Challenge SSTIC 2016

Erwan Hamon
@r1hamon

Avril, 2016

1 Introduction

Voici ma solution au challenge du SSTIC 2016 qui est récupérable ici: http://communaute.sstic.org/ChallengeSSTIC2016. Le code sera disponible après la conférence sur https://github.com/r1git. Les épreuves résolues dans chacun des niveaux sont: Level 1: SoS-Fant0me & calc. Level 2: foo & huge. Level 3: strange

2 Récupération du fichier

Le fichier challenge.pcap récupéré est une capture réseau d'un téléchargement du fichier challenge.zip. Ce téléchargement a apparemment eu quelques ratés et des packets ont été réémis. Le script extract.py en annexe réassemble les payload des paquets dans le bon ordre en suivant les numéros de séquence. On me dit dans l'oreillette que Wireshark exporte cela très bien et que mon script est une perte de temps. Bref...

```
$ python extract.py > challenge.zip
```

Après décompression, on trouve un jeu de rôle à l'ancienne basé sur le framework RPG JS. En se promenant dans le jeu, on dialogue avec des personnages plus vrais que nature qui vous donnent des épreuves à résoudre. Chaque épreuve consiste à trouver des clefs hexadécimales de 32 caractères. Il faut 2 clés par niveau, sauf au niveau 3 ou certaines clefs valent double.
Les sections suivantes décrivent les épreuves choisies pour chaque niveau.

3 Level 1

3.1 SOS-Fant0me

Il s'agit une nouvelle fois d'une capture réseau. On remarque que beaucoup de paquets contiennent la chaîne "Gh0st". Une rapide recherche sur internet indique que c'est la signature d'un Trojan: Gh0st Rat. Ce Trojan permet à un client distant de contrôler la machine infectée. Le protocole est simple: des indications de taille de paquets, la chaîne "Gh0st" suivie des données compressées. On retrouve des échanges amusants, un mot de passe de fichier zip, un fichier vidéo et un fichier zip protégé par mot de passe. Le

mot de passe trouvé déchiffre bien le fichier zip qui contient une des clés recherchées. Le script sc.py affiche les échanges, ignore les gros fichiers (dont vidéo), sauvegarde le zip dans code.zip.

```
$ python sc.py
[....suprise...essayez...]
$ unzip code.zip
Archive:  code.zip
[code.zip] solution.txt password:
 extracting: solution.txt
$ cat solution.txt
368BE8C1CC7CC70C2245030934301C15
```

3.2 calc

Après décompression du fichier calc.zip, on obtient un fichier SSTIC16.8xp contenant la chaîne TI83F. Google nous apprend qu'il s'agit d'un programme de la vieille calculatrice TI83 de Texas Instrument. On peut récupérer un émulateur (http://lpg.ticalc.org/prj_tilem/) et lancer le programme en question. Le programme attend un entier et si on entre le bon l'utilise pour donner la clef recherchée. Le programme est court, légèrement spaghetti et peut être porté facilement en python. Il effectue 4 tours de calculs sur la valeur entrée en utilisant un tableau de valeurs de substitution suivi de xor et décalages. Si l'état final vaut une valeur test, l'entrée est la bonne. Il suffit donc de bruteforcer les 2^{32} entrées possibles jusqu'à obtenir cette valeur. Le script brute_TI.py en annexe montre le portage en python et donne l'entier cherché. On le donne à manger au programme original dans l'émulateur qui renvoie la seconde clé.

```
$ python brute_TI.py
Gagne: 89594902
```

4 Level 2

4.1 huge

Ici il s'agit du tar d'un binaire x64 démesurément grand. L'outil tar permet en effet d'archiver des "sparse files", des fichiers avec des trous. Ces trous sont remplis de 0 par tar lors de l'extraction. Le binaire est donc constitué de petits bouts de codes (ceux présents dans l'archive) séparés par de vastes plages de 0. Les "0" ayant le mauvais gôut d'être du code parfaitement valide.

On extrait le binaire du tar manuellement donc sans l'expansion des plages de "0":

```
$ python ex.py
$ readelf -l huge_1

Elf file type is EXEC (Executable file)
Entry point 0x51466a42e705
There are 3 program headers, starting at offset 64

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
```

L'approche choisie a été de reconstituer un binaire dont le code existant serait valide. Pour cela, on va ajouter autant de program headers au binaire qu'il y a de morceaux de code déclarés dans le tar. Le mapping mémoire sera donc le même que l'original, sauf que les trous ne seront pas mappés.

Le fichier obtenu s'exécute correctement, demande un mot de passe et segfault. En effet, le binaire parcoure les fameux "0" comme du code valide et arrivé en fin de zone, se retrouve dans une zone non mappé par les program headears d'où le segfault.

Il s'agit ensuite de rajouter des jmp dans le code afin de sauter par-dessus ces "trous". Cela a été fait au fur et à mesure du reverse du binaire qui vérifie progressivement des parties de la clé entrée. Le programme final segfault donc toujours sur les mauvaises clés mais pas sur la bonne. Le script ph.py fait les 2 étapes, création des program headers et ajout des jmp:

```
$ python ph.py
0x20000 >> 0xe97ea9c0000 block  15 warning: 16581727076352
map  0x11000 to 0x6f4b0e00000 size 4096
[...]
$ readelf -l huge_2
Elf file type is EXEC (Executable file)
Entry point 0x51466a42e705
There are 24 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000011000 0x000006f4b0e00000 0x000006f4b0e00000
                 0x0000000000001000 0x0000000000001000  R E    1000
  LOAD           0x0000000000012000 0x000006f4b0e0f000 0x000006f4b0e0f000
                 0x0000000000001000 0x0000000000001000  R E    1000
[... 24 program headers en tout ... ]
  LOAD           0x000000000000f000 0x000059cb440c4000 0x000059cb440c4000
                 0x0000000000001000 0x0000000000001000  R E    1000
  LOAD           0x0000000000010000 0x00005a4815650000 0x00005a4815650000
                 0x0000000000001000 0x0000000000001000  R E    1000
```

Le reverse ne pose pas de problème particulier, surtout en analyse dynamique puisqu'elle est à présent possible, à 2 exceptions. La première difficulté vient d'un test effectué sur le nombre d'exécutions du code représenté par des "0". En effet chaque exécution ajoute 3 à une valeur et la cible à atteindre n'est pas sur les multiples de 3. Il s'agit cependant

d'une valeur sur un octet et l'astuce est donc de travailler modulo 256 pour retomber sur la valeur souhaitée. La seconde difficulté consiste en un calcul flottant effectué à 10 octets de précision par des instructions dédiées. Il s'agit de résoudre $cos(x) = x$. Cependant, sur les 10 octets recherchés, seul 4 dépendent de l'entrée utilisateur. L'approche a donc été de dupliquer ce code et le bruteforcer jusqu'à trouver le résultat.

```
$ gcc -o brute brute_float.c
$ python patch_float.py
$ chmod +x brute_2
$ ./brute_2
en est a 0
[...]
en est a 3900000000
Victory: ec ae 34 bd
```

Une fois toutes les valeurs d'entrées trouvées, le programme nous renvoie la clé recherchée.

4.2 foo

Après décompression, on obtient le fichier foo.efi. L'outil file confirme qu'il s'agit d'une application EFI. Un émulateur est disponible ici: https://github.com/tianocore/edk2/tree/master/EmulatorPkg. Cet émulateur dispose d'un gdb intégré avec des définitions de structure permettant de voir les détails de l'état de l'émulateur. L'approche suivie a été d'instrumenter l'émulateur pour le faire produire des traces d'execution (patchs complets sur github) du genre:

```
@@ -3848,6 +3878,7 @@ ExecuteXOR (
   IN UINT64       Op2
   )
 {
+  DEBUG ((EFI_D_INFO, "\n%x XOR %llu(%x), %llu (%x)", VmPtr->Ip, Op1, Op1, Op2, Op2));
   return Op1 ^ Op2;
 }
```

Après avoir désactiver l'ASLR sur l'hôte pour avoir des traces comparables et à coup de diff des différentes traces, on reverse le binaire. Il y sûrement mieux, mais je positionnais mes breakpoints avec le patch suivant:

```
@@ -1481,7 +1480,9 @@ EbcExecute (
     // function pointer is null then generate an exception.
-    //
+    /*if(VmPtr->Ip == (UINT8 *)0x48D8155C)
+       asm("int $3");*/
     ExecFunc = (UINTN) mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction;
     if (ExecFunc == (UINTN) NULL) {
       EbcDebugSignalException (EXCEPT_EBC_INVALID_OPCODE, EXCEPTION_FLAG_FATAL, VmPtr);
```

Le reverse en lui-même ne pose pas de problème particulier. On porte l'algo, on bruteforce, on a la clé: brute_efi.py.

```
$ python brute_efi.py
34 7d 8c 72 72 0d 6e c7 a5 01 58 3b e0 bc cc 0c
```

5 Level 3

5.1 strange

Deux fichier sont décompressés à cette étape. Le premier est un exécutable identifié par l'outil file comme un exécutable IA64. Une architecture bien documentée dont l'originalité est qu'elle exécute les instructions par groupes (des bundles) tant que leurs résultats n'ont pas d'inter-dépendances. De plus, elle dispose d'un groupe de registres conditionnels dont dépendent l'exécution des instructions qui leurs sont indéxées. Enfin, un grand nombre de registres et une rotation des registres lors des appels de fonctions achèvent de justifier le titre choisi pour cette épreuve.

L'approche suivie a été d'utiliser ski (http://ski.sourceforge.net/), l'émulateur IA64 d'hp. Il faut un patch pour permettre la compilation:

```
diff --git a/src/linux/syscall-linux.c b/src/linux/syscall-linux.c
index 722aed1..a112473 100644
--- a/src/linux/syscall-linux.c
+++ b/src/linux/syscall-linux.c
@@ -72,7 +72,7 @@
 #include <sys/uio.h>

 #include <linux/serial.h>
-#include <asm/page.h>
+//#include <asm/page.h>
 #include <asm/unistd.h>

 #include "std.h"
@@ -2252,8 +2252,8 @@ doSyscall (HWORD num, REG arg0, REG arg1, REG arg2, REG arg3, REG arg4,
        case TIOCSERSETMULTI:   /* Set multiport config */
        case TIOCMIWAIT:        /* wait for a change on serial input line(s) */
        case TIOCGICOUNT:       /* read serial port inline interrupt counts */
-       case TIOCGHAYESESP:     /* Get Hayes ESP configuration */
-       case TIOCSHAYESESP:     /* Set Hayes ESP configuration */
+       //case TIOCGHAYESESP:   /* Get Hayes ESP configuration */
+       //case TIOCSHAYESESP:   /* Set Hayes ESP configuration */
        case SIOCRTMSG: /* call to routing system       */
        case SIOCSIFLINK:       /* set iface channel            */
        case SIOCGIFMEM:        /* get memory address (BSD)     */
```

Un patch pour corriger l'affichage ncurse:

```
diff --git a/program.c b/program.c
index 26da97c..6bf6151 100644
--- a/program.c
+++ b/program.c
@@ -268,11 +268,11 @@ char *prgwLine(ADDR ofs, unsigned *srcrows, unsigned *asmrows)
        dasInit(DasPseudoOps|DasTemplate|DasRegNames, prgColumns - 20);
        dasBundle(&bndl, i0Str, i1Str, i2Str);
        ipp = instPtr(ofs, i0Str);
-       p += sprintf(p, "%s%c%c %s %s\r\n", srcp, bpn, ipp, buf, i0Str);
+       p += sprintf(p, "%s%c%c %s %s\n", srcp, bpn, ipp, buf, i0Str);
        if (i1Str[0]) {         /* not MLX */
            bpn = ((i = isbpt(ofs + 4)) >= 0) ? (i + '0') : ' ';
            ipp = instPtr(ofs + 4, i1Str);
-           p += sprintf(p, "%c%c %16s %s\r\n", bpn, ipp, "", i1Str);
+           p += sprintf(p, "%c%c %16s %s\n", bpn, ipp, "", i1Str);
            bpn = ((i = isbpt(ofs + 8)) >= 0) ? (i + '0') : ' ';
            ipp = instPtr(ofs + 8, i2Str);
        } else {                /* MLX */
@@ -280,7 +280,7 @@ char *prgwLine(ADDR ofs, unsigned *srcrows, unsigned *asmrows)
                    ? (i + '0') : ' ';
            ipp = instPtr(ofs + 4, i2Str);
        }
-       p += sprintf(p, "%c%c %16s %s\r\n", bpn, ipp, "", i2Str);
+       p += sprintf(p, "%c%c %16s %s\n", bpn, ipp, "", i2Str);
        *asmrows = i1Str[0] ? 3 : 2;
    } else {
 xxx:
@@ -419,10 +419,10 @@ BOOL dasmDump(unsigned argc, char *argv[])

                dasAddr = adr;
                dasBundle(&bndl, i0Str, i1Str, i2Str);
-               p += sprintf(p, "%s %s\n", adrStr, i0Str);
+               p += sprintf(p, "%s %s", adrStr, i0Str);
                if (i1Str[0])
-                   p += sprintf(p, "%16s %s\n", "", i1Str);
-               p += sprintf(p, "%16s %s\n", "", i2Str);
+                   p += sprintf(p, "%16s %s", "", i1Str);
+               p += sprintf(p, "%16s %s", "", i2Str);
            } else
                p += sprintf(p, "%s xxxxxxxx\n", adrStr);
        }
```

Et un patch pour faire tracer les commandes exécutées par ski dans un fichier:

```
diff --git a/sim.c b/sim.c
index f2f953e..c8102d6 100644
--- a/sim.c
+++ b/sim.c
@@ -62,6 +62,8 @@
 #include "instinfo.h"
 #include "icnt_core.h"
 #include "os_support.h"
+#define NPROC 1
+#include "program.h"

 #define ALARM

@@ -335,6 +337,10 @@ static BOOL instDecode(ADDR adr)

     pa = adr & ~(ADDR)0xF;

+    FILE *myf = fopen("trace.asm","a+");
+    fprintf(myf, "%s", prgwLine(pa, &i, &i));
+    fclose(myf);
+
     i = BitfX(pa,52,10);
     /* XXX - Do we need to lookup the whole page? Call memIRd instead? */
     /* XXX - What if page is not mapped? */
```

Le reverse nous apprend qu'un fichier est attendu en argument. Plusieurs contraintes lui sont demandées. A chaque contrainte correctement répondue, le programme incrémente le code d'erreur renvoyé. Ce qui est bien urbain.

Cela commence par des contraintes simples, puis la présence de 2 entiers dont le produit doit valoir 0x3200. Les données lues dans le fichier d'entrées ne peuvent être ensuite que des 0 ou des 255. On se rend compte qu'il s'agit en fait d'éléments de matrices 20x20 lues de façon entrelacées. Des tests sont effectués sur la matrice, par exemple que sa première ligne est pleine de 255, la dernière optionnellement, que le nombre de colonnes à gauche et à droites pleines de 255 sont au même nombre (ou +1 à gauche)...

Puis des données sont lues dans le second fichier présent dans l'épreuve (par bloc de 0x10144 octest). Ensuite une très longue série d'instructions manipule ces blocs de 20x20 un par un, 4 octets en sont déduis et de leurs valeurs dépend l'arrêt du programme. Cela est effectué 32 fois. Il n'y a pas d'autre entrée utilisateur possible. On sait que l'on doit trouver 32 caractères comme dans chaque épreuve et on ne dispose que de ces 32 blocs de 20x20.

Sans autre issue, on fait donc l'hypothèse que ces blocs 20x20 de valeurs binaires sont en fait des représentations graphiques de caractères. Ce que tend à faire penser les premiers tests sur les matrices et dans cette hypothèse le large code et le second fichier seraient les éléments d'une sorte d'OCR qui reconnaitrait une gamme de caractères. Le haiku fournit par le personnage qui donne l'épreuve nous incite à penser que seul des chiffres sont utilisés dans cette épreuve.

On peut donc demander à sa moitié de dessiner des caractères en ASCII dans un fichier texte pour aider à la résolution. L'idée pouvait paraître séduisante, mais l'effet de bord principal est que celle-ci raconte maintenant à tout le monde qu'elle a résolu le challenge du SSTIC et que c'est vraiment pas aussi compliqué que ça. Pas de quoi

gâcher 3 week-ends en tout cas.

Exemple de matrice représentant un "2":

```
11111111111111111111
11110000000000001111
11100000000000000111
11100000000000000111
11100011111111000111
11100011111111000111
11111111111111000111
11111111111111000111
11111111111111000111
11111111111000000111
11111111110000001111
11111110000001111111
11111110000111111111
11111000001111111111
11110000011111111111
11110000111111111111
11100000000000000111
11100000000000000111
11100000000000000111
11111111111111111111
```

Le programme check_strange.py prend ce fichier en entrée, essaye tous les symboles à toutes les étapes grâce à bski, la version batch de ski en se basant sur le code de retour. Il retourne alors tous les "hits" en fonction de la position du caractère. Le programme ne parvient pas à différencier les 5 et les 6, les 3 et les 9. Parfois les 1 et les 2. Et un 8 est douteux. Au mieux on se retrouve avec 2048 clés possibles. Au pire 10 fois plus si le 8 douteux est un autre caractère.

```
$ python check_strange.py
Trying [...]
1 ['0x2']
2 ['0x3', '0x9']
3 ['0x4']
4 ['0x2']
5 ['0x5', '0x6']
6 ['0x0']
7 ['0x3', '0x9']
8 ['0x8']
9 ['0x4']
10 ['0x7']
11 ['0x1', '0x2']
12 ['0x5', '0x6']
13 ['0x0']
14 ['0x8']
[...]
```

Enfin, pour essayer les 2048 possibilités restantes, on attaque le JDR: en beautifiant le script sham.min.js, on apprend que c'est le partage de clé secrète de Shamir qui est utilisé par le garde pour vérifier nos résultats aux épreuves. Lorsqu'on lui soumet une clé, il tente de déchiffrer les "shares" du protocole qu'il a pour chaque niveau. On va donc faire de même. Le navigateur en mode développeur nous donne accès aux IV et "shares" chiffrés dans la variable ssm_data (lev3.shares en annexe). Le script brute.py essaye ensuite de les déchiffrer avec les clés possibles et nous retourne la bonne.

```
$ python brute.py
Found key: 234250384725082873357720855440 35
[{"x":3,"y":"a7eee9045d4f96ddaf737675d3c373eb"},{"x":4,"y":"a78801e89078188318c29d5d23e265d3"}]
2048 keys tested
```

(On retrouve les coordonnées de 2 points du polynôme du protocole de Shamir dans le JSON déchiffré.)

Fin.

Annexe

Récupération

extract.py

```
from scapy.all import *
import sys

packets = rdpcap("challenge.pcap")

seq_next = packets[15][IP].seq
f = sys.stdout

mem = []
lengt = 0
i = 0
size = len(packets)

while(lengt < 52331069):
        # Regarder dans les paquets memorises
        for m in mem:
                if m[IP].seq == seq_next:
                        leng = len(m[IP].load)
                        lengt += leng
                        seq_next += leng
                        f.write(m[IP].load)
                        mem.remove(m)
                        break
        # Chercher le paquet suivant
        while(i<size):
                if(packets[i][IP].src == "195.154.171.95"):
                        if packets[i][IP].seq == seq_next:
                                leng = len(packets[i][IP].load)
                                lengt += leng
                                seq_next += leng
                                f.write(packets[i][IP].load)
                        else:
                                # Paquet dans le desordre, memoriser.
                                mem.append(packets[i])
                        break
                i += 1
        i += 1
```

Level 1

sc.py

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

ini = rdpcap("SOS-Fant0me.pcap")

i = 0
for p in ini:
        i += 1
        try:
                if(p.load):
                        d = p.load
```

```
                        if d[0:5] == 'Gh0st':
                                leng = unpack("<I", d[9:13])[0]
                                #Plusieurs fichiers plus grands sont presents
                                try:
                                        print zlib.decompress(d[13:])
                                # On les skip salement
                                except:
                                        pass
                                # Le fichier zip est le 522eme
                                if(i==522):
                                        f = open("code.zip", "w")
                                        f.write(zlib.decompress(d[13:]))
                                        f.close()
        except AttributeError:
                continue
```

brute_TI.py

```
from multiprocessing import Pool

# Table de substitution (256 elements)
L1 = [0, 1996959894, 3993919788, 2567524794,....]

def calc(inp):
        str1 = 4294967295
        N = 0
        for N in range(0,4):
                start = str1
                str1 = (str1 & 0xff) ^ ((inp>>(N*8)) & 0xff)
                str1 = L1[str1]
                str2 = start >> 8
                str1 = str1 ^ str2

        return str1 ^ 0xffffffff


def brute(a):
   size = pow(2,32)/4
   for inp in xrange(a*size, (a+1)*size):
        if calc(inp) == 3298472535:
                print "Gagne:", inp


p = Pool(4)
p.map(brute, range(0,4))
```

Level 2

brute_efi.py

```
def calc(val, cnt):
        i = cnt % 8
        loc = val
        val = (val >> i) & 0xffffffff
        loc = (loc << (8-i)) & 0xffffffff
        fun = ~(val | loc) & 0xff
        return fun

stat = [0xcb, 0x41, 0xdc, 0xb1, 0xd8, 0x97, 0x46, 0x70, 0x5a, 0x7f, 0xe9, 0x98, 0xf1, 0x1a, 0xcc, 0xe7]
for i in range(0,16):
```

```
        for k in range(0,255):
                if calc(k, i) == stat[i]:
                        print '{0:02x}'.format(k),
```

ex.py

```
data = open("huge.tar").read()

start = data.find("ELF")
open("huge_1","w").write(data[start-1:])
```

ph.py

```
import struct

data = open("huge.tar").read()[0x603:0x7ea]
#Extrait les zones sparses du tar:
#La liste des offsets
off = map(int, data.split('\x0a')[0::2])
#Et leur taille (4096 sauf 1 a 8192)
spare = map(int, data.split('\x0a')[1::2])

#Correspondance adresses virtuels et physique
# voir readelf -l huge_1
vir = [0x49f000000000, 0x00002b0000000000, 0x0000000000020000]
load = [0x00002affffe1000 ,0x0000000000001000, 0x000049effffe1000]

#Remplace les octets en position start dans ori par p
def repl(ori, start, p):
        return ori[0:start] + p + ori[start+len(p):]

#Traduit une adresse virtuel en adresse physique
def toReal(add):
        assert(add>=0)
        b = 0
        while(add < vir[b]):
                b += 1
        return add - vir[b] + load[b]

#Traduit une adresse dans huge en adresse virtuelle
def toS(add):
        assert(add>=0)
        b = len(off) - 1
        while(add < off[b]):
                b -= 1
        block = b
        b -= 1
        while(b>=0):
                add -= (off[b+1]-off[b]) - spare[b]
                b -= 1

        return add, block

ff = open("huge_2", "w")
wlen = 0

#Cree un program header
def write_ph(file_off, virtaddr, size):
        global wlen
        buff =  "\x01\x00\x00\x00\x05\x00\x00\x00"
```

```
        buff += struct.pack("<Q", file_off)
        buff += struct.pack("<Q", virtaddr)
        buff += struct.pack("<Q", virtaddr)
        buff += struct.pack("<Q", size)
        buff += struct.pack("<Q", size)
        buff += "\x00\x10\x00\x00"
        buff += "\x00\x00\x00\x00"
        print "header size", len(buff)
        ff.write(buff)
        wlen += len(buff)


#Genere les program header necessaires
def gen_ph((add, sizeload)):
        in_big = toReal(add)
        (in_real, b) = toS(in_big)
        print hex(add), ">>", hex(in_real), "block ", b, "warning:", in_big-off[b-1]
        b+=1 #Attenton si en milieu de zone legale
        while(b+1<len(off) and off[b+1]-in_big < sizeload):
                fi = toS(off[b])[0]
                print "map ", hex(fi), "to", hex(add+off[b]-in_big), "size", spare[b]
                write_ph(fi, add+off[b]-in_big, spare[b])
                b+=1
        fi = toS(off[b])[0]
        print "map ", hex(fi), "to", hex(add+off[b]-in_big), "size", spare[b]
        write_ph(fi, add+off[b]-in_big, spare[b])

ph = [(0x0000000000020000, 0x00002affffe0000), (0x00002b0000000000, 0x00001ef000000000), (0x000049f000000000, 0x0000161

el = open("huge_1").read()
#Ajoute 0x18 program headers
el = repl(el, 0x38, '\x18')
ff.write(el[0:0x40])
wlen+=0x40
for p in ph:
        gen_ph(p)

#Ajoute tous les jmp par dessus les "trous"
el = repl(el, 0x1500c, "e937750000".decode("hex"))
el = repl(el, 0x700c, "e9dd0a0000".decode("hex"))
el = repl(el, 0xb00c, "e9cfed0000".decode("hex"))
el = repl(el, 0x1100c, "e95ff30000".decode("hex"))
el = repl(el, 0x800c, "e908be0000".decode("hex"))
el = repl(el, 0x300c, "e9bd3b0000".decode("hex"))
el = repl(el, 0xe00c, "e913450000".decode("hex"))
el = repl(el, 0x1700c, "e913ae0000".decode("hex"))
el = repl(el, 0x1300c, "e996750000".decode("hex"))

ff.write(el[wlen:])
ff.close()
```

brute_float.c

```
#include <stdio.h>

unsigned int floatf(char *input) {

        // Les instructions ne sont pas exactement les bonnes.
        // Le fichier patch_float.py les remplace par les originaux
        __asm__(
            "mov %rsp, %rbx;"
```

```
            "mov %rdi, %rsp;"
            "wait;"
            "fnclex;"
            //"fld tbyte ;"
            "fnstsw %ax;"
            "fld %st(0);"
            "fcos;"
            "fcompp;"
            "wait;"
            "fnstsw %ax;"
            "and %ax, 0xffdf;"
            "mov %rbx, %rsp;"
        );
}

main() {
        unsigned int i;
        int res;
        // Les 10 octets récupérés en analyse dynamique
        // seuls les octets 4, 5, 6 et 7 sont bruteforcés
        unsigned char buff[] = {0xcb, 0x6d, 0x71,0x1e, 0x93, 0xb5, 0x57, 0x25, 0xfe, 0x3f, 0x33};

        // En fait c'est un while forever... puisque 0xffffffff+1=0
        for(i=3800000000; i<=0xffffffff; i++) {
                if (i%100000000 == 0)
                        printf("en est a %u\n", i);
                buff[4] = (i & 0xff000000) >> 24;
                buff[5] = (i & 0x00ff0000) >> 16;
                buff[6] = (i & 0x0000ff00) >> 8;
                buff[7] = i & 0x000000ff;
                res = floatf(buff);
                // Le test demandé
                if ((res & 0xffff) == 0x4000) {
                        printf("Victory: %x %x %x %x\n", buff[4], buff[5], buff[6], buff[7]);
                }
        }
}
```

patch_float.py

```
place = 0x53b

#Extrait le code en calcul flottant de l'original
data = open("huge_1").read()[0x18e41:0x18e55]+'\x90\x90'
dest = open("brute").read()

#Patch le binaire avec
buff = dest[0:place]+data+dest[place+len(data):]
buff = buff[0:place+0x56f-0x569] + '\x00' + buff[place+0x56f-0x569+1:]

open("brute_2","w").write(buff)
```

Level 3

check_strange.py

```
import subprocess
import random

def trans(n):
```

```
        if n == "255":
                return "1"
        else:
                return "0"

def trans2(n):
        if n == "1":
                return "255"
        else:
                return "0"

# Affiche une matrice 20x20 pour controle visuel
def np(tab):
        for i in range(20):
                print "".join(map(trans, tab[i*20:(i+1)*20]))

# Cree le fichier attendu
def gen_file(sol, tk, n, tab):
        f = open(n, "w")
        f.write("P2\n#\n640 20\n255\n")
        for i in range(20):
                for j in sol:
                        f.write("\n".join(tk[j][i*20:(i+1)*20]))
                        f.write("\n")
                f.write("\n".join(tab[i*20:(i+1)*20]))
                f.write("\n")
                for i in range(620-(len(sol)*20)):
                        f.write("0\n")
        f.close()

# Lance bski
def exec_chal(keyf):
        try:
                return subprocess.check_output(["./bski -noconsole -stats a.out "+ \
                        keyf], stderr=subprocess.STDOUT, shell=True), 0;
        except subprocess.CalledProcessError, e:
                return e.output, e.returncode

# Charge le fichier "ascii art"
def load():
        f = open("haiku", "r")
        res = []
        i = 0
        line = f.readline().strip()
        while(line != ""):
                char = []
                while(line[0:2] != "=="):
                        char += map(trans2, line)
                        line = f.readline().strip()
                assert(len(char) == 400)
                res.append(char)
                line = f.readline().strip()
        f.close()
        return res

# Les matrices a essayer
try_keys = []
try_keys = load()
# Affichage visuel pour controle
for tt in try_keys:
        np(tt)
        print("==")
```

```
# Les matrices repondant a chaque etape pour tester a n'importe quelle position
sol = [2, 9, 4, 2, 6, 0, 9, 0xf, 4, 7, 2, 6, 0, 8, 1, \
        0xf, 7, 3, 0xb, 5, 7, 7, 2, 0, 0xc, 5, 5, 4, 0, 0, 3]
trylevel = range(0, len(sol)+1)
start_try = 0x0
end_try = len(try_keys)

resall = []
for tryall in trylevel:
        resloc = []
        for i in range(start_try, end_try):
                fi = "key_gen"
                gen_file(sol[0:tryall], try_keys, fi, try_keys[i])
                print "Trying ",hex(i),
                (outp, code) = exec_chal(fi)
                print code
                if code > tryall*3+14 or code == 0:
                        print "Found:",tryall, "=",  hex(i), "retour;", code
                        resloc.append(hex(i))
        resall.append(resloc)
        cnt = 1
        for i in resall:
                print cnt, i
                cnt += 1
```

lev3.shares

```
VTOb+Y/tmh56nWdszS3z3hfRhel1nIoMWrjm5wjQgOa8gIJhyX6/kUpM9pTxsaOplzM88VkMEWX5De8PfAmrvkZw1FE2R19PAa1TqEkPkxsPEannfNgTmITz
504552ed5171b71887ebabcaaa81a6a1
QAFU+xVpKuse2z7asOSFMtUuOm+EUM85YbVB2TGpqs/0YLXDpnHdMGmUp4D4OuTO3arPNJFOsGwC5o2/QoIVyQX9Tc4MBOtQ9upNkUeDGti8l7d/3tzL5q93
81a9f986471976e41cf7f09c706ba03a
gODjs+MPfbr218SJcsK2izaxpG8OxMgSYkOxulJSY4uHX5c7ulKnUe3eaGPo+9HI1NYEkCa9uO2ZjZl8mila+A==
231b9266eb2f3f55d8f642f6776ab9ff
q6pOBdUAVUQaeEIrEnrbXtw1QSu8jqtbCAj3X6daR/RfthUyxUP6dnfeLQ1F7vdQUGoLlyUCJGu/UGh833GX4g==
4749f03f5af74f0f672e7b41d993959d
```

brute.py

```
from Crypto.Cipher import AES
import base64
import itertools

# Controle le padding du dechiffre
def valid(s):
        c = int(s[-1:].encode("hex"),16)
        i = 1
        while(s[len(s)-i-1] == s[-1:]):
                i += 1
        return c == i

# Dechiffre
def dec(key, iv, dat):
        obj = AES.new(key, AES.MODE_CBC, iv)
        d = obj.decrypt(dat)
        if(valid(d)):
                return d[:-1*int(d[-1:].encode("hex"),16)]
        else:
                return ""
```

```
iv = []
dat = []
# Lit les iv et chiffre du niveau 3
f = open("lev3.shares")
for i in range(4):
        d = base64.b64decode(f.readline().strip())
        dat.append(d)
        iv.append(f.readline().strip().decode("hex"))

# Characteres possibles par position
# Chaine renvoyee par check_strange.py
sol = [
['0x2'],
['0x3', '0x9'],
['0x4'],
['0x2'],
['0x5', '0x6'],
['0x0'],
['0x3', '0x9'],
['0x8'],
['0x4'],
['0x7'],
['0x1', '0x2'],
['0x5', '0x6'],
['0x0'],
['0x8'],
['0x1', '0x2'],
['0x8'],
['0x7'],
['0x3'],
['0x3'],
['0x5', '0x6'],
['0x7'],
['0x7'],
['0x2'],
['0x0'],
['0x8'],
['0x5', '0x6'],
['0x5', '0x6'],
['0x4'],
['0x0', '0x4'],
['0x0'],
['0x3'],
['0x5', '0x6']
]
# Transformation de sol en string hexa
res = []
for e in sol:
        res.append(map(lambda x: str(int(x,16)), e))

cnt = 0
# Bruteforce
for k in itertools.product(*res):
   for i in range(0,len(iv)):
        key = "".join(k)
        t = dec(key.decode("hex"), iv[i], dat[i])
        if( t!="" and t[0] == "[" and t[-1] == "]"):
                print "Found key:", key
                print t
   cnt += 1
print cnt, "keys tested"
```