Challenge SSTIC 2016 : solution

Pierre Zurek

17 mai 2016

Résumé

Ce document présente les étapes que j'ai suivies afin de résoudre le challenge SSTIC 2016.

La validation du challenge nécessite de pouvoir extraire, depuis un fichier téléchargé sur le site de la conférence, une adresse email de la forme `@sstic.org`.

Cette année le challenge est constitué de trois niveaux décomposés en plusieurs épreuves, une épreuve valant un ou deux points. Il faut deux points à chaque niveau pour passer au niveau suivant.

— les fichiers à analyser au premier niveau sont un programme de calculatrice, une trace réseau et une trace GSM
— les fichiers à analyser au deuxième niveau sont un programme ELF de plusieurs To, un binaire UEFI, et un programme Windows
— les fichiers à analyser au troisième niveau sont un programme ELF, deux programmes Windows et un screen-saver Windows

Cette solution présente la résolution de toutes les épreuves ci-dessus à l'exception de l'épreuve dite "ring".

Table des matières

Chapitre 1

Fichier du challenge

1.1 Découverte du RPG

En premier lieu on télécharge le fichier du challenge à l'adresse fournie et on vérifie son intégrité :

```
$ wget --quiet http://static.sstic.org/challenge2016/challenge.pcap
$ sha256sum challenge.pcap
0d39c9c1d09741a06ef8e35c0b63e538f60f8d5a7f995c7764e98a3ec595e46f  challenge.pcap
$ file challenge.pcap
challenge.pcap: tcpdump capture file (little-endian) - version 2.4 (Ethernet, capture length 32767)
```

Le fichier pcap peut être ouvert avec Wireshark et présente la récupération d'un fichier challenge.zip depuis un serveur HTTP. La fonctionnalité Follow TCP Stream de Wireshark permet de sauver la réponse complète du serveur (headers HTTP suivis du fichier zip). En effaçant les headers HTTP de la réponse on obtient bien un fichier zip valide.

```
$ file challenge.zip
challenge.zip: Zip archive data, at least v2.0 to extract
$ unzip challenge.zip
Archive:  challenge.zip
  inflating: rpgjs/core/scene/Scene_Gameover.js
  inflating: Audio/BGM/level_1a_24db_44100_56k_mono.ogg
  inflating: plugins/sham-data.js
   creating: fancybox/
  inflating: plugins/bonus-data.js
   creating: plugins/Sham/
  inflating: rpgjs/core/scene/Scene_Title.js
  inflating: fancybox/jquery.fancybox-1.3.4.css
   creating: plugins/Bonus/
  inflating: wood.png
  inflating: rpgjs/core/scene/Scene_Map.js
  inflating: fancybox/jquery.fancybox-1.3.4.pack.js
  inflating: plugins/Sham/Game_Sham.js
  inflating: rpgjs/rpgjs.min.js
  inflating: fancybox/jquery.js
  inflating: Audio/BGM/level_2a_24db_44100_56k_mono.ogg
  inflating: rpgjs/core/scene/Scene_Load.js
  inflating: index.html
  inflating: Audio/BGM/maison.ogg
  inflating: plugins/Bonus/Game_Bonus.js
  inflating: rpgjs/core/scene/Scene_Menu.js
  inflating: Audio/BGM/level_1b_24db_44100_56k_mono.ogg
  inflating: rpgjs/core/scene/Scene_Generated.js
  inflating: plugins/sham.min.js
  inflating: Audio/BGM/level_2b_24db_44100_56k_mono.ogg
  inflating: plugins/Sham/Sprite_Sham.js
  inflating: rpgjs/canvasengine.min.js
  inflating: plugins/fs-data.js
  inflating: Audio/BGM/final_44100_56k_mono.ogg
  inflating: rpgjs/core/scene/Scene_Window.js
  inflating: plugins/soundmanager2-jsmin.js
  inflating: Audio/BGM/level_3_24db_44100_56k_mono.ogg
```

```
inflating: plugins/Bonus/Sprite_Bonus.js
```

On peut alors ouvrir le fichier index.html dans un navigateur Web et l'on obtient un RPG en javascript.

En discutant avec différents personnages du RPG, on récupère trois fichiers .zip : calc.zip, SOS-Fant0me.zip et radio.zip. Chacun de ces fichiers correspond à une épreuve. Elles seront étudiées au chapitre suivant.

1.2 Analyse rapide du javascript du RPG

1.2.1 Données

Le fichier sham-data.js contient les données du niveau, on constate les données suivantes :

— l'objet JSON next_level avec les clés "iv" et "data"
— threshold qui vaut 2
— l'objet JSON shares :

```
copy(ssmdata[0].shares)

{
  "c6423e4560a9063b095159ccdd0024a10c004d6bde08476484b2d923cff26cde" : {
    "data" : "NjXhESoqCRi/YZa3DhAN9IY58gUxNuDJXu2JRWnBhfhwYxjC8CvaZqwjptvm/uhexdUigFEHo2NPpEqiM7Pm6A==",
    "iv" : "3a981974530c706ac975e90daaadec46"
  },
  "8dc78d6c32608e3f1d251cff09b33e3d61b94dcedb3736c0c829f91f2568b07d" : {
    "data" : "Gi9I0aO5mVnmGUWwisyBbP029xjHwqyaZG0GzQ+T2Jga80OeOkWiM0NpIBT3S0ICwFEoAosdlf4pZUQzcRfU8HdBR5iqTZxD1HOI7AkKw
↪   XJIfLmA+DYRMV/Ba7wRLKKI",
    "iv" : "a4098f85085ebd9f6a1681b7cabc5b23"
  },
  "e75d33d256611799a66722a528b9ee77793487376515d40f7d376d3cf0c04ed2" : {
    "data" : "MOpkde32auDRQqAhXOum88iUs1lrqU9bdbc4jddCSu2pT3XaON9EUsAZ9BBlz6Nte/2U2e+68Jj/LEJWJYbLHw==",
```

```
    "iv" : "1671d25f1d044417674514684098a9c1"
  }
}
```

On remarque que les sections ”data” sont encodées en Base64.

L’étude des fichiers sham.min.js et Sprite_Sham.js permet de comprendre que chacune des clés de l’objet shares est le SHA256 de la clé d’une épreuve, cette clé d’épreuve servant à déchiffrer la sous-section data avec l’algorithme aes-128-cbc et l’iv fourni.

1.2.2 Exemple en utilisant la clé de calc.zip

Calcul du SHA256

```
K=57d9f82b49c1eb3993cb82d26e37f69c
echo -n "$K" | xxd -r -p | sha256sum
e75d33d256611799a66722a528b9ee77793487376515d40f7d376d3cf0c04ed2  -
```

On retrouve bien ce SHA256 dans la section shares ci-dessus. Aussi, on peut utiliser aes-128-cbc et la clé pour déchiffrer la partie data correspondante.

AES

```
$ K="57d9f82b49c1eb3993cb82d26e37f69c"
$ iv="1671d25f1d044417674514684098a9c1"
$ B64="MOpkde32auDRQqAhXOum88iUs1lrqU9bdbc4jddCSu2pT3XaON9EUsAZ9BBlz6Nte/2U2e+68Jj/LEJWJYbLHw=="
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :4,"y" :"7d59b7dc2a7937eba782af4e41b33477"}]
```

On comprend que le résultat obtenu permettra (combiné avec le résultat de l’épreuve S0S-Fantome.zip) de déchiffrer la sous-section ”data” de la section next_level.

Le fait de connaître les SHA256 des clés permet de vérifier très rapidement qu’une clé obtenue est correcte et aura une utilité pratique pour l’épreuve strange.

Chapitre 2

Niveau 1

2.1 Données JS

Pour rappel les données JS du niveau sont :

```
copy(ssmdata[0].shares)

{
  "c6423e4560a9063b095159ccdd0024a10c004d6bde08476484b2d923cff26cde" : {
    "data" : "NjXhESoqCRi/YZa3DhAN9IY58gUxNuDJXu2JRWnBhfhwYxjC8CvaZqwjptvm/uhexdUigFEHo2NPpEqiM7Pm6A==",
    "iv" : "3a981974530c706ac975e90daaadec46"
  },
  "8dc78d6c32608e3f1d251cff09b33e3d61b94dcedb3736c0c829f91f2568b07d" : {
    "data" : "Gi9I0aO5mVnmGUWwisyBbP029xjHwqyaZG0GzQ+T2Jga80OeOkWiM0NpIBT3S0ICwFEoAosdlf4pZUQzcRfU8HdBR5iqTZxD1HOI7AkKwXJIfLmA
↪  +DYRMV/Ba7wRLKKI",
    "iv" : "a4098f85085ebd9f6a1681b7cabc5b23"
  },
  "e75d33d256611799a66722a528b9ee77793487376515d40f7d376d3cf0c04ed2" : {
    "data" : "MOpkde32auDRQqAhXOum88iUs1lrqU9bdbc4jddCSu2pT3XaON9EUsAZ9BBlz6Nte/2U2e+68Jj/LEJWJYbLHw==",
    "iv" : "1671d25f1d044417674514684098a9c1"
  }
}
```

2.2 calc.zip

On commence par extraire le fichier :

```
$ unzip calc.zip
Archive:  calc.zip
  inflating: SSTIC16.8xp
$ file SSTIC16.8xp
SSTIC16.8xp: TI-83+ Graphing Calculator (program)
```

On a affaire à un programme TI-83+. On installe l'émulateur tilem pour pouvoir l'exécuter.

```
$ tilem2 SSTIC16.8xp
```

Le programme SSTIC16 nous demande de saisir un code, et si l'on saisit un code au hasard il affiche "PERDU".

2.2.1 Analyse du code décompilé

Le programme tibasic-1.4.2 téléchargeable sur https://sourceforge.net/projects/tibasic/ permet de décompiler des fichiers .8xp.

On le lance sur notre fichier :

```
$ tibasic -d SSTIC16.8xp
```

On obtient un fichier texte SSTIC16.tib tout à fait lisible.

Dans ce fichier on voit apparaître une liste de constantes commençant par :

```
"0,1996959894,3993919788,2567524794..."
```

Une recherche de ces constantes sur Internet permet de comprendre qu'on a affaire à un calcul de CRC32. Le programme calcule le CRC32 du nombre saisi, et si ce CRC32 est égal à 3298472535, il affichera la clé, qui sera obtenue par appels successifs de la fonction rand.

2.2.2 Bruteforce du CRC32

On récupère une implémentation de CRC32 à l'adresse suivante : http://opensource.apple.com//source/xnu/xnu-1456.1.26/bsd/libkern/crc32.c?txt

On applique le patch suivant pour effectuer le bruteforce :

```
$ diff crc32.c crc32.c.orig
43,45c43,44
< #include <stdint.h>
< #include <stdlib.h>
< #include <stdio.h>
---
> #include <sys/param.h>
> #include <sys/systm.h>
105,116d103
< }
<
< int main(void)
< {
<     for (unsigned int j = 0xffffffff; j-- > 0; )
<     {
<         uint32_t crc = crc32(0, &j, 4);
<         if (crc == 3298472535)
<             printf("found for j=%u\n", j);
<     }
<
<     return 0;
```

On exécute :

```
$ gcc -o bruteforce crc32.c
$ ./bruteforce
found for j=89594902
```

Ne reste qu'à saisir le nombre 89594902 dans le programme SSTIC16 pour récupérer la clé, qui est : 57D9F82B49C1EB3993CB82D26E37F69C.

On vérifie que le SHA256 de la clé se trouve bien dans les données du RPG :

```
$ K="57d9f82b49c1eb3993cb82d26e37f69c"
$ echo -n "$K" | xxd -r -p | sha256sum
e75d33d256611799a66722a528b9ee77793487376515d40f7d376d3cf0c04ed2  -
$ iv="1671d25f1d0444176745146840 98a9c1"
$ B64="MOpkde32auDRQqAhXOum88iUs1lrqU9bdbc4jddCSu2pT3XaON9EUsAZ9BBlz6Nte/2U2e+68Jj/LEJWJYbLHw=="
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :4,"y" :"7d59b7dc2a7937eba782af4e41b33477"}]
```

2.3 SOS-Fant0me.zip

On extrait l'archive :

```
$ unzip SOS-Fant0me.zip
Archive:  SOS-Fant0me.zip
  inflating: SOS-Fant0me.pcap
```

On ouvre le fichier SOS-Fant0me.pcap dans Wireshark qui fait apparaître une connexion TCP. On voit apparaître le mot "Gh0st" dans les contenus. Une recherche Internet nous mène à https://en.wikipedia.org/wiki/Gh0st_RAT et http://www.mcafee.com/us/resources/white-papers/foundstone/wp-know-your-digital-enemy.pdf.

Ce pdf présentant le protocole utilisé, on comprend que le serveur envoie des requêtes, et que le client exfiltre des données.

Un fichier C implémentant le protocole Gh0st permet d'extraire les contenus de cette communication.

On aperçoit notamment les messages suivants :

```
[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
```

On peut alors utiliser ce mot de passe sur le fichier zip issu de l'extraction :

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

On vérifie que cette clé a bien le SHA256 attendu :

```
$ K="368be8c1cc7cc70c2245030934301c15"
$ echo -n "$K" | xxd -r -p | sha256sum
c6423e4560a9063b095159ccdd0024a10c004d6bde08476484b2d923cff26cde  -
$ iv="3a981974530c706ac975e90daaadec46"
$ B64="NjXhESoqCRi/YZa3DhAN9IY58gUxNuDJXu2JRWnBhfhwYxjC8CvaZqwjptvm/uhexdUigFEHo2NPpEqiM7Pm6A=="
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :1,"y" :"2012e892d20635a3c1205d37321bc68a"}]
```

2.4 radio.zip

On décompresse l'archive :

```
$ unzip radio.zip
Archive:  radio.zip
  inflating: rtl2832-f9.452000e+08-s1.000000e+06.bin.lzma
$ lzma -d rtl2832-f9.452000e+08-s1.000000e+06.bin.lzma
$ file rtl2832-f9.452000e+08-s1.000000e+06.bin
rtl2832-f9.452000e+08-s1.000000e+06.bin: International EBCDIC text, with very long lines, with no line terminators
```

Internet nous indique que nous avons affaire à une trace RTLSDR et qu'il faut la convertir en un fichier cfile pour pouvoir l'utiliser. Cela se fait en utilisant gnuradio-companion. Malheureusement l'API a changé et le fichier .grc disponible sur Internet qui permettait d'effectuer la conversion ne fonctionne plus.

Heureusement le post stackoverflow suivant fournit un programme C qui convertit le .bin : http://stackoverflow.com/questions/25587959/bin-to-cfile-flowgraph-for-grc-3-7-2-1.

On le compile et on l'exécute :

```
$ gcc -o rtlsdr-to-gqrx rtlsdr-to-gqrx.c
$ ./rtlsdr-to-gqrx rtl2832-f9.452000e+08-s1.000000e+06.bin capture.cfile
```

Il faut ensuite lancer Wireshark avec la commande suivante :

```
$ sudo wireshark -i lo -k -Y gsmtap
```

Il suffit alors de lancer la commande suivante pour visualiser le contenu de la trace dans Wireshark :

```
$ airprobe_decode.py -c capture.cfile -s 1000000 -f 945200000 -t 0 -m BCCH_SDCCH4
```

En cliquant dans Wireshark sur le message dont le protocole est GSM SMS, puis sur GSM SMS TPDU et TP-User-Data, on voit apparaître le message suivant :

```
Bonjour, votre cle est 1ac3d8c409e656380a06f6f2c6de6b4a
```

On vérifie le SHA256 :

```
$ K="1ac3d8c409e656380a06f6f2c6de6b4a"
$ echo -n "$K" | xxd -r -p | sha256sum
8dc78d6c32608e3f1d251cff09b33e3d61b94dcedb3736c0c829f91f2568b07d  -
$ iv="a4098f85085ebd9f6a1681b7cabc5b23"
$
↪  B64="Gi9I0aO5mVnmGUWwisyBbP029xjHwqyaZG0GzQ+T2Jga80OeOkWiM0NpIBT3S0ICwFEoAosdlf4pZUQzcRfU8HdBR5iqTZxD1HOI7AkKwXJIfLmA+DYRMV/Ba7wRLKKI"
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :2,"y" :"eb2bdda885d334641cbe0ce01c839756"},{"x" :3,"y" :"adc33141489fcb26a8343c52f90ba7e0"}]
```

Chapitre 3

Niveau 2

3.1 Données JS

Les nouvelles données shares de ce niveau sont :

```
copy(ssmdata[1].shares)


{
  "3aaa4de2fc1f067877ef5219dd3af6a5301ed0573fc6ce856107135fe81d0c3d" : {
    "data" : "zYER1E8jhumV80LfDgI0EnyB9JBaDTFgUQgmpN2dvDrgFKYIKiREpKEOwXyfsKeRsFRQc6kJUIm1vQgl6KQP6g==",
    "iv" : "571f1ed126d2c51efae9c458811d6351"
  },
  "0c193b5fb9234e538d8dc869600dc58503e6a6884595827b97ca600dd1e45213" : {
    "data" : "v1xX7AtwBv32xUCcCKlpppj+k9afyNJ+H56EEZ/SMIhfZO7XEerKdSfi6HwdhCWmcn7arW+Pf8j4igKFqgrARA==",
    "iv" : "6f95c94d9cf685a20d218655b05357e6"
  },
  "d28f73a4e9c48a2c55c74a6b5d66c5e5a4bea73a73d273397c6055843f9de5b2" : {
    "data" : "RtCCbuMnsgj7EikAMmot1EnKJHwLZLWfcCcoPWvsaUVN/WGRCLTuoDe5/5qJi6ELd3/ptK0aH8km+VbvaDRqCA==",
    "iv" : "9b412f3cf43047cfa58bd6c9c80aaad4"
  }
}
```

3.2 foo.zip

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

radare2 permet d'étudier le code désassemblé de ce binaire. Afin d'exécuter et débugger ce programme, j'ai utilisé EmulatorPkg disponible dans https://github.com/tianocore/edk2 en le modifiant légèrement afin de pouvoir exécuter le programme foo.efi et avoir une interface gdb.

On comprend que le programme décompresse la chaîne secrète "cb41dcb1d89746705a7fe998f11acce7" et la compare avec la chaîne donnée en entrée du programme. Le programme C suivant permet d'obtenir la clé à partir de la chaîne secrète :

```c
#include <stdio.h>

unsigned char secret_data[16] = {
    0xcb, 0x41, 0xdc, 0xb1, 0xd8, 0x97, 0x46, 0x70, 0x5a, 0x7f, 0xe9, 0x98, 0xf1, 0x1a, 0xcc, 0xe7
};
```

```c
int main(void)
{
    int i;
    for (i = 0; i < 16; i ++)
    {
        int imod = i % 8;
        unsigned int e = secret_data[i];
        unsigned int not_e = ~e;
        not_e = not_e & 0xff;
        unsigned char res = ((not_e << imod) & 0xff) | ((not_e >> (8 - imod)) & 0xff);
        printf("%02hhx", res);
    }
    printf("\n");
}
```

Son exécution donne la clé suivante : 347d8c72720d6ec7a501583be0bccc0c.

On vérifie son SHA256 :

```
$ K="347d8c72720d6ec7a501583be0bccc0c"
$ echo -n "$K" | xxd -r -p | sha256sum
3aaa4de2fc1f067877ef5219dd3af6a5301ed0573fc6ce856107135fe81d0c3d  -
$ iv="571f1ed126d2c51efae9c458811d6351"
$ B64="zYER1E8jhumV80LfDgI0EnyB9JBaDTFgUQgmpN2dvDrgFKYIKiREpKEOwXyfsKeRsFRQc6kJUIm1vQgl6KQP6g=="
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :1,"y" :"028a3e484c106c9e2c660b1371227039"}]
```

3.3 huge.zip

On extrait l'archive :

```
$ unzip huge.zip
Archive:  huge.zip
  inflating: huge.tar
$ file huge.tar
huge.tar: POSIX tar archive
```

Le fichier huge.tar contient un fichier Huge de 117 To. Pour pouvoir l'extraire il faut un système de fichiers supportant des fichiers de cette taille, par exemple XFS. Les étapes suivantes montrent l'extraction et l'exécution du programme :

```
$ dd if=/dev/zero of=image.xfs count=200000
200000+0 records in
200000+0 records out
102400000 bytes (102 MB) copied, 0.188993 s, 542 MB/s
$ mkfs.xfs image.xfs
meta-data=image.xfs              isize=256    agcount=4, agsize=6250 blks
         =                       sectsz=512   attr=2, projid32bit=1
         =                       crc=0        finobt=0
data     =                       bsize=4096   blocks=25000, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0 ftype=0
log      =internal log           bsize=4096   blocks=853, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
$ mkdir /tmp/xfs
$ sudo mount -o loop image.xfs /tmp/xfs
$ sudo mkdir /tmp/xfs/extract
$ sudo chmod 777 /tmp/xfs/extract
```

```
$ tar xv -C /tmp/xfs/extract -f huge.tar
Huge
$ /tmp/xfs/extract/Huge
Please enter the password:
```

Dans le fichier tar, la sparse map indique à quels offsets seront placés les zones extraites. Elle peut être extraite

avec la commande suivante :

```
$ dd if=huge.tar of=sparse_map skip=$((0x600)) bs=1 count=$((0x1eb))
```

Son format est : nombre d'éléments (25 ici), premier offset, taille de la première section, deuxième offset, taille de la deuxième section, etc... Son contenu est le suivant :

Offset	Taille
0x0	4096
0x17affef1000	4096
0x17affef7000	4096
0xa2845ab1000	4096
0xa2845ab4000	4096
0x17021da91000	4096
0x17021da9d000	4096
0x18abdb4a1000	4096
0x1ee7e5411000	4096
0x1ee7e541c000	8192
0x2b2706801000	4096
0x2b270680f000	4096
0x32566a40f000	4096
0x3adb440a1000	4096
0x3adb440a5000	4096
0x3b5815631000	4096
0x50e4b0dc1000	4096
0x50e4b0dd0000	4096
0x538a38011000	4096
0x538a38018000	4096
0x5ae338cb1000	4096
0x5ae338cb8000	4096
0x746ee2461000	4096
0x746ee246b000	4096
0x74effffc0000	4096

Le header ELF donne quant à lui les informations suivantes :

```
$ readelf -l /tmp/xfs/extract/Huge

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
```

En mixant les informations de la sparse map avec les informations du header ELF, on obtient la map suivante des données de code du programme :

Offset
0x06f4b0e00000
0x06f4b0e0f000
0x099a38050000
0x099a38057000
0x10f338cf0000
0x10f338cf7000
0x2a7ee24a0000
0x2a7ee24aa000
0x2affffff000
0x2c7affef0000
0x2c7affef6000
0x352845ab0000
0x352845ab3000
0x42021da90000
0x42021da9c000
0x43abdb4a0000
0x49e7e5410000
0x49e7e541b000
0x4a1706820000
0x4a170682e000
0x51466a42e000
0x59cb440c0000
0x59cb440c4000
0x5a4815650000

Cette map rend beaucoup plus facile l'analyse du code. Après analyse, on comprend que les différentes parties effectuent des comparaisons avec le mot de passe saisi pour continuer ou au contraire arrêter l'exécution. Chaque comparaison permet de connaître une partie du mot de passe à saisir, qui est : 29897ed1d4d68c99d54ec08c89341bec.

```
$ /tmp/xfs/extract/Huge
Please enter the password: 29897ed1d4d68c99d54ec08c89341bec
The key is: E574B514667F6AB2D83047BB871A54F5
```

On vérifie le SHA256 :

```
$ K="e574b514667f6ab2d83047bb871a54f5"
$ echo -n "$K" | xxd -r -p | sha256sum
0c193b5fb9234e538d8dc869600dc58503e6a6884595827b97ca600dd1e45213  -
$ iv="6f95c94d9cf685a20d218655b05357e6"
$ B64="v1xX7AtwBv32xUCcCKlpppj+k9afyNJ+H56EEZ/SMIhfZO7XEerKdSfi6HwdhCWmcn7arW+Pf8j4igKFqgrARA=="
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :2,"y" :"49bc2121192db7f888a2e82e977dc5fb"}]
```

3.4 loader.zip

On extrait l'archive :

```
$ unzip loader.zip
Archive:  loader.zip
  inflating: loader.exe
$ file loader.exe
loader.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

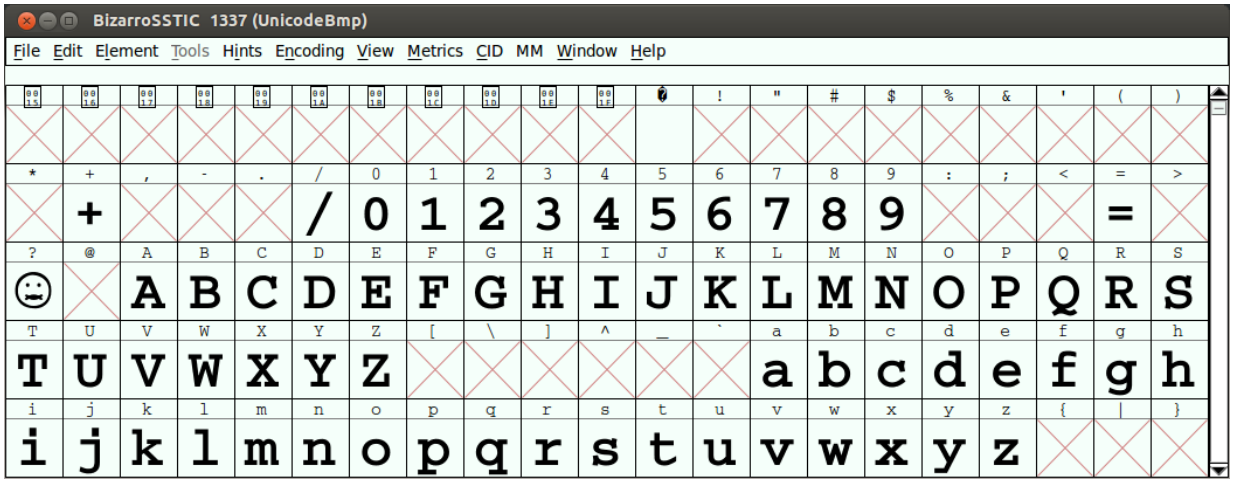
L'analyse du code désassemblé nous montre que le binaire charge la ressource 1337, et lui "envoie" le texte saisi par l'utilisateur. On comprend qu'il faut donc analyser cette ressource.

Tout d'abord on extrait cette ressource :

```
$ wrestool -x -R -n 1337 -o 1337 loader.exe
$ file 1337
1337: TrueType font data
```

On comprend qu'on a affaire à une fonte TrueType. Le logiciel fontforge permet d'ouvrir cette fonte.

En ouvrant fontforge on comprend mieux :



Le caractère ' ?' a été remplacé par un smiley pouvant être heureux ou triste. On comprend qu'il faut faire apparaître le visage heureux du smiley.

On vérifie en saisissant un mot au hasard :

3.4.1 Analyse du bytecode des caractères

Dans fontforge on peut lire les instructions de chacun des caractère de la fonte.

Soit r l'ensemble des registres, soit f la fonction suivante :

$$f(X) = \{r_{r_{99}} = X; r_{99} \leftarrow r_{99} + 1\}$$

La majorité des caractères de la fonte utilise la fonction f avec un paramètre X différent. Le tableau ci-dessous regroupe les valeurs :

Caractère	Fonction
'='	$f(0)$
'a'	$f(0)$
'b'	$f(1)$
...	...
'y'	$f(24)$
'z'	$f(25)$
'A'	$f(26)$
'B'	$f(27)$
...	...
'Y'	$f(50)$
'Z'	$f(51)$
'0'	$f(52)$
'1'	$f(53)$
...	...
'8'	$f(60)$
'9'	$f(61)$
'+'	$f(62)$
'/'	$f(63)$

Quant au smiley, j'ai écrit un script python interprêtant ses instructions et stockant le contenu de la pile en fonction des valeurs des registres au départ. Une instruction IF affichera ou non le sourire en fonction de la valeur au sommet de la pile. Mon script me donne comme valeur au sommet de la pile la formule suivante :

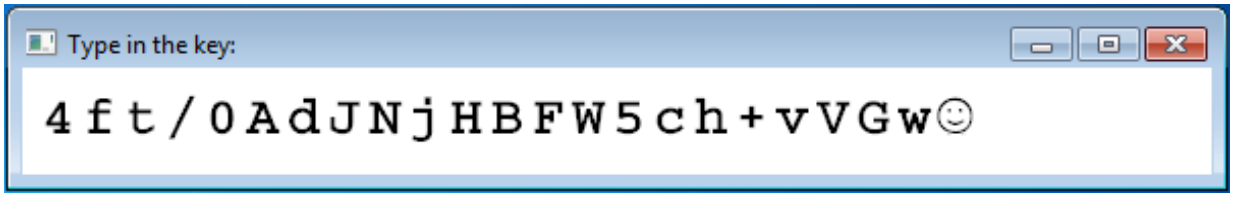
```
r99 && (19 == r2) && (186 == (r12 * 6)) && (63 == r3) && (39 == r8) && (35 == r7) && (263 == (((r13 +
↪  6) * 5) - 7)) && (26 == r5) && (28 == (r21 + 6)) && (21 == r18) && (53 == (r0 - 3)) && (21 == (r11
↪  - 6)) && (582 == (((r19 * 4) + 6) * 3)) && (20 == (r1 * 4)) && (2 == r15) && (1463 == (((r4 * 4) +
↪  1) * 7)) && (1415 == (((r14 * 5) - 2) * 5)) && (70 == (((r17 + 2) + 4) + 2)) && (32 == (r10 - 1))
↪  && (5 == (r16 - 2)) && (90 == ((((r9 + 2) + 7) * 1) * 5)) && (3 == ((((((r6 + 5) - 1) - 3) + 5) -
↪  6) * 1)) && (1970 == (((((((r20 - 7) - 1) * 4) + 3) * 4) - 2) * 5))
```

Cette formule se traduit par le système d'équations suivant :

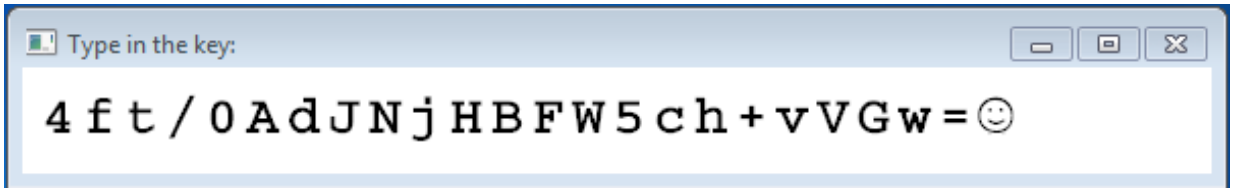
$$\begin{cases} r_0 = 56 \\ r_1 = 5 \\ r_2 = 19 \\ r_3 = 63 \\ r_4 = 52 \\ r_5 = 26 \\ r_6 = 3 \\ r_7 = 35 \\ r_8 = 39 \\ r_9 = 9 \\ r_{10} = 33 \\ r_{11} = 27 \\ r_{12} = 31 \\ r_{13} = 48 \\ r_{14} = 57 \\ r_{15} = 2 \\ r_{16} = 7 \\ r_{17} = 62 \\ r_{18} = 21 \\ r_{19} = 47 \\ r_{20} = 32 \\ r_{21} = 22 \\ r_{99} \neq 0 \end{cases}$$

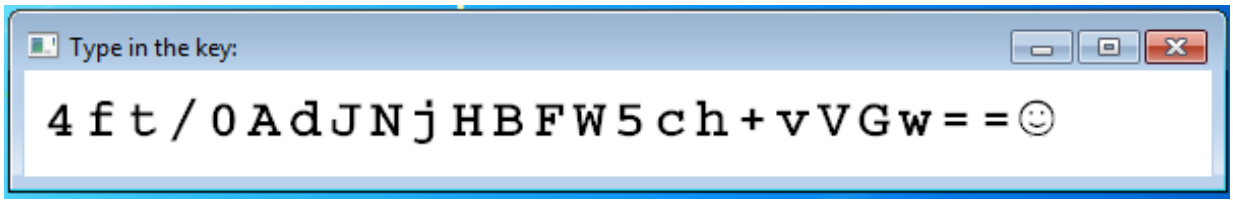
En partant de l'hypothèse que r_{99} vaut 0 au départ, on en déduit qu'il faut saisir la valeur suivante : "4ft/0AdJNjHBFW5ch+vVGw ?" pour faire apparaître le smiley souriant.

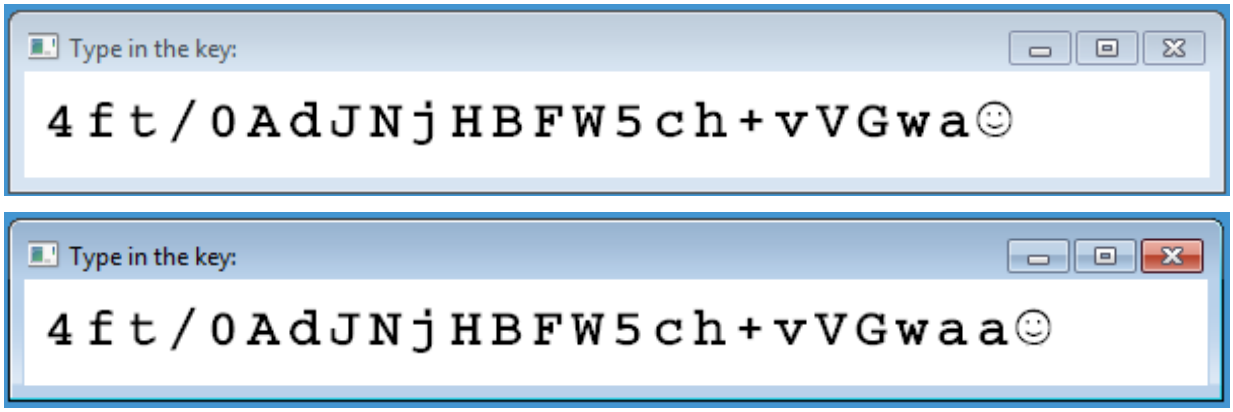
On vérifie :



Note : "4ft/0AdJNjHBFW5ch+vVGw= ?", "4ft/0AdJNjHBFW5ch+vVGw== ?", "4ft/0AdJNjHBFW5ch+vVGwa ?" et ""4ft/0AdJNjHBFW5ch+vVGwaa ?" fonctionnent également.







3.4.2 Clé

On remarque que ”4ft/0AdJNjHBFW5ch+vVGw==” est une chaîne Base 64. On peut donc la décoder pour récupérer la clé :

```
$ echo -n '4ft/0AdJNjHBFW5ch+vVGw=='|base64 -d|xxd -p
e1fb7fd007493631c1156e5c87ebd51b
```

On effectue les vérifications habituelles :

```
$ K="e1fb7fd007493631c1156e5c87ebd51b"
$ echo -n "$K" | xxd -r -p | sha256sum
d28f73a4e9c48a2c55c74a6b5d66c5e5a4bea73a73d273397c6055843f9de5b2  -
$ iv="9b412f3cf43047cfa58bd6c9c80aaad4"
$ B64="RtCCbuMnsgj7EikAMmot1EnKJHwLZLWfcCcoPWvsaUVN/WGRCLTuoDe5/5qJi6ELd3/ptK0aH8km+VbvaDRqCA=="
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :3,"y" :"8f51d4062a39012514e1493a3548a93a"}]
```

Chapitre 4

Niveau 3

4.1 Données JS

Les nouvelles données shares de ce niveau sont :

```
copy(ssmdata[2].shares)

{
  "b3cc615b555a12460fc0acbb525f0801bba30f331321a9fdef2f740f3f4146d8" : {
    "data" : "VT0b+Y/tmh56nWdszS3z3hfRhel1nIoMWrjm5wjQg0a8gIJhyX6/kUpM9pTxsaOplzM88VkMEWX5De8PfAmrvkZw1FE2R19PAa1TqEkPkxsPEann
↪   fNgTmITzSNCWVaxk",
    "iv" : "504552ed5171b71887ebabcaaa81a6a1"
  },
  "06d44e5be842550344bf740a942ec86542d81c7636f7701662f250afd784d2d6" : {
    "data" : "QAFU+xVpKuse2z7as0SFMtUuOm+EUM85YbVB2TGpqs/0YLXDpnHdMGmUp4D4OuT03arPNJF0sGwC5o2/QoIVyQX9Tc4MBOtQ9upNkUeDGti8l7d/
↪   3tzL5q93EjZPGiS5",
    "iv" : "81a9f986471976e41cf7f09c706ba03a"
  },
  "210584eff5be27c674c37210ee0ba81a41d0996974bf7b8454d9f1dd37963275" : {
    "data" : "g0Djs+MPfbr218SJcsK2izaxpG8OxMgSYk0xulJSY4uHX5c7ulKnUe3eaGPo+9HI1NYEkCa9uO2ZjZl8mila+A==",
    "iv" : "231b9266eb2f3f55d8f642f6776ab9ff"
  },
  "91243d8d4adf63ab54c3bb6a4ed33123496bb9371a8dbf2376bb0037ec869e18" : {
    "data" : "q6pOBdUAVUQaeEIrEnrbXtw1QSu8jqtbCAj3X6daR/RfthUyxUP6dnfeLQ1F7vdQUGoLlyUCJGu/UGh833GX4g==",
    "iv" : "4749f03f5af74f0f672e7b41d993959d"
  }
}
```

4.2 usb.zip

Tout d'abord on extrait l'archive :

```
$ unzip usb.zip
Archive:  usb.zip
  inflating: img.bz2
  inflating: userSSTIC.bin
$ file userSSTIC.bin
userSSTIC.bin: PE32+ executable (GUI) x86-64, for MS Windows
$ bunzip2 img.bz2
$ file img
img: DOS/MBR boot sector; partition 1 : ID=0xb, start-CHS (0x0,0,4), end-CHS (0x88,233,5), startsector 3, 2097152 sectors;
↪   partition 2 : ID=0xb, start-CHS (0x89,19,6), end-CHS (0xcd,135,36), startsector 2099201, 1048575 sectors; partition 3 :
↪   ID=0xb, start-CHS (0xcd,135,38), end-CHS (0xce,218,57), startsector 3147777, 20480 sectors; partition 4 : ID=0xb, start-CHS
↪   (0xce,218,58), end-CHS (0x14,93,50), startsector 3168257, 819199 sectors
```

4.2.1 userSSTIC.bin

L'analyse du programme userSSTIC.bin dans IDA nous montre que le programme charge un driver drvSSTIC.sys et écoute l'arrivée de nouveaux volumes et transmet la lettre du volume au driver.

4.2.2 drvSSTIC.sys

Le driver stocke des fichiers qui se trouvaient dans C :\SSTIC dans les espaces inter-partitions du disque usb. Il utilise deux algorithmes de chiffrement :

- une version modifiée de RC6 (avec une initialisation de la clé différente et l'échange des opérations ROR et ROL) en CBC, avec la clé "551C2016B00B5F00" et l'IV 0
- l'algorithme RC4 dont la clé est stockée chiffrée par RC6 sur le disque

4.2.3 Extraction

Un simple programme C extrait les différents fichiers stockés dans les espaces inter-partitions. Le premier de ces fichiers contient le texte :

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

On peut donc extraire le fichier zip :

```
$ unzip -P '!WooYouAreSuchAnAwesomeGuy!' mapped_329027.zip
Archive:  mapped_329027.zip
  inflating: 4.jpg
 extracting: key
$ xxd -p key
0928bde1e3ed89698632dbff4a231138
```

On vérifie la clé obtenue :

```
$ K="0928bde1e3ed89698632dbff4a231138"
$ echo -n "$K" | xxd -r -p | sha256sum
210584eff5be27c674c37210ee0ba81a41d0996974bf7b8454d9f1dd37963275  -
$ iv="231b9266eb2f3f55d8f642f6776ab9ff"
$ B64="g0Djs+MPfbr218SJcsK2izaxpG8OxMgSYk0xulJSY4uHX5c7ulKnUe3eaGPo+9HI1NYEkCa9uO2ZjZl8mila+A=="
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :5,"y" :"ca294f1109a92c47ff240853de59625c"}]
```

4.3 video.zip

On extrait l'archive :

```
$ unzip video.zip
Archive:  video.zip
  inflating: Stage_anti_APT_chez_Airlhes/Airlhes_screensaver_setup.exe
  inflating: Stage_anti_APT_chez_Airlhes/Airlhes_CYBER_SECRET_possible_exfiltration.mp4
  inflating: Stage_anti_APT_chez_Airlhes/mission.txt
```

L'exécutable Airlhes_screensaver_setup.exe installe le fichier C :\windows\syswow64\Airlhes_screensaver.scr.

```
$ file Airlhes_screensaver.scr
Airlhes_screensaver.scr: PE32 executable (GUI) Intel 80386 (stripped to external PDB), for MS Windows, UPX compressed
```

On peut utiliser l'utilitaire upx pour décompresser le fichier :

```
$ upx -d Airlhes_screensaver.scr
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91        Markus Oberhumer, Laszlo Molnar & John Reiser   Sep 30th 2013

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     75776 <-     54272   71.62%    win32/pe     Airlhes_screensaver.scr

Unpacked 1 file.
```

Une fois décompressé on peut étudier le code désassemblé dans IDA.

On comprend que la vidéo va chercher la valeur du registre :
HKEY_CURRENT_USER\Software\Airlhes\Screensaver\config\trajectories\spot0
et quand la fonction sub_403750() retourne 0, le screensaver affiche à l'écran des couleurs dépendant des valeurs dans la base de registre.

Le triplet de couleurs BLANC-NOIR-BLANC affiché un peu plus longtemps permet de faire la synchronisation. Ensuite chaque triplet de couleurs correspond à un octet secret.

Un programme C décode chaque triplet de couleurs de la vidéo, il m'affiche le résultat suivant :

```
found byte 0x18 for colors TURQUOISE MAUVE TURQUOISE
found byte 0x00 for colors MAUVE BLEU MAUVE
found byte 0x00 for colors ROUGE BLEU NOIR
found byte 0x00 for colors ROUGE BLEU TURQUOISE
found byte 0x78 for colors BLANC JAUNE BLEU
found byte 0xda for colors ROUGE NOIR MAUVE
found byte 0x0b for colors BLANC VERT JAUNE
found byte 0x16 for colors NOIR ROUGE JAUNE
found byte 0x34 for colors TURQUOISE BLANC NOIR
found byte 0x17 for colors TURQUOISE NOIR VERT
found byte 0x2b for colors BLANC JAUNE BLEU
found byte 0xda for colors JAUNE BLEU MAUVE
found byte 0xc5 for colors BLEU MAUVE NOIR
found byte 0x58 for colors BLANC BLEU TURQUOISE
found byte 0xf9 for colors BLEU VERT BLEU
found byte 0xcb for colors BLANC BLEU BLANC
found byte 0x6e for colors ROUGE BLANC VERT
found byte 0x62 for colors TURQUOISE ROUGE JAUNE
found byte 0x57 for colors TURQUOISE VERT BLEU
found byte 0xf3 for colors ROUGE VERT TURQUOISE
found byte 0xbe for colors ROUGE JAUNE MAUVE
found byte 0x7b for colors BLANC VERT MAUVE
found byte 0x5b for colors TURQUOISE JAUNE BLANC
found byte 0x00 for colors NOIR ROUGE TURQUOISE
found byte 0x32 for colors BLANC NOIR TURQUOISE
found byte 0x50 for colors NOIR BLEU ROUGE
found byte 0x07 for colors BLANC MAUVE NOIR
found byte 0x7e for colors VERT ROUGE MAUVE
```

Soit le message secret "18 00 00 00 78 da 0b 16 34 17 2b da c5 58 f9 cb 6e 62 57 f3 be 7b 5b 00 32 50 07 7e".

Les 4 premiers octets (0x18) donnent le nombre d'octets qui les suivent. La valeur du registre exfiltrée dans la vidéo est donc : 78da0b1634172bdac558f9cb6e6257f3be7b5b003250077e.

78da nous fait penser à un header zlib, on peut obtenir la clé :

```
$ echo -n 78da0b1634172bdac558f9cb6e6257f3be7b5b003250077e|xxd -r -p|zlib-flate -uncompress|xxd -p
5311371672ba0179fa3e918a83bedeb4
```

On effectue les vérifications habituelles :

```
$ K="5311371672ba0179fa3e918a83bedeb4"
$ echo -n "$K" | xxd -r -p | sha256sum
91243d8d4adf63ab54c3bb6a4ed33123496bb9371a8dbf2376bb0037ec869e18  -
$ iv="4749f03f5af74f0f672e7b41d993959d"
$ B64 = "q6pOBdUAVUQaeEIrEnrbXtw1QSu8jqtbCAj3X6daR/RfthUyxUP6dnfeLQ1F7vdQUGoLlyUCJGu/UGh833GX4g=="
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :6,"y" :"7cca9c1ba3da710ad70fb740d8946acb"}]
```

4.4 strange.zip ou le retour de Jean-Michel Bruteforce

On extrait l'archive :

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

Il existe un émulateur IA-64 : ski, téléchargeable sur https://sourceforge.net/projects/ski/. L'analyse du code désassemblé nous apprend que le binaire prend en entrée une image au format PGM ASCII (https://en.wikipedia.org/wiki/Netpbm_format). Cette image a pour format 640x20 (640 pixels de large, 20 pixels de haut).

On remarque trois fonctions :
— sub_400000000019C700 est la fonction main, qui découpe l'image en 32 images de 20x20 et passe chacune de ces 32 sous-images à sub_400000000019C410 puis sub_400000000019AFC0
— sub_400000000019C410 calcule la bounding box de la sous-image qui lui est passée en argument
— sub_400000000019AFC0 appelle un grand nombre de fonctions, toutes ces fonctions appelant la fonction sigmoïde : $f(x) = \frac{1}{1+e^{-\lambda x}}$

La fonction sigmoïde est utilisée dans les réseaux de neurones. Etant donnés les paramètres d'entrée, on devine qu'on a affaire à un réseau de neurones entraîné à reconnaître les caractères de la clé.

La recherche Internet "neural networks 196" nous mène au pdf suivant : http://cagewebdev.com/wp-content/uploads/2016/01/Neural-Networks-for-Dummies.pdf. Dans ce pdf il est fait mention de la base de données du MNIST des chiffres écrits : http://yann.lecun.com/exdb/mnist/.

4.4.1 Premier bruteforce

Un premier bruteforce utilisant les données du MNIST lance bski (le mode console de ski) avec a.out et une image générée contenant les chiffres que l'on veut tester. a.out affiche des points quand les caractères sont reconnus.

Cette première version a duré quelques heures et a donné l'image suivante :

Les caractères dans cette image sont : 2342503147250121733577201554403 9. On voit que les 1 sont des gros pâtés de pixels, ce que je n'avais pas remarqué au moment de la génération des images. Il faut noter que le "9" final n'a pas été reconnu par a.out, cette image est la dernière image générée lors du bruteforce.

4.4.2 Deuxième bruteforce

Les caractères '0' et '1' ne me semblant pas exacts dans l'image trouvée précédemment, je décide de lancer un deuxième bruteforce, avec pour condition d'arrêt les deux SHA256 non identifiés dans les données JS du RPG.

Cela fait un bruteforce à 10^9 possibilités (quatre caractères '0', quatre caractères '1' et le dernier caractère de l'image).

Le résultat tombe au bout de quelques minutes : 234250384725082873357720855440 35.

On effectue les vérifications habituelles :

```
$ K="234250384725082873357720855440 35"
$ echo -n "$K" | xxd -r -p | sha256sum
06d44e5be842550344bf740a942ec86542d81c7636f7701662f250afd784d2d6  -
$ iv="81a9f986471976e41cf7f09c706ba03a"
$
 ↪  B64="QAFU+xVpKuse2z7as0SFMtUuOm+EUM85YbVB2TGpqs/0YLXDpnHdMGmUp4D4OuT03arPNJF0sGwC5o2/QoIVyQX9Tc4MBOtQ9upNkUeDGti8l7d/3tzL5q93EjZPGiS5"
$ echo $(openssl enc -aes-128-cbc -d -iv "$iv" -in <(echo -n "$B64" | base64 -d) -K "$K")
[{"x" :3,"y" :"a7eee9045d4f96ddaf737675d3c373eb"},{"x" :4,"y" :"a78801e89078188318c29d5d23e265d3"}]
```

Chapitre 5

Adresse mail finale

Une fois passé le dernier garde, un personnage nous remet un fichier final.txt dont le contenu est le suivant :

```
Coucou !

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

Un ROT13 a été appliqué, le script python suivant affiche le message décodé :

```
>>> import codecs
>>> codecs.encode("I01p1 y'4qe3553 z41y : 8Y6d5j9Vy88HUGHfGSKsJvqA@ffgvp.bet", 'rot_13')
"V01c1 l'4dr3553 m41l : 8L6q5w9Il88UHTUsTFXfWidN@sstic.org"
```

Il ne reste qu'à envoyer un mail à l'adresse `8L6q5w9Il88UHTUsTFXfWidN@sstic.org`.

Chapitre 6

Conclusion

Je tiens à remercier les concepteurs du challenge ainsi que l'ensemble du comité d'organisation du SSTIC.

Les épreuves du challenge de cette année étaient variées et intéressantes. Je vais essayer de faire l'épreuve ring.