCA, 0x8A]
>>> auth_guid = [0xD0, 0xD9, 0x71, 0x76, 0xDB, 0x53, 0x73, 0x41, 0xAA,
0x69, 0x23, 0x27, 0xF2, 0x1F, 0xB, 0xC7]
>>> [hex(a ^ b) for a, b in zip(tbl1, auth_guid)]
['0xfe', '0x7c', '0x11', '0xd8', '0xa6', '0x94', '0xd4', '0x11',
'0x9a', '0x3a', '0x0', '0x90', '0x27', '0x3f', '0xc1', '0x4d']
```

On obtient alors le GUID EFI_DECOMPRESS_PROTOCOL_GUID. Il s'agit de la première sournoiserie de l'auteur. Une instance du protocole EFI_Decompres est ensuite récupérée :

[1] Je n'ai jamais regardé les polices Windows, et je ne savais pas s'il existait des outils permettant de les analyser rapidement. Après le challenge, j'ai jeté rapidement un coup d'œil et il s'est avéré que ttfdump semblait très adapté, mais passons.

```
.text:10000620  MOVnd       [SP], [SP+0x85E0+var_90] ; Protocol
.text:10000626  MOVnw       [SP+4], R6  ; Registration
.text:1000062A  MOVnd       [SP+8], SP+0x85E0+var_Interface
.text:10000634  CALL32EXa   [R7+EFI_BOOT_SERVICES.LocateProtocol]
```

Ensuite, deux méthodes exposées par ce protocole sont appelées :

```
.text:1000063A  MOVqd       [SP+0x85E0+var_pEfiDecompress], R7
.text:10000676  MOVnd       [SP], [SP+0x85E0+var_Interface] ; This
.text:1000067C  MOVnd       [SP+4], [SP+0x85E0+var_98] ; Source
.text:10000686  MOVdd       [SP+8], [SP+0x85E0+var_60] ; SourceSize
.text:10000690  MOVnd       [SP+0xC], SP+0x85E0+var_54 ;
DestinationSize
.text:1000069A  MOVnd       [SP+0x10], SP+0x85E0+var_58 ; ScratchSize
.text:100006A4  CALL32EXa   [R7]        ; GetInfo
.text:100006A6  MOVqd       [SP+0x85E0+var_pEfiDecompress], R7
.text:100006AC  MOVqd       R7, [SP+0x85E0+var_pEfiDecompress]
.text:100006B2  CMP64eq     R7, R6
.text:100006B4  JMP8cc      loc_10000710
.text:100006B6  MOVnd       R7, [SP+0x85E0+var_Interface]
.text:100006BC  MOVnd       [SP], [SP+0x85E0+var_Interface] ; This
.text:100006C2  MOVnd       [SP+4], [SP+0x85E0+var_98] ; Source
.text:100006CC  MOVdd       [SP+8], [SP+0x85E0+var_60] ; SourceSize
.text:100006D6  MOVnd       [SP+0xC], SP+0x85E0+decompressed_data ;
Destination
.text:100006E0  MOVdd       [SP+0x10], [SP+0x85E0+var_54] ;
DestinationSize
.text:100006EA  MOVnw       [SP+0x14], SP+0x85E0+var_85A8 ; Scratch
.text:100006F0  MOVdd       [SP+0x18], [SP+0x85E0+var_58] ; ScratchSize
.text:100006FA  CALL32EXa   [R7+4]      ; Decompress
```

Les données à l'adresse var_98, précédemment initialisé à 10001200 + 0x270, sont donc a priori décompressées. Quel est l'algorithme de compression utilisé ? Il semble que trois algorithmes sont couramment mis en œuvre : Efi, Tiano et LZMA. Un module Python, uefi_firmware, les gère tous les trois. C'est Efi qui est utilisé :

```
from uefi_firmware import efi_compressor
s =
"3F0000005C000000003C4C8D823302ED6400B717307DA812AF1B021DFAB86124FE3B24
F51FCCCE8BA7738572726A518F09C1D4B10C7BA3A1B3CF078FCD9D553475DED96383200
0".decode("hex")
print(efi_compressor.EfiDecompress(s, len(s)))
```

Le script retourne une chaîne de caractères :

secret data: cb41dcb1d897467o5a7fe998f11acce7

La chaîne hexadécimale est convertie en une séquence d'octets. Le code spécifié par l'utilisateur est ensuite traité dans la boucle à l'adresse 0x1000091A.

La boucle est très simple : elle appelle une fonction que nous appellerons « rotateRight » sur chaque caractère du code, et applique un « ou exclusif » avec la chaîne secrète précédemment obtenue. Le résultat de cette opération doit être chaque fois nul.

Il faut donc : rotateRight(code[i], i) = secret_data[i], soit: code[i] = rotateLeft(secret_data[i], i).

La fonction rotateRight applique une rotation à droite de chaque octet par la valeur i modulo 8 passée en paramètre. En plus de cela, elle applique un NOT final, ce qui me semble assez sournois.

La solution s'obtient alors ainsi :

```
def rotate_left(c, n):
    shift = n % 8
    c ^= 0xff
    return ((c << shift) | (c >> (8 - shift))) & 0xff

secret_data = "cb41dcb1d89746705a7fe998f11acce7".decode("hex")
out = ""
for i in range(16):
    out += chr(rotate_left(ord(secret_data[i]), i))
print(out.encode("hex"))
```

```
347d8c72720d6ec7a501583be0bccc0c
```

Le soldat veut une autre clé. Il faut donc résoudre une autre épreuve. J'ai choisi de regarder Huge.

HUGE

Le défi est contenu dans une archive Tar au format POSIX (pax). Le fichier PaxHeader/Huge nous indique qu'il s'agit d'un fichier épars, extrêmement gros : 12857414071500830 octets. Le contenu du fichier se trouve dans GNUSparseFile/Huge. C'est un fichier ELF x86-64. Pour l'analyser, j'ai choisi d'écrire un script IDA minimaliste.

Niveau 3

USB

VIDEO

Conclusion