Une solution du Challenge SSTIC 2016

Jeremy Buet

April 16, 2016

Write-Up SSTIC

1 Introduction

Le challenge se présente initialement sous la forme d'un pcap.

Le pcap retrace une transaction HTTP entre un Firefox version 142 (!) et un Serveur CERN 3.0. Outre les versions de logicielles amusantes, la transaction consiste en une demande d'un zip (challenge.zip) et la reception dudit zip.

En extrayant de wireshark la conversation, l'utilitaire unzip remarque automatiquement le zip situé à un offset et extrait une page HTML avec beaucoup de ressources js, css et images.

En regardant ce que fait la page, on se rend compte que c'est un petit RPG en Javascript dont le but va être d'atteindre le niveau final en résolvant des challenges et en donnant des flags au garde pour passer à la partie suivante.

2 1ere Partie

En visitant, on tombe sur un personnage sur le pont qui nous dit d'aller au Nord-Ouest pour avoir les épreuves. La taverne est une distraction qui nous apporte un peu de distraction et d'amusement.

La première épreuve sur laquelle je tombe est un professeur qui me dit qu'il a caché sa clé dans un instrument qu'il a ramené du Texas. Ça ressemble beaucoup à une référence à Texas Instruments, et vu son allure, on va jouer avec une calculette ...

2.1 TI-83+

En effet, après décompression du zip téléchargé, on tombe sur un fichier 8xp. l'utilitaire `file` le reconnait automatiquement comme étant un "TI-83+ Graphing Calculator (program)"

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

Une simple recherche google nous donne https://www.cemetech.net/sc/ qui a la gentillesse de retransformer le binaire en instructions assembleurs.

Ce programme part d'un tableau (qui ressemble beaucoup à celui de CRC32) pour effectuer plusieurs opérations bits à bits avant de checker le résultat. Étant donné qu'il n'attend qu'un entier de 32 bits en entrée et que la valeur finale attendue est également directement checkée, j'ai recodé les calculs en python et testé successivement les 2^{32} entrées jusqu'à obtenir la bonne sortie. Je trouve ainsi que le nombre à entrer était 89594902.

Sauf que le programme TI-83 ne s'arrete pas là. Si l'entrée est bonne, il seed la RNG avec l'entrée et sort 32 entiers 8-bits au hasard, qu'il affiche en hexa : notre clef.

N'ayant ni calculatrice ni émulateur Ti-83+ sous la main, je parts à la recherche de l'algorithme RNG utilisé dans la TI, et je tombe sur une implémentation C++ sur stack overflow (http://stackoverflow.com/questions/32788122/ti-84-plus-random-number-generator-algorithm). Après modification, je suis capable de seeder cette RNG par le nombre de mon choix et en sortir la clef attendue \o/

2.2 SOS Fantome

Un deuxième pcap !

Celui-ci est moins joli que le premier, même s'il s'agit encore d'une unique transaction TCP entre un serveur et un host. En suivant la conversation TCP, on s'aperçoit que `Gh0st` est souvent répété. Une recherche google plus tard, et l'on s'aperçoit que ça semble correspondre à du traffic d'un Gh0st RAT, un vieil remote access tool.

`Gh0st` communique de la façon suivante : il commence par la chaine `Gh0st` (en tant que magic bytes), puis écrit la taille complete du paquet envoyé, sur 4 octets en little-endian. Puis la taille du payload décompressé, toujours sur 4 octets en little-endian, et enfin ajoute son payload, compressé avec zlib, l'algorithme utilisé par gzip.

La solution s'avère ainsi simple : On va extraire la communication, splitter en fichiers contenant chacun un message Gh0st, retirer l'en-tête de ces fichiers et les décompresser en ajoutant devant une petite chaine machine pour que gzip comprenne qu'il peut effectivement les décompresser :

`\x1f\x8b\x08\x00\x00\x00\x00\x00`

Après, il suffit de faire un file sur chacun des fichiers et de regarder leur contenu pour voir :

- Un message “Illuminati”
- Un Rickroll
- Un message de félicitations, coupé en plein de petits bouts
- Un zip nommé “solution.zip”, protégé par mot de passe
- Le mot de passe du zip, écrit caractère par caractère : Cyb3rSSTIC_2016

Décompresser le zip nous donne la deuxième clef nécessaire pour passer le premier garde, et ainsi passer au deuxième niveau, tout enneigé :)

3 2eme Partie

Seules trois épreuves s'offrent à nous dans une petite taverne, ainsi que quelqu'un voulant nous donner le mot de passe d'une boucherie, qui s'avère être cheval. (Bonjour M. Bruder, comment va le CERT CA ?) Aucune des trois ne semble apporter les 2 points nécessaires.

On voit donc :

- Une épreuve loader.exe, qui semble un minimum protégée au démarrage.
- Une épreuve foo, dont le personnage nous la confiant semble sous-entendre qu'il y a un lien avec EFI
- Une épreuve "indigeste", Huge.tar, qui après quelques amusements, s'avère également être une épreuve de reverse.

Donc pas le choix, il va falloir faire du reverse.

3.1 foo.efi

Il s'agit en effet d'un executable EFI. Les Executables EFI se présentent sous un format PE, et sont compilé en un assembleur propre, dont la seule référence trouvable est la syntaxe complète d'EFI. Au moins on ne risque pas de manquer d'informations ...

Démarrons donc IDA.

IDA reconnait assez bien l'assembleur EFI, et nous montre 4 fonctions en plus de la fonction Start.

3.1.1 sub_10000400

La première (`sub_10000400`) est assez simple, et prend visiblement deux arguments, a et b, et effectue une opération bit à bit sur le premier argument assez simple :

f(a,b) = not[(a << b%8) | (a >> (8-b%8)]

On s'aperçoit assez bien que si a désigne un entier 8 bit, il s'agit simplement de la négation bit-à-bit de a dont les bits auraient subi une rotation de b%8 emplacements. Ainsi, f(f(a,b),8-(b%8)) = a Implémentons cette fonction dans un coin, elle pourra être utile plus tard

3.1.2 sub_100009BC

La troisième (`sub_100009BC`) semble moins simple. Mais la représentation en graphe, bloc par bloc, nous éclaire beaucoup sur son fonctionnement : On reconnait une boucle où le programme teste si un octet représente un nombre,

une lettre entre a et f ou entre A et F inclus, envoyant une erreur si ce n'est pas le cas, convertissant l'octet en la valeur hexadécimale représentée en ASCII.

Une dernière boucle regroupe les octets par paire.

Cette fonction est donc une fonction de parsing, qui convertit des strings représentant 16 octets en ASCII en un tableau comprenant les 16 octets réprésentés. Elle doit très probablement parser une entrée utilisateur.

3.1.3 sub_10000C70

La quatrième semble assez simple, donnant effectivement une entrée en argument à `sub_100009BC`, vérifiant si le parsing a réussi, puis donnant le résultat du parsing à la dernière fonction non reversée, `sub_10000530`.

3.1.4 sub_10000530

Cette fonction a l'air plus compliquée. On remarque immédiatement les structures de parsing vues dans `sub_100009BC`. On s'aperçoit qu'un des arguments, le résultat du parsing de `sub_100009BC` est traité par la fonction `sub_10000400` avant d'être xoré avec le résultat du deuxième parsing, le résultat du XOR étant OR-é dans un accumulateur.

Toute dernière opération : on check si le OR vaut zéro. Si oui, le programme affiche "Succès", sinon, c'est un échec.

Comme un OR de plusieurs conditions ne peut être nul que si toutes les conditions sont nulles, on en conclut que chacun des XOR doit être nul, et donc que l'entrée passée doit parfaitement correspondre au résultat du parsing fait un peu plus tôt.

Intéressons-nous à présent au début de la fonction : En tout premier lieu, une boucle s'execute en `loc_10000570`, qui fait un memcopy de 16 octets situés à l'adresse 0x10001698. Puis cette donnée est XORée à une autre, avant d'être passée en argument à une fonction non référencée. En regardant le message d'erreur, on s'aperçoit qu'il s'agit en réalité d'un appel UEFI à LocateProtocol, Demandant donc de chercher le protocole dont le GUID est le résultat du XOR, soit 0xfe7c11d8a694d4119a3a0090273fc14d.

Ce protocole est le protocole Decompress.
Que sont les deux appels suivants ?
D'abord un à GetInfo, qui permet de chercher les informations sur un message compressé,
puis un à Decompress, qui décompresse effectivement le message compressé.

Ces fonctions sont données l'adresse 0x10001470 où se situe la donnée compressée. En la décompressant avec Python et le module eficompressor, on parvient à décompresser le message, et obtenir le message qui est effectivement parsé juste après :

```
secret data: cb41dcb1d89746705a7fe998f11acce7
```

Donc reprenons : L'argument donné au programme UEFI est converti directement de l'hexadécimal à la chaine représentée, puis transformée octet par

octet par une fonction facilement inversible avant d'être comparée à une chaine qu'on a.

Il reste plus qu'à demander à python d'appliquer `sub_10000400` à l'envers (ou plutôt en changeant juste le second argument) pour obtenir l'entrée censée être valide.

Un test avec Qemu et OVMF confirme la validité de l'entrée, et le garde du second niveau également.

3.2 Huge.tar

Une épreuve amusante : dans le zip se trouve un tar, ce qui est peu commun. Si on décompresse le tar dans notre filesystem standard, ext4, on a une erreur, et un fichier énorme. Si à la place on dé-tar l'archivve dans une btrfs ou dans un tmpfs, tar ne sort pas d'erreur, mais un ELF de 117To, dont l'execution plante sur le serveur que j'utilisais...

Un `objdump -x Huge` nous apprend que l'ELF demande à mapper plus de 150Go en mémoire, ce qui est un peu beaucoup.

En revanche si on regarde le contenu du fichier tar on voit l'ensemble des octets non nuls de Huge.tar. Du coup, si on sait où mapper chaque zone d'instructions dans l'ELF lors de l'execution, on peut reconstruire un Huge beaucoup plus accessible !

Regardons plus attentivement le format sparse et l'utilité de la série de nombres précédent les zones d'instructions : - Le premier nombre est 25, soit la moitié du nombre de lignes qui suivent. - Les lignes suivantes ont le format suivant : Un nombre, suivi d'une ligne contenant 4096 (et dans un cas, 8192), ce premier nombre commençant à 0 et étant de plus en plus grand. - Le Binaire fait 26 "pages" (est en 26 blocs de 4096 octets), le premier bloc ressemblant fortement à celui d'un en-tête ELF.

Les autres nombres sont grands, il y a fort à parier qu'il s'agit de la table de mapping du tar vers le Huge décompressé, ayant le sens suivant:

Place à l'octet 0 les 4096 premiers octets que tu lis, puis en 1627791495168 les 4096 octets suivants, etc...

Si l'on croise cette table avec la table de mapping inscrite dans l'entête ELF, on sait où en mémoire doivent être mappées chacune des "pages" du binaire extrait. Un petit script python nous permet de réécrire l'entête ELF en cumulant les deux tables de mapping.

On a donc un executable qui demande une entrée et ... segfault.

Analysons son fonctionnement dans gdb, et suivons-le étape par étape. Le point d'entrée mène à une zone de code qui demande d'entrer le mot de passe, et fait un appel. Cet appel pose problème car l'adresse indiquée n'est pas mappée dans notre nouveau binaire, mais l'est dans l'ancien. Le fait est qu'en amd64, l'instruction 00 00 veut dire `add %al, (%rax)`, ce qui, dans la plupart des cas, équivaut à un NOP. On peut donc remplacer l'adresse d'appel par l'adresse de la prochaine page mappée, ce qui nous permet d'éviter une grosse partie du NOP sled, et de ne pas segfault.

La première fonction appelée est une fonction qui, comme dans EFI, s'assure que l'entrée fait bien la bonne taille, représente bien 16 octets en hexadécimal et remplace la chaine par les 16 octets représentés.

Après le `ret` qui s'ensuit, vient un jump. Après une correction de l'adresse, On tombe sur

```
0x43abdb4a0aee: nop
0x43abdb4a0aef: movabs $0x43abdb4a0b0b,%rbx
0x43abdb4a0af9: cmpb   $0x29,(%rsp)
0x43abdb4a0afd: jne    0x43abdb4a0b09
0x43abdb4a0aff: movabs $0x4a170682000c,%rbx
0x43abdb4a0b09: jmpq   *%rbx
```

qui vérifie si le premier octet qu'on a rentré (une fois parsé) est 0x29, ce qui nous permet de connaitre le premier octet du mot de passe. Le jump, après correction de l'adresse, nous emmène à ce set d'instructions :

```
0x4a170682ede0: nop
0x4a170682ede1: cmpw   $0xd17e,0x2(%rsp)
0x4a170682ede8: movabs $0x6f4b0e0000c,%rbx
0x4a170682edf2: movabs $0x4a170682ee02,%r14
0x4a170682edfc: cmovne %r14,%rbx
0x4a170682ee00: jmpq   *%rbx
```

Ce qui nous permet d'avoir les octets 3 et 4 du mot de passe.

Plus tard, on obtient le 12e octet (en 0x99a3804f370).

À présent, les parties plus compliquées arrivent. en 0x49e7e541be18 on voit

```
0x49e7e541be18: nop
0x49e7e541be19: push   %rax
0x49e7e541be1a: lea    0x10b(%rsp),%rax
0x49e7e541be22: movzbq 0x9(%rsp),%rbx
0x49e7e541be28: movb   $0x0,(%rax)
0x49e7e541be2b: lea    0x6(%rip),%rcx        # 0x49e7e541be38
0x49e7e541be32: lea    (%rcx,%rbx,2),%rcx
0x49e7e541be36: jmpq   *%rcx
```

AL change à cet endroit et donc l'octet où le jump suivant se fait, dépend de l'octet 10 du mot de passe, et influer sur la valeur située à l'adresse pointée par rax à la fin.

En effet, le saut se fera à 0x49e7e541be38+2*, et va incrémenter de 3 le compteur situé à l'addresse 0x7fffffffcd03

à la fin le dernier octet à cette valeur doit valoir 0x65

```
0x49e7e541c038: cmpb   $0x65,(%rax)
0x49e7e541c03b: movabs $0x352845ab3a0c,%rbx
0x49e7e541c045: movabs $0x49e7e541c056,%r14
```

```
0x49e7e541c04f: cmovne %r14,%rbx
0x49e7e541c053: pop    %rax
0x49e7e541c054: jmpq   *%rbx
```

On remarque que 0x65 n'est pas divisible par 3, mais 0x165 l'est, et son tiers est 0x77 soit 119. Il faut donc effectuer 119 fois `add %al, (%rax)` et donc arriver à l'adresse 0x0x49e7e541bf4A (et au passage modifier l'ELF pour que cette page soit mappée) et donc que ce dixième octet soit 0x89.

l'étape suivante est

```
0x352845ab3bce: lea    0x0(%rip),%rbx        # 0x352845ab3bd5
0x352845ab3bd5: xor    0xc(%rsp),%ebx
0x352845ab3bd9: cmp    $0xa9b00f5c,%ebx
0x352845ab3bdf: movabs $0x352845ab3bf9,%rbx
0x352845ab3be9: movabs $0x59cb440c440c,%r14
0x352845ab3bf3: cmove  %r14,%rbx
0x352845ab3bf7: jmpq   *%rbx
```

Le xor des 4 derniers octets du mot de passe avec 0x45ab3bd5 doit être 0xa9b00f5c. On a donc les 4 derniers octets du mot de passe.

Étape suivante

```
0x59cb440c4524: movabs $0x59cbc8cc0b83,%rbx
0x59cb440c452e: mov    0x19(%rip),%rcx        # 0x59cb440c454e
0x59cb440c4535: push   %rax
0x59cb440c4536: mov    0x10(%rsp),%eax
0x59cb440c453a: xor    %rbx,%rax
0x59cb440c453d: movabs $0x2a7ee24aae0c,%rbx
0x59cb440c4547: cmpxchg %rbx,%rcx
0x59cb440c454b: pop    %rax
0x59cb440c454c: jmpq   *%rcx
```

Ce que ce code vérifie, c'est si `0x59cbc8cc0b83 xor AABBCCDD` vaut bien `0x59cb440c454e`, avec AABBCCDD les octets 8 9 10 et 11 du mot de passe rentré.

Du coup, on connait les octets 1 2 3 4 8 9 10 11 12 13 14 15 et 16 du mot de passe attendu. Reste plus que 4 octets.

L'étape suivante est :

```
0x2a7ee24aae25: push   %rax
0x2a7ee24aae26: lea    0x18(%rsp),%rdi
0x2a7ee24aae2b: lea    0x42(%rip),%rsi        # 0x2a7ee24aae74
0x2a7ee24aae32: mov    $0xa,%ecx
0x2a7ee24aae37: rep movsb %ds:(%rsi),%es:(%rdi)
0x2a7ee24aae39: mov    0xc(%rsp),%ebx
0x2a7ee24aae3d: xor    %ebx,0x1c(%rsp)
0x2a7ee24aae41: fclex
```

```
0x2a7ee24aae44: fldt   0x18(%rsp)
0x2a7ee24aae48: fld    %st(0)
0x2a7ee24aae4a: fcos
0x2a7ee24aae4c: fcompp
0x2a7ee24aae4e: fstsw  %ax
0x2a7ee24aae51: and    $0xffdf,%ax
0x2a7ee24aae55: cmp    $0x4000,%ax
0x2a7ee24aae59: movabs $0x99a3805740c,%rbx
0x2a7ee24aae63: movabs $0x2a7ee24aae7e,%r14
0x2a7ee24aae6d: cmovne %r14,%rbx
0x2a7ee24aae71: pop    %rax
0x2a7ee24aae72: jmpq   *%rbx
```

Ce code va charger les octets manquants dans ebx, xorer ebx avec 0x24b87838, le résultat va donner les 4 octets de poids fort d'un long double (nombre à virgule flottante sur 10 octets), qu'il va donner au co-processeur x87 de calculs à virgule flottante, et vérifier que le nombre obtenu vérifie cos(x) = x.

WolframAlpha donne la solution à cette équation : 0.7390851332151606416553 ...

Avec un peu de C, on trouve la valeur hexadécimale attendue : 0x3FFEBD34AEEC1E7170, et ainsi on en déduit les 4 octets manquants du mot de passe.

Une fois le jump effectué, le programme execute une dernière fonction qui applique une transformation au mot de passe et l'imprime, donnant la clef hexadécimale à donner au garde.

4 3eme Partie

On arrive dans le sous-sol du batiment, plusieurs personnes sont là, et 4 épreuves, dont 2 à deux points nous sont proposées.

Attaquons-nous à strange, qui vaut deux points...

4.1 Strange

Strange est un zip qui contient deux fichiers, 196 et a.out. 196 est un fichier binaire, qui présente une certaine particularité lorsqu'on regarde le contenu en hexadécimal :

Les octets aux adresses qui finissent par 7 ou F valent pratiquement tous 0x3F. La plupart des autres valent 0xBF.

a.out est un executable IA64. IA64 était une des architectures proposées pour les processeurs CISC 64 bits. Cette architecture n'a pas beaucoup été utilisées, les entreprises préférant amd64, qui étendait x86. Par conséquent, l'étudier, va s'averer fastidieux. radare2 par exemple ne connait pas IA64 et donne un désassemblage completement à coté de la plaque.

Heureusement, ida64 sait le lire, bien qu'il ait quelques problèmes à reconnaitre des zones de code. On va donc devoir identifier certaines parties du code, ainsi que les fonctions à la main.

Le point d'entrée est cependant désassemblé. Il consiste principalement en un appel à `__libc_start_main`. C'est là que quelques points semblent important à comprendre, sous peine d'arrachage de cheveux le temps de comprendre. Il y a pas de convention d'appel simple en IA64 comme c'est le cas dans la plupart des autres assembleurs. À la place, IA64 a la commande alloc i,l,o,s qui va permettre de reserver et prédéclarer les registres r32 à r128. Les valeurs qui vont nous être importantes ici sont l et o. l désigne les variables locales et o les variables d'output. Ainsi, au sein d'une fonction, ce vont être les registres d'output qui vont être utilisés pour valeurs les valeurs eds registres aux fonctions, qui vont être, au sein de ces fonctions, les premiers registres de manière générale. Les registres changent automatiquement de valeur lors d'entrée dans les fonctions.

Ici, dans start, on "alloue" 7 registres à l'output, ce qui implique que tous les appels de fonctions au sein de start prendront au maximum 7 arguments. Le premier argument de `__libc_start_main` est le plus interessant, en suivant le code assembleur de cette fonction (disponible sur le git de glibc, par exemple https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ia64/start.S;hb=HEAD). Cet argument est l'adresse à laquelle l'adresse de main se trouve. Donc en 0x40000000004FBCC0 se trouve l'adresse de main, 0x400000000019C700.

À cette addresse se trouve que de la data, Il s'agit de dire à IDA qu'il s'agit de code pour le forcer à le désassembler. IDA a la fâcheuse tendance à ne pas désassembler tout main, donc il faut reperer où il s'est arreté et lui dire de continuer. Une fois arriver à la din de la fonction, on peut revenir au début et faire savoir à IDA qu'il s'agit d'une fonction (touche P par exemple), et voilà, main est pret à être étudier comme d'habitude sous IDA.

Il faudra faire ainsi pour chaque fonction à désassembler. C'est fastidieux. Une autre option est d'écrire le script IDA qui le fait tout seul, mais je n'ai pas cherché en ce sens.

Main fait beaucoup de choses. On s'aperçoit qu'IDA ouvre deux fichiers, l'un étant 196, l'autre est passé en premier argument. Le format de ce fichier est ensuite testé : Il doit s'agir d'un fichier dont les 4 premiers octets sont `P2\n\#` et où la ligne suivante contient deux nombres, et celle d'encore après le produit de ces deux nombres, devant ici être 12800.

Ce format de fichier correspond à un format d'image PGM en niveau de gris.

Plus loin dans la fonction, on s'aperçoit que cette image doit être en 640x20, et que les pixels sont lus et convertis en flottants. (Des flottants 64bits en IA64, c'est-à-dire des doubles. un pixel noir correspondra à 0, et un pixel blanc à 1.)

La fin de la fonction main est celle qui va véritablement traiter les données. Afin d'éviter trop de parole pour rien, voilà un pseudocode correspondant à la fin de la fonction :

```
for(i = 0; i <= 31; i++) {
    printf(".\n")
    fread(r43, 8, 65860, r49_file196)

    for (j = 0; j < 20; j++) {
        for(k = 0; k < 20; k++) {
            out = 20 * j + k

            r15 = r43 + (out * 8)
            [r15] = floatpixels[j][20*i+k]   # floatpixels étant la table de floats tirés
                                             # de l'image donnée en argument.
        }
    }
    r8 = call sub_400000000019C410(r43)
    assert r8 == 0
    call sub_400000000019AFC0(r43, r36)

    assert (float)[r36] < 0.15
    assert (float)[r41] < 0.15
    assert (float)[r39] < 0.15
    assert (float)[r38] < 0.15
}
```

On s'aperçoit ainsi que 196 est en réalité un fichier contenant 65860*32=210750 doubles. 3f et bf en octets de poids fort dans le format double indique des nombres de la forme 0.x et -0.x, ce qui explique leur surabondance remarquée plus haut.

Chose interessante, les 400 premiers octets de chaque bloc de 65860 octet n'est pas lu, directement réécrasé.

La première fonction lance une série de vérifications et de tests de sanité sur l'entrée.

La deuxième semble assez indigeste, faisant appel à 161 fonctions, les 160 premières étant très similaires, la dernière semblant cumuler 4 de ces fonctions.

Avant de regarder plus en détails ces 161 fonctions, essayons de voir à quoi correspondent les 400 octets écrasés de chaque bloc de 196.

Avec

```
def convert_float_to_int(float_in):
    return int(float_in)

def convert_read_value_to_float(hex_in):
    return struct.unpack( "<d", hex_in)

f196_handle=open("196","rb")
img = Image.new('RGB',(640,20))
pixels=img.load()
```

```
for n in range(32):
    bloc = f196_handle.read(8*65860)
    for i in range(20):
        for j in range(20):
            value = convert_float_to_int(256*convert_read_value_to_float(bloc[(i*20+j)*8:(i*
            pixels[20*n+j,i]=(value,value,value)

img.save("196.png")
```

On obtient l'image suivante :

32 images représentant chacune un chiffre.
Visiblement, les fonctions font de l'ocr et attendent la représentation graphique d'un caractère hexadécimal pour valider chacun des 32 caractères entrés, qui doit être la clef attendue par le garde suivant.

Dommage, on n'a pas de a-f. Il faudra surement creer des images a b c d e f dans des polices similaires pour avoir la clef.

Mais ne grillons pas les étapes. Comprenons les sous-fonctions.

En regardant les différences dans les sous-fonctions, on s'aperçoit que chacune des 160 premieres fonctions prend chacun des pixels de l'image passées en entrée, et prend des valeurs à des offsets bien particuliers où ils sont les seuls à les prendre. Chacune de ces fonctions se réserve ainsi 405 valeurs.

Les opérations de fin peuvent être surprenantes : on prend un résultat, on le convertit en son opposé, on rajoute un et on fait appel à encore une autre fonction.

Cette autre fonction est assez simple : elle commence par prendre l'inverse de son argument et fait une série d'opérations assez simple en ajoutant des carrés à une somme. En la réimplémentant, on s'aperçoit que si le nombre en entrée est assez petit, cette fonction renvoie l'inverse de son argument, c'est à dire qu'elle fait l'opération 1/x. Il s'agit probablement d'une augmentation de précision de l'intruction assembleur d'inverse, probablement en appliquant la méthode de Newton sur quelques itérations.

Donc, on fait prend plein de nombres, puis on applique la fonction $x \mapsto \frac{1}{1+\exp -x}$. Cette fonction s'appelle la fonction sigmoïde et est en général utilisée pour des réseaux de neurones.

La structure des fonctions précédentes nous permet de deviner que ce réseaux de neurone possède une structure de perceptron à 3 couches, la première étant composée des 400 = 20x20 pixels du fragment d'image étudié (un caractère), la seconde de 160 neurones (nos 160 fonctions similaires prenant 405 offsets propres), et la dernière de 4 neurones (à la fois parce que la 161e fonctions de `sub_400000000019AFC0` ressemble à une version modifiée des autres fonctions répétée 4 fois (notamment on voit 4 sigmoïdes), et également parce que notre boucle vérifie 4 valeur, possiblement les outputs des 4 neurones de sorties.)

Nous avons à présent la structure de notre programme : il s'agit d'un programme d'ocr par réseau de neurones simples.

Regardons plus en détail le calcul d'un neurone : à quoi servent nos 405 variables ? Fort heureusement les offsets sont pas chamboulés d'un neurone à l'autre. Donc si je prends le k-ième double de ces 405 variables pour un neurone, le k-ième double des 405 variables pour un autre neurone aura exactement la même fonction. Pour des raisons de simplicité, j'appellerai le i-ième double de ces blocs de 405 le double `[400+i+k*405]`.

Retraçons à présent les calculs effectués :

```
[400+k*405] <-- [401+k*405]
[401+k*405] <-- pix[0...399].[403+k*405...802+k*405]+[402+405*k] = somme
[803+k*405] <-- sigmoïde(somme)=s
[804+k*405] <-- s*(1-s)
```

Le . entre les deux tableaux dénote un produit scalaire : le premier élément du premier tableau fois le premier élément du deuxième tableau plus le deuxième élément du premier tableau fois le deuxième élément du deuxième tableau plus le troisième élément du premier tableau fois le troisième élément du deuxième tableau et ainsi de suite.

Regardons à présent les quatre derniers neurones. On s'aperçoit que la structure est identique. Les blocs ne font plus que 165 doubles (160 coefficients pour les 160 neurones précédents + un biais + 4 variables auxiliaires.). On s'aperçoit également que ces neurones vont chercher les outputs de 160 neurones de la couche précédente en regardant dans les cases de la forme `[803+k*405]`.

Nous sommes à présent prêt à recoder les neurones dans un langage plus facile à manier, comme python. Pour commencer, on a déjà des inputs possibles, on peut essayer d'essayer tous les inputs sur les 32 réseaux de neurones, et voir lesquels vérifient les conditions de sorties (ie : les outputs des 4 neurones doivent être inférieurs à 0.15).

Voici le code réécrit :

```
import math
import numpy
import struct

def to_float(i):
    return struct.unpack("<d",i)[0]

def sigmoide(x):
    return 1/(1+math.exp(-x))

def neuron(data,float_input,nb_input):
    np_coeff = numpy.array(data[3:3+nb_input])
    np_input = numpy.array(float_input)
    somme = data[2] + numpy.dot(np_coeff,np_input)
    return sigmoide(somme)
```

```
f196_handle=open("196","rb")
NN_raw = [f196_handle.read(8*65860) for i in range (32)]
NN_floats= [ [to_float(NN_raw[n][8*i:8*(i+1)])for i in range(65860)] for n in range(32)]

pix = [NN_floats[n][0:400] for n in range(32)]

limit = 0.15
num_possibilities_pre = [0 for n in range(32)]
possibilities_pre=[ [] for n in range(32)]
for n in range(32):

    float_list=NN_floats[n]
    for j in range(32):
        float_input = pix[j]
        middlelayer=[neuron(float_list[400+405*i:400+405*(i+1)],float_input,400)
           for i in range(160)]
        endlayer = [neuron(float_list[65200+165*i:65200+165*(i+1)], middlelayer, 160)
           for i in range(4)]
        if abs(endlayer[0]) < limit
           and abs(endlayer[1]) < limit
           and abs(endlayer[2]) < limit
           and abs(endlayer[3]) < limit:
            print("match found for NN Number {} : {} : {}".format(n,j+1,endlayer))
            if j < 10:
                num_possibilities_pre[n]+=1
                possibilities_pre[n].append((j+1)%10)

num_possibilities = [6 if num_possibilities_pre[n] == 0 else num_possibilities_pre[n]
   for n in range(32)]
possibilities = [ ['a','b','c','d','e','f'] if num_possibilities_pre[n] == 0
   else possibilities_pre[n] for n in range(32)]

print(num_possibilities)
print(possibilities)
```

On s'aperçoit assez vite que si un chiffre matche un réseau de neurone, les autres représentations de ce même chiffre matche également le réseau de neurone, ce qui est ce qu'on attendait. (d'où le fait de n'enregistrer les images qui matchent que si c'est une des dix premières).

Par chance, il s'avère que les seuls chiffres dont on avait des exemples (ceux de 1 à 9, pas de a à f) permettent d'obtenir une solution, que le garde accepte !!!

On a ainsi fini le challenge SSTIC !!! À moins que ...

5 Conclusion

Le troisième garde passé, on arrive dans une nouvelle salle, avec une musique de fanfare, et le seul personnage, le roi, nous fait directement télécharger le fichier final.txt, dont le contenu est

```
Coucou !

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

La dernière ligne n'est pas compréhensible, on dirait un dirait ne portant que sur les caractères alphabétiques. Aussi, on s'attend à trouver une adresse e-mail en @sstic.org et on voit une répétition de ff après le @, le nombre de lettres étant le bon ...

On dirait fortement un chiffrement monoalphabétique. De plus, après une simple vérification, f et s sont espacés de 13 lettres dans l'alphabet. Ça me semble assez clair que c'est un rot13. On cherche une bonne implémentation de rot13 (et non une qui n'applique le rot13 que sur les minuscules ...) et la dernière ligne devient :

```
V01c1 l'4dr3553 m41l : 8L6q5w9Il88UHTUsTFXfWidN@sstic.org
```

Tout de suite plus compréhensible, malgré le l33t présent...

Le challenge se termine là, avec beaucoup de joie !