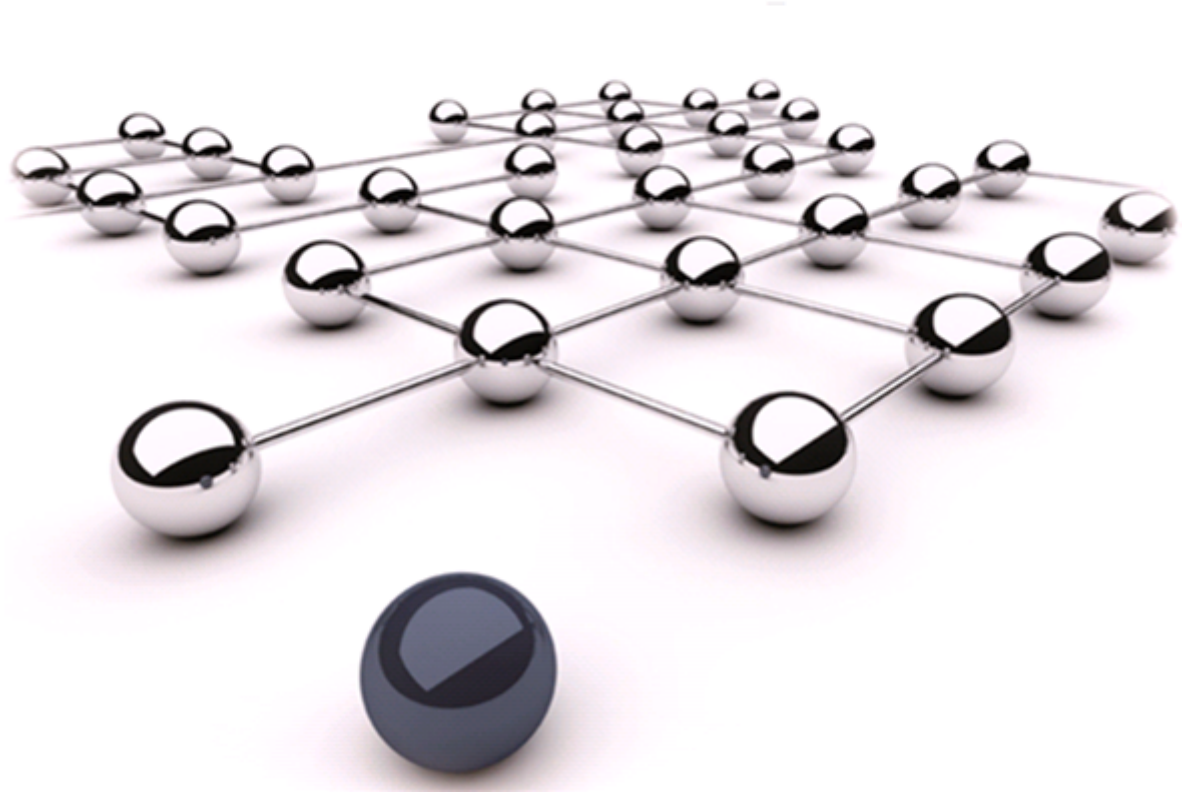
OPPIDA

EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION



Challenge SSTIC 2016

Auteur :
Damien Millescamps

Référence : OPPIDA/DOC/2016/INT/678/1.0

10/04/2016

Table des matières

1 Introduction

Comme chaque année, le principe du challenge reste le même :

Le défi consiste à résoudre les épreuves proposées dans un jeu de rôle. L'objectif est d'y retrouver une adresse e-mail (...@sstic.org).

En revanche, il semble que l'organisation des "stages" soit différente cette année :

Il n'est pas nécessaire de résoudre l'intégralité des épreuves proposées pour obtenir l'adresse e-mail.

Le jeu propose pour chaque niveau un certain nombre d'épreuves (3 aux niveaux 1 et 2 et 4 au niveau 3). Chaque épreuve permet de déchiffrer un "share" qui contient un morceau de clef (ou toute la clef si l'épreuve vaut 2 points). Pour passer au niveau suivant, il est possible de reconstituer la clef permettant de le déchiffrer à partir des informations contenues dans deux "shares" différents.
Le fichier du challenge se trouve à l'adresse : `http://static.sstic.org/challenge2016/challenge.pcap`.

L'ensemble du code présenté ici pourra être mis à disposition. Probablement par le biais de `github`.

2 Prologue

Après avoir ouvert ouvert le fichier PCAP avec `Wireshark`, on peut voir qu'il s'agit d'une trace d'un téléchargement de l'url suivante : `http://static.sstic.org/challenge2016/challenge.zip`.

On peut alors utiliser la fonction "Follow TCP Stream" de `Wireshark` afin d'obtenir le fichier téléchargé :

```
wireshark challenge.pcap
    "Follow TCP Stream"
    Select "195.154.171.95:http -> 10.69.16.64:40586"
    "Save As (Raw)" -> challenge.zip
```

L'en-tête HTTP n'est pas gênant puisque les fichiers ZIP se lisent en partant de la fin. Le contenu est un jeu en JavaScript :

```
mkdir jeu
unzip -djeu challenge.zip
```

3 Stage 1

Lors du premier niveau, on obtient 3 challenges différents. La résolution de seulement 2 épreuves sur les 3 suffit à passer au niveau suivant.

Une analyse rapide des binaires des challenges donne :
- SOS-Fant0me.zip (1 point) : capture réseau d'un RAT basique
- calc.zip (1 point) : CrackMe sur TI83+
- radio.zip (1 point) : Capture au format EBCDIC de données radio avec un RTL2832

L'épreuve radio.zip n'est pas explorée dans cette solution.

3.1 Gh0st

Le fichier extrait, "SOS-Fant0me.pcap" est une capture réseau d'une session avec une machine infectée par le Remote Access Tool Gh0st. Les trames sont préfixées par "Gh0st", suivi de la longueur des données sur 32 bits :

```
struct _ghost_hdr {
   uint8_t  prefix[5]; // "Gh0st"
   uint32_t plen;      // payload length
   uint32_t out_sz;    // decompressed data length
   void     *zdata;    // Zlib compressed data
};
```

Les données sont compressées avec Zlib. Une fois extraites, le premier octet de ces données permet d'identifier la commande C&C ou le token de réponse du bot. La liste de ces commandes peut se trouver dans cette analyse de Gh0st : `http://www.mcafee.com/fr/resources/white-papers/foundstone/wp-know-your-digital-enemy.pdf`.

Pour résoudre le challenge, une première étape consiste en :
- Extraire le flow TCP avec `Wireshark`
- Écrire un outil pour décompresser les données
- Afficher les données sous une forme lisible (ex. : [origine] commande : payload en ascii/hexa)

Cela permet d'identifier un message faisant référence à un fichier ZIP chiffré, ainsi que plusieurs téléchargements de fichiers. Pour la suite de la résolution, une seconde étape consiste en :
- Traiter la commande COMMAND_KEYBOARD de manière à reconstruire le message complet
- Gérer la commande COMMAND_DOWN_FILES afin de reconstruire les fichiers en local

Le message complet de la capture clavier donne :

```
[2016/02/27 - 23:14] New message: [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
```

Les fichiers obtenus sont les suivants :

```
how_to_rule_the_world.txt
sstic2016-stage1-solution.zip
visio_stage2.mp4
```

La vidéo est un "rickroll" classique, et le fichier texte contient de l'ASCII art faisant référence à une banane. Le mot de passe `Cyb3rSSTIC_2016` fonctionne sur le fichier `sstic2016-stage1-solution.zip`, qui donne la clef suivante :

```
unzip -PCyb3rSSTIC_2016 sstic2016-stage1-solution.zip
Archive:  sstic2016-stage1-solution.zip
 extracting: solution.txt
cat solution.txt ; echo
368BE8C1CC7CC70C2245030934301C15
```

3.2 TI83+

Le fichier extrait, “SSTIC16.8xp“ est un programme pour TI83+ en TI-Basic. Le programme peut être lancé dans un émulateur ou téléchargé sur une vraie TI83+ via le câble série. Pour l'émulateur, TilEm est disponible à l'adresse `http://lpg.ticalc.org/prj_tilem/`.

Une fois lancé le programme affiche ”Entrez le code : ”. Le code demandé est sous forme décimale. Après un long moment de calcul, le programme affiche ”PERDU” si le code n'est pas bon.

Le format 8xp est assez simple et peut se décompiler facilement. Des décompilateurs pour ce format doivent probablement exister, sinon il est possible d'en réécrire un avec une trentaine de lignes de C.

Le programme est en mode “code spaghetti“ avec des `goto` toutes les deux ou trois lignes. On remarque assez vite certaines fonctions comme la conversion décimal vers binaire, binaire vers décimal, décimal vers hexadécimal, rotation de 8 bits vers la droite et un ou-exclusif entre deux nombres binaires.

Par exemple, pour la fonction décimal vers binaire :

```
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
```

Le programme travaille sur des entiers sur 32 bits, et une liste de 256 nombres est initialisé au début. Si le code entré est correct, le morceau de code suivant s'exécute :

```
If A=3298472535:Then
Z->rand
21->S
1->X
4->Y
Output(3,7,"KEY:")
Goto 21

' ...

Lbl 21
iPart(rand*256)->A
23->S
Goto 22
```

Le code entré sert à initialiser le générateur de nombres pseudo-aléatoires qui est ensuite utilisé pour calculer la clef, octet par octet.

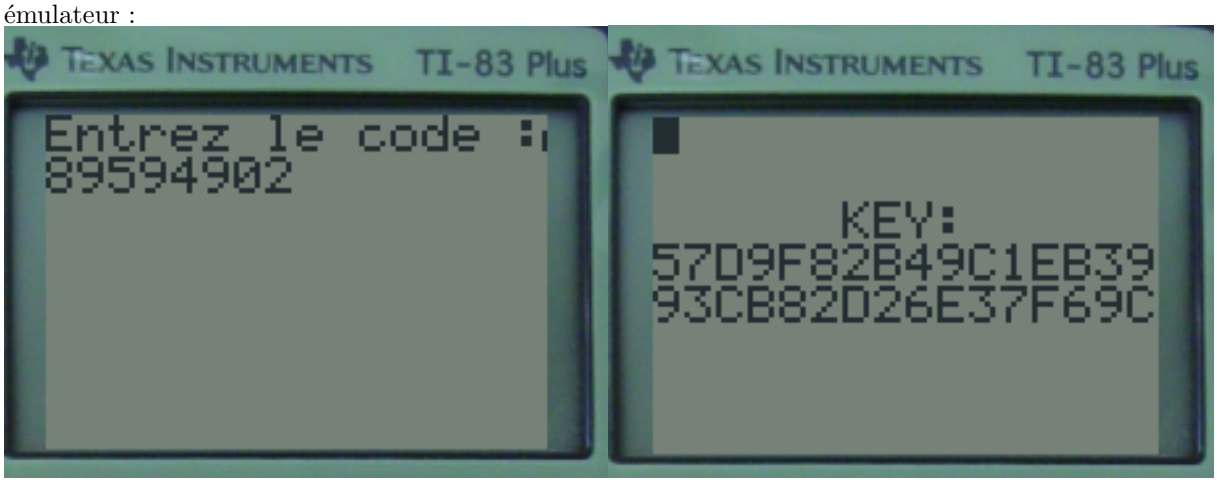
Finalement, le code C suivant permet de recréer la vérification du code entré dans la variable `Z`. La liste `list1` correspond à celle initialisé au début du programme en TI-Basic :

```
    C = 4294967295;
    for (N = 0; N < 32; N += 8) {
        C = list1[((Z & (0xFF << N)) >> N) ^ (C & 0xFF)] ^ (C >> 8);
    }
    C = ~C;
    if (C == 3298472535) {
        printf("Code: %d\n", Z);
        break;
    }
```

Une recherche exhaustive sur 32 bits permet d'obtenir le code demandé en quelques millisecondes :

```
Code: 89594902
```

Pour obtenir la clef, on peut réimplémenter le PRNG de la TI, ou soumettre le code obtenu dans un émulateur :



4 Stage 2

Un panonceau à l'entrée du niveau permet d'obtenir l'adresse `UkQhxwnHoZIIKw9lPGK5BNLg@sstic.org` pour valider le Stage 1. Une fois encore, 3 épreuves sont proposées :

— huge.zip (1 point) : contient huge.tar qui contient un ELF AMD64 de 117TB
— foo.zip (1 point) : crackme en EFI Bytecode
— loader.zip (1 point) : un binaire windows

L'épreuve loader.zip n'est pas explorée dans cette solution.

4.1 GNU Sparse File

Le fichier extrait, "huge.tar" est une archive TAR au format POSIX.1-2001/pax. Il contient le fichier ELF AMD64 "Huge", qui fait 128574140715008 octets, soit environ 117TB.
Pour l'extraire au format GNUSparseFile, il faut utiliser la commande `pax` :

```
pax -r < huge.tar
ls -CF
GNUSparseFile/
huge.tar
PaxHeader/
```

En temps normal, pour obtenir le fichier final, il faudrait exécuter la commande :

```
xsparse -x PaxHeader/Huge GNUSparseFile/Huge
```

Cependant, l'intérêt est très limité puisqu'il sera difficile de charger l'ELF en mémoire pour l'analyser ou l'exécuter.

Le fichier obtenu dans le répertoire `GNUSparseFile/` décrit le fichier de sortie, qui est composé de 24 segments non vides de 4kB et d'un segment non vide de 8kB. Le premier segment qui commence à 0 contient l'en-tête ELF avec les correspondances entre les adresses dans le fichier et les adresses mémoire de l'exécutable :

```
readelf -a 000000000000-000000001000.bin
#[...]
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x51466a42e705
#[...]
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000001000 0x00002b0000000000 0x00002b0000000000
                 0x00001ef000000000 0x00001ef000000000  R E    1000
  LOAD           0x00002affffe1000 0x000049f000000000 0x000049f000000000
                 0x0000161000000000 0x0000161000000000  R E    1000
  LOAD           0x000049effffe1000 0x0000000000020000 0x0000000000020000
                 0x00002affffe0000 0x00002affffe0000  R E    1000
#[...]
```

Ce qui donne les correspondances de segments mémoires suivants :

Fichier début	Fichier fin	Mémoire début	Mémoire fin
0x000000001000	0x1EF000001000	0x2B0000000000	0x49F000000000
0x2AFFFFFE1000	0x410FFFFE1000	0x49F000000000	0x600000000000
0x49EFFFFE1000	0x74EFFFFC1000	0x000000020000	0x2B0000000000

La solution proposée consiste à charger en mémoire aux bonnes adresses les segments de fichier, et de modifier les instructions de saut qui tombent sur des adresses non allouées de manière à les faire arriver sur la prochaine page mémoire chargée. Le but étant de pouvoir tracer le programme pour en faciliter la rétroingénierie. Le code C suivant permet de correctement charger le programme :

```
typedef int (*fct_t)(int, char **);

int main()
{
    int fd[24], i;
    fct_t Main = (fct_t)0x51466a42e705;

    for (i = 0; i < 24; i++) {
        char bf[13];
        long staddr;
        size_t slen = 4096;
        bf[12] = 0;
        memcpy(bf, fname[i], 12);
        staddr = strtoll(bf, NULL, 16);
        fd[i] = open(fname[i], O_RDWR);
        if (i==17) {
            slen = 8192;
        }
        (void)mmap((void *)staddr, slen, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE, fd[i], 0);
    }
    *(unsigned char *)0x51466a42e74f = 0x70; // check_key_fmt() (next page  7)
    *(unsigned char *)0x43abdb4a0b02 = 0xe0; // check_cst1()    (next page 14)
    *(unsigned char *)0x4a170682edeb = 0xf0; // check_cst2()    (next page 15)
    *(unsigned char *)0x06f4b0e0f382 = 0xb0; // check_cst3()    (next page 11)
    *(unsigned char *)0x49e7e541c03e = 0x30; // jump_offt()     (next page  3)
    *(unsigned char *)0x352845ab3bec = 0x40; // xor_addr_cst()  (next page  4)
    *(unsigned char *)0x59cb440c4540 = 0xa0; // xor_csts()      (next page 10)
    *(unsigned char *)0x2a7ee24aae5c = 0x70; // cos_x()         (next page  7)
    *(unsigned char *)0x099a380575be = 0x60; // final_xor()     (next page  6)

    Main(1, NULL);
}
```

Aidé de GDB pour tracer le programme, on peut reconstruire le mot de passe demandé qui se trouve être une clef de 128 bits en hexadécimal. Le format de la clef est le suivant :

0x29 [1B]	jmp_offt [1B]	0xd17e [2B]	cos_x [4B]	xor_csts [3B]	0x8c [1B]	xor_addr_cst [4B]

Octets constants Les trois premières vérification concernent les constantes. Évidemment, little-endian oblige, il faut penser à inverser l'ordre des octets lorsque la vérification se fait sur plus d'un octet. Cela permet de reconstruire progressivement la clef demandée :

```
29..............................
29..7ed1........................
29..7ed1............8c........
```

Offset de saut La vérification suivante cherche à obtenir `0x65` dans l'octet pointé par le registre `$RAX`, sachant qu'un octet de la clef donnée en entrée est utilisé comme offset d'une instruction de saut. Le binaire est largement composé de 0, qui lorsqu'ils sont pris 2 par 2 génèrent l'instruction `add BYTE PTR [rax],al`. Le registre `$AL` est initialisé avec la valeur 3, il faut donc passer sur `k` instructions tel que $3 \times k \equiv$ `0x65` (mod 256), ce qui donne $k = 119$. On obtient alors la valeur `0x89` $= \frac{(\texttt{0xC038}-(119\times 2))-(\texttt{0xBE32}+6)}{2}$.

```
29897ed1............8c........
```

XOR entre RIP et une constante La vérification suivante s'effectue sur un mot de 4 octets. Le mot de la clef donnée en entrée doit annuler une opération de ou-exclusif entre le pointeur d'instruction et une constante dans le code. On obtient rapidement le mot en question : `0xec1b3489` = `0xa9b00f5c` $\oplus$ (`0x352845ab3bd5` $\wedge$ `0xffffffff`)

```
29897ed1............8c89341bec
```

XOR entre deux constantes La même démarche que précédemment est appliquée, à la différence que deux constantes sont impliquées dans le calcul, une codée dans l'instruction de `MOV` et l'autre adressée par le pointeur d'instruction. On obtient rapidement le mot recherché, dont on connaissait déjà l'octet de poids fort (`0x8c`) : `0x8cc04ed5` = `0x59cbc8cc0b83` ⊕ `0x59cb440c4556`.

```
29897ed1........d54ec08c89341bec
```

Point fixe de cosinus Finalement, la dernière condition d'acceptation de la clef demandée en entrée vérifie $\cos(x) = x$. Le calcul se fait sur des nombres à virgule flottante codés sur 80 bits (`REAL10`), une première valeur par défaut est chargée et les 4 octets de poids fort de la mantisse subissent une opération de `XOR` avec les 4 octets manquants de la clef. La solution du point fixe, $x \approx 0.7390851332151606417$, codé en `REAL10` donne : `0x3ffebd34aeec1e716dcb`. La valeur chargée étant `0x3ffe24b878381e716dcb`, on obtient le mot manquant `0x998cd6d4` = `0x24b87838` ⊕ `0xbd34aeec`.

```
29897ed1d4d68c99d54ec08c89341bec
```

Maintenant que la clef demandée est valide, on peut exécuter le binaire reconstruit :

```
./Huge
Please enter the password: 29897ed1d4d68c99d54ec08c89341bec
The key is: E574B514667F6AB2D83047BB871A54F5
```

4.2 EFI Byte Code

Le fichier extrait, "foo.efi" est un binaire en EFI Byte Code (EBC). Une analyse rapide avec `strings` permet d'en savoir plus sur le binaire :

```
strings -el foo.efi
Invalid character when parsing data
protocol not found
Key must be exactly 32 characters
Invalid character when parsing key. Offending location:
Key must be exactly 32 characters
Missing key ?
Success!
Sorry :(
UEFI checker
```

Le programme prend une clef en entrée, sur 32 caractères. A priori il s'agit d'une clef de 128 bits en hexadécimal. On peut aussi noter un message d'erreur signifiant un possible appel à `LocateProtocol`, il sera intéressant de découvrir de quel service il s'agit en retrouvant son GUID.

Pour pouvoir l'exécuter, une possibilité est d'utiliser Open Virtual Machine Firmware (OVMF) avec `QEMU`. Pour cela, il faut compiler TianoCore dont la page concernant OVMF se trouve ici : `http://www.tianocore.org/ovmf/`.

Afin de faciliter l'analyse, il faudrait un debugger spécifique EBC. Malheureusement, le code de ce debugger a disparu entre EDK et EDK2, bien que la machine virtuelle EBC fasse quand même appel au service du debugger (GUID 5B1B31A1-9562-11D2-8E3F-00A0C969723B). Il est possible de porter les sources du debugger dans l'environnement de compilation de l'EDK2, ou simplement de récupérer une version précompilée sur le site de l'UEFI : `http://www.uefi.org/sites/default/files/resources/EBC_Debugger.zip`.

Une fois les outils prêts, il ne reste plus qu'à créer une image disque et tester le tout :

```
qemu-img create -f raw efi.img 1.4M
mkfs.vfat efi.img
mkdir tmp/
mount -o loop efi.imp tmp/
cp EbcDebugger.efi tmp/
cp foo.efi tmp/
umount tmp/
qemu-system-x86_64 --enable-kvm -L . -bios OVMF.fd -hda efi.img -nographic
```

Après avoir sélectionné “Boot Manager“ puis “EFI Internal Shell“, on peut maintenant charger le debugger afin de rendre le service accessible à la machine virtuelle EBC, puis exécuter le binaire à analyser :

```
Shell> fs0:
FS0:\> ls
Directory of: FS0:\
04/07/2016  12:33              91,552  EbcDebugger.efi
04/07/2016  12:33               6,656  foo.efi
04/07/2016  10:35               1,828  NvVars
          3 File(s)     100,036 bytes
          0 Dir(s)
FS0:\> load EbcDebugger.efi
Image 'FS0:\EbcDebugger.efi' loaded at 692E000 - Success
FS0:\> foo
EBC Interpreter Version - 1.0
EBC Debugger Version - 0.1
Break on Entrypoint
00000689EDA0: B7 37 00 00 01
00000689EDA5: 00             MOVIqd    R7, 65536
00000689EDA6: 00 06          BREAK     6
00000689EDA8: 60 00 50 80    MOVqw     R0, R0(-0,-80)
00000689EDAC: 77 36 00 00    MOVIqw    R6, 0
00000689EDB0: B9 37 4A 02 00
00000689EDB5: 00             MOVreld   R7, 0x0000024A

Please enter command now, 'h' for help.
(Using <Command> -b <...> to enable page break.)

EDB >  q
UEFI checker
Missing key ?
FS0:\>
```

Sans trop rentrer dans les détails de l'UEFI, une “system table“ est fournie en deuxième argument des binaires UEFI. Deux sous-tables sont intéressantes : “Boot Services“ et “Runtime Services“. La table “Boot Services“ fournie la méthode “LocateProtocol“ qui est une piste intéressante pour attaquer le binaire.

À l'adresse `0x634` du binaire se trouve l'appel qui correspondrait à l'offset de cette méthode. Il faut récupérer le premier argument pour obtenir le GUID qui correspond au service demandé :

```
FS0:\> foo 00000000000000000000000000000000
Break on Entrypoint
[...]
EDB > bp 0000069C4634

EDB > g
UEFI checker
Break on Breakpoint
[...]
EDB > r
  R0 - 0x0000000006725920, R1 - 0x0000000000000000
[...]
EDB > dq 0x0000000006725920
000006725920: 00000000069C53D8
EDB > dd 069C53D8
0000069C53D8: D8117CFE
EDB > dw 069C53DC
0000069C53DC: 94A6
EDB > dw 069C53DE
0000069C53DE: 11D4
EDB > dq 069C53E0
0000069C53E0: 4DC13F2790003A9A
```

Cela donne le GUID D8117CFE-94A6-11D4-9A3A-0090273FC14D, qui correspond au service “Decompress“, on peut le retrouver dans le fichier `BaseTools/Source/C/Common/Decompress.h` de l'EDK2 :

```
typedef
EFI_STATUS
(EFIAPI *EFI_DECOMPRESS_GET_INFO) (
  IN EFI_DECOMPRESS_PROTOCOL            * This,
  IN   VOID                             *Source,
  IN   UINT32                           SourceSize,
  OUT  UINT32                           *DestinationSize,
  OUT  UINT32                           *ScratchSize
  );

typedef
EFI_STATUS
(EFIAPI *EFI_DECOMPRESS_DECOMPRESS) (
  IN EFI_DECOMPRESS_PROTOCOL              * This,
  IN     VOID                             *Source,
  IN     UINT32                           SourceSize,
  IN OUT VOID                             *Destination,
  IN     UINT32                           DestinationSize,
  IN OUT VOID                             *Scratch,
  IN     UINT32                           ScratchSize
  );
```

Les appels à ces deux services se retrouvent facilement aux adresses `0x6A4` et `0x6FA`. La suite de la résolution va donc consister à retrouver la source et la taille des données à décompresser (arguments 2 et 3 de GetInfo) :

```
EDB > bp 0000069C46A4

EDB > g
Break on Breakpoint
0000069C46A4: 03 2F          CALLEX    @R7
[...]
EDB > r
  R0 - 0x0000000006725920, R1 - 0x0000000000000000
[...]
EDB > dq 0x0000000006725928
000006725928: 00000000069C5470
EDB > dq 0x0000000006725930
000006725930: 0000000000000047
```

L'adresse de la source correspond à `0x1470` dans foo.efi, et la taille est de `0x47` octets. Pour obtenir le résultat décompressé, le mieux est probablement de compiler une application “standalone“ de décompression compte tenu des fonctionnalités limités du debugger.

```
dd if=foo.efi of=secret.lzma bs=1 skip=$((0x1470)) count=$((0x47))
./EfiDecompress secret.lzma out; cat out

Comp Size = 71
Decomp Size = 92
secret data: cb41dcb1d89746705a7fe998f11acce7
```

Ces “données secrètes“ sont ensuite utilisées pour comparaison avec la clef donnée en argument dans la boucle située à l'adresse `0x91A`, après un traitement octet par octet de notre clef effectué par la fonction située à l'adresse `0x400`. La fonction en question effectue une rotation vers la droite de n bits, avec $n = i \pmod 8$, `i` étant l'indice de l'octet traité, puis prend le complément du résultat pour comparaison avec les données secrètes.

En remplaçant la rotation vers la droite par une rotation vers la gauche pour inverser la transformation, on obtient alors la clef demandée en entrée. Par exemple avec ce petit code C :

```c
#include <stdio.h>

unsigned char target[] = "\xcb\x41\xdc\xb1\xd8\x97\x46\x70\x5a\x7f\xe9\x98\xf1\x1a\xcc\xe7";
#define ROL(c, i)   (((c) << (i)) | ((c) >> (8 - (i))))

int main()
{
    int pos;
    for (pos = 0; pos < 16; pos++) {
        unsigned char c = ~ROL(target[pos], pos % 8);
        printf ("%02X", c);
    }
    printf("\n");
}
```

On peut finalement tester la clef :

```
Shell> fs0:
FS0:\> foo 347D8C72720D6EC7A501583BE0BCCC0C
UEFI checker
Success!
FS0:\>
```

5 Stage 3

Un panonceau à l'entrée du niveau permet d'obtenir l'adresse `RkrjBeyqFzsQApQhUbPvvTmJ@sstic.org` pour valider le Stage 2. Cette fois, 4 épreuves sont proposées :

— usb.zip (1 point) : un binaire windows et une image de clef USB
— ring.zip (2 points) : un binaire windows
— video.zip (1 point) : un binaire windows et une vidéo
— strange.zip (2 points) : contient un ELF IA64 pour linux 2.4 et un blob binaire

Seule l'épreuve strange.zip est explorée dans cette solution.

5.1 IA64

Un des fichiers extrait, "a.out" est un ELF pour IA64. Le second fichier nommé 196 fait 16860160 octets et semble avoir une structure composé d'éléments sur 64 bits.

La première étape consiste à désassembler le binaire afin d'obtenir une image globale de ce qu'il fait. On peut alors compiler `binutils` pour IA64 pour obtenir un `ia64-gnu-objdump` qui supporte la plateforme concernée. Pour clarifier un peu le résultat en sortie, quelques commandes à base de `grep` et de `sed -i` sur les `br.call.*` permet de délimiter les fonctions et de leur attribuer un nom.
La section `.IA_64.unwind` de l'ELF permet d'avoir la liste des fonctions internes au binaire, tandis que la section `.rela.IA_64.pltoff` donne la liste des fonctions importées. Rapidement, on peut s'apercevoir que la fonction `main` correspond à cette entrée :

```
<>: [0x400000000019c700-0x400000000019cff0], info at +0x4fc878
```

Cette fonction commence par appeler `fopen("196", "rb");` :

```
400000000019c7c0:   [MMI]       mov r32=r1;;
400000000019c7c6:               ld8 r52=[r52]                   ; "196"
400000000019c7cc:               nop.i 0x0
400000000019c7d0:   [MFB]       ld8 r53=[r53]                   ; "rb"
400000000019c7d6:               nop.f 0x0
400000000019c7dc:               br.call.sptk.many b0=.fopen;;
```

Un second appel à `fopen()` est effectué juste après :

```
400000000019c850:   [MMB]       ld8 r52=[r14]                   ; argv[1]
400000000019c856:               ld8 r53=[r53]                   ; "r"
400000000019c85c:               br.call.sptk.many b0=.fopen;;
```

La seconde étape consiste à pouvoir exécuter le binaire, et de préférence dans GDB. Pour mettre en place la plateforme, le site suivant propose toute l'information nécessaire : `http://www.gelato.unsw.edu.au/IA64wiki/SkiSimulator`.
Une fois la plateforme fonctionnelle, on peut installer des outils supplémentaires à partir d'un miroir de `Debian Woody` pour `IA64`. Par exemple, on peut trouver `GDB` ici : `ftp://ftp.gnome.org/mirror/debian-archive/pool/main/g/gdb/gdb_5.2.cvs20020401-6_ia64.deb`.

La suite de l'analyse est assez triviale. Le fichier fourni en argument doit être au format P2, représenter une image noir et blanc de 12800 pixels, de 20 pixels de hauteur, avec un commentaire entre la signature P2 et la taille de l'image. La valeur des pixels doit être comprise entre 0 et 255 mais seuls les valeurs 0 et 255 sont acceptées.
L'image est ensuite découpée en 32 blocs de 20x20 qui sont transformés en matrice 20x20 de nombres à virgule flottante à double précision. La fonction située à l'adresse `0x400000000019c410` vérifie que la matrice fournie soit bien centrée et que toute la hauteur soit utilisée à l'exception de la première ligne.

Finalement, la fonction située à l'adresse `0x400000000019afc0` appelle 161 sous-fonctions et prend en entrée la matrice précédente ainsi qu'un jeu de matrices provenant du fichier `196`. Chacune de ces fonction fait une opération de multiplication et addition pour chaque paire d'élément des matrices et calcul au moins une exponentielle. Le résultat se compose de 4 nombres à virgule flottante qui doivent tous être strictement inférieurs à une valeur de seuil : 0.15. Si c'est le cas, le bloc suivant de l'image fournie en

entrée est transformé en matrice 20x20, puis les même calculs sont effectués avec de nouvelles matrices provenant du fichier `196`.

Pour récapituler, le binaire prend en entrée une image contenant 32 symboles qui sont comparés par rapport à un fichier d'entrée pour valider si le symbole correspond ou non afin de passer au suivant. Les calculs effectués correspondraient à un classificateur à base de réseau de neurones, et 32 symboles pourraient correspondre à une clef de 128 bits en hexadécimal. La fonction située à l'adresse `0x400000000019afc0` serait donc une fonction d'OCR à base de PNN (Probabilistic Neural Network).

Une solution possible consiste à reconstituer la clef en utilisant une police de caractères monospace et en jouant sur du "Gaussian blur" sur les glyphes couplé avec un script `GDB` pour retrouver quel caractère est à quelle position. Le script `GDB` consiste à mettre un breakpoint à l'adresse `0x400000000019cde0`, comparer les registres `R36`, `R38`, `R39` et `R41` par rapport à la valeur de seuil de 0.15, puis à directement sauter à l'adresse `0x400000000019ce90` pour tester le glyphe suivant.
Au final, cette méthode ne fonctionnera pas car la clef est en réalité en décimal, et non pas hexadécimal comme on pourrait s'y attendre, générant de nombreux faux positifs. L'auteur de l'épreuve n'a probablement pas jugé utile de générer des données d'OCR qui intègrent les caractères `[A-F]` pour l'hexadécimal.

En fait, la police de caractère est insérée de manière cryptique dans le fichier `196`, sous la forme de matrice 20x20 de nombres proches de 0 ou de 1 tous les 526880 octets, permettant de se rendre compte que la clef est composée exclusivement des caractères 0 à 9. Muni de cette information et de la forme exacte des glyphes attendus, il ne reste plus qu'à bruteforcer la clef.

Le code C pour générer du P2 est trivial, ce format devant son existence aux cours d'initiation à la manipulation d'image en programmation. Pour préparer le bruteforce, on peut commencer par parser un argument de taille variable correspondant au début de la clef :

```
unsigned char orders[32];
int parse_arg(int ac, char **av)
{
    int cnt = 0, j;
    memset(orders, 10, 32);
    if (ac > 1) {
        cnt = strlen(av[1]);
        for (j = 0; j < cnt; j++) {
            char nb[2]; nb[1] = 0; nb[0] = av[1][j];
            orders[j] = atoi(nb);
        }
        if (cnt < 32) orders[j] = 11;
    } else orders[0] = 11;
}
```

Puis la fonction de chargement de la police de caractère pour le bruteforce :

```
unsigned char pics[11][20][20];
void load_font()
{
    int j, idx;
    FILE *fp;
    fp = fopen("196", "r");
    for (j = 0; j < 10; j++) {
        double db;
        fseek(fp, j * 526880, SEEK_SET);
        for (idx = 0; idx < 400; idx++) {
            fread(&db, 8, 1, fp);
            pics[(j + 1) % 10][idx / 20][idx % 20] = db > 1. ? 0 : 255;
        }
    }
    for (j = 0; j < 400; j++) pics[10][j/20][j%20] = 255;
    fclose(fp);
}
```

Pour finir, la génération des images :

```
int main(int ac, char **av)
{
    int j;
    load_font();
    parse_arg(ac, av);
    for (j = 0; j < 10; j++) {
        int i;
        char ff2[32];
        FILE *fs;
        sprintf(ff2, "try%X.pgm", j);
        fs = fopen(ff2, "w+");
        fprintf(fs, "P2\n#\n640 20\n255\n");
        for (i = 0; i < 20; i++) {
            int k;
            for (k = 0; k < 32; k++) {
                int l;
                for (l = 0; l < 20; l++) {
                    if (orders[k] == 11) fprintf(fs, "%d\n", pics[j][i][l]);
                    else fprintf(fs, "%d\n", pics[orders[k]][i][l]);
                }
            }
        }
        fclose(fs);
    }
    return 0;
}
```

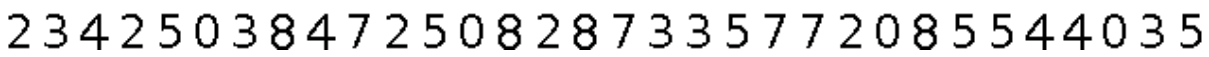
Après avoir installé `GCC` sur la plateforme `IA64`, on peut compiler le générateur d'images au format demandé par le binaire.

```
dmesg -n4
gcc-2.96 -o bf_img bf_img.c
```

Finalement, l'épreuve se résout avec ce simple "one-liner" :

```
KEY=""; for ((pos=2; pos < 33; pos++)); do ./bf_img $KEY; for i in try*; do res=$(./a.out $i |
    egrep '^\.$' | wc -l); if [ $res -eq $pos ]; then nc=$(echo $i | sed 's:try::;s:\.pgm::');
    KEY=${KEY}${nc}; break; fi; done; done; ./bf_img $KEY; for i in try*; do ./a.out $i | /bin/
    egrep '^pass$' && (nc=$(echo $i | sed 's:try::;s:\.pgm::'); KEY=${KEY}${nc}; echo $KEY;
    break); done
pass
234250384725082873357720855440 35
```

Le résultat s'obtient en quelques minutes :



6 Épilogue

Le seul fichier présent dans le dernier niveau est final.txt, qui annonce une bonne nouvelle :

```
Coucou !

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

Pour la dernière étape, un `rot13` devrait faire l'affaire :

```
echo "I01p1 y'4qe3553 z41y : 8Y6d5j9Vy88HUGHfGSKsJvqA@ffgvp.bet" | rot13
V01c1 l'4dr3553 m41l : 8L6q5w9Il88UHTUsTFXfWidN@sstic.org
```

FIN.