Solution

du

Défi

organisé dans le cadre du

Symposium sur la Sécurité des Technologies de l'Information et de la Communication

Chapitre 1 - Introduction

L'aventure débute lorsqu'un chevalier errant découvrit, à l'issue d'une longue et périlleuse quête, un légendaire et ancien PCAP (Parangon Contenant Abituellement des Perles).

Quelle ne fût pas la surprise de notre preux compagnon lorsqu'il découvrit la véritable nature de ce trésor ! A la place des perles promises, celui-ci ne trouva qu'un ramassis de morceaux de TCP (Très Commun Parangon), qu'il ne savait comment rassembler.

Invoquant alors, à l'aide d'un parchemin de puissance, l'avatar terrestre du légendaire Requin des Câbles, il pût alors requérir son assistance, et réassembler son trésor, produisant alors un merveilleux ZIP (Zèbre Inspirant la Puissance). Celui-ci, dès son apparition, envoya notre héros directement dans une contrée mystérieuse...

Chapitre 2 – Les gens du Village

La contrée dans laquelle se retrouva notre brave héros était peuplée d'étranges personnages. Se rendant en premier lieu à l'auberge du Cactus, haut lieu de débauche des pays de l'Ouest, il pût aborder plusieurs héros légendaires peuplant les histoires contées à travers tout le continent :

- Sir Renzo
- Le Chevalier de la Serpillère et son fidèle écuyer
- E. F. Perseus, le pourfendeur de l'AES
- L'Ange Polyglotte
- Le Seigneur du Grenier
- Et de nombreux autres personnages dont l'énumération serait inutile et fastidieuse

Après une partie de mots-croisés bien méritée, notre chevalier rejoignit le nord du village afin de rencontrer un groupe d'autochtones désirant lui remettre des quêtes.

Un triumvirat accueillit notre héros, et chacun d'entre eux l'envoya à la recherche d'une clé permettant de quitter le village :

- Le premier homme souhaite retrouver un fantôme ;
- Le second a caché sa clé dans du matériel Texan ;
- Au troisième, proposant des activités indécentes, notre héros ne répondit pas.

La quête du Fantôme

Une capture de champ électromagnétique fût remise à notre brave, dans le but de l'étudier pour comprendre ce que le fantôme avait à dire.

Par chance, la besace du preux chevalier était bien remplie, et un outil magique fabriqué par un grand magicien du pays des fromages à trous lui permis immédiatement d'entendre la voix du fantôme. Si cet outil ne lui avait pas été disponible, il aurait pu se tourner vers l'acquisition d'un Chopchop auprès d'un bon armurier…

Les communications suivantes étaient particulièrement intéressantes aux oreilles de notre héros :

```
2016-02-29T13:51:12.692232|Keyboard Data|0x00000000     -  [2016/02/27 - 23

2016-02-29T13:51:12.692232|Keyboard Data|0x00000010     -  :15] sstic2016-s

2016-02-29T13:51:12.692232|Keyboard Data|0x00000020     -  tage1-solution.z

2016-02-29T13:51:12.692232|Keyboard Data|0x00000030     -  ip - Saisir mot

2016-02-29T13:51:12.692232|Keyboard Data|0x00000040     -  de passe

2016-02-29T13:51:12.693916|Keyboard Data|0x00000000     -  C

2016-02-29T13:51:12.695411|Keyboard Data|0x00000000     -  y

2016-02-29T13:51:12.696929|Keyboard Data|0x00000000     -  b

2016-02-29T13:51:12.698457|Keyboard Data|0x00000000     -  3

2016-02-29T13:51:12.699976|Keyboard Data|0x00000000     -  r

2016-02-29T13:51:12.701498|Keyboard Data|0x00000000     -  S

2016-02-29T13:51:12.703031|Keyboard Data|0x00000000     -  S

2016-02-29T13:51:12.704543|Keyboard Data|0x00000000     -  T

2016-02-29T13:51:12.706244|Keyboard Data|0x00000000     -  I

2016-02-29T13:51:12.707734|Keyboard Data|0x00000000     -  C

2016-02-29T13:51:12.709257|Keyboard Data|0x00000000     -  _

2016-02-29T13:51:12.710772|Keyboard Data|0x00000000     -  2

2016-02-29T13:51:12.712301|Keyboard Data|0x00000000     -  0

2016-02-29T13:51:12.713826|Keyboard Data|0x00000000     -  1

2016-02-29T13:51:12.715348|Keyboard Data|0x00000000     -  6
```

Il s'agit vraisemblablement d'une incantation permettant l'ouverture d'un Parangon.

Celui-ci est également transmis par le fantôme :

```
2016-02-29T13:51:13.046098|File Size|Download 'C:\Users\sstic\Documents\Challenge SSTIC
2016\Stage 1\sstic2016-stage1-solution.zip' (234 bytes)

2016-02-29T13:51:13.049173|0x00000000     00 00 00 00 00 00 00 00  50 4B 03 04 0A 00 09 00  -
........  PK......

2016-02-29T13:51:13.049173|0x00000010     00 00 38 76 5D 48 E1 1B  DF 65 2C 00 00 00 20 00  -
..8v]H..  .e,... .

2016-02-29T13:51:13.049173|0x00000020     00 00 0C 00 1C 00 73 6F  6C 75 74 69 6F 6E 2E 74  -
......so  lution.t

2016-02-29T13:51:13.049173|0x00000030     78 74 55 54 09 00 03 7B  4C D4 56 A5 4C D4 56 75  -
xtUT...{  L.V.L.Vu

2016-02-29T13:51:13.049173|0x00000040     78 0B 00 01 04 E8 03 00  00 04 E8 03 00 00 03 6F  -
x.......  .......o

2016-02-29T13:51:13.049173|0x00000050     8E DF 08 F1 6C A5 14 EE  5C B3 4B A5 CE 77 DE 38  -
....l...  \.K..w.8
```

```
2016-02-29T13:51:13.049173|0x00000060     97 A5 EE 25 E7 62 60 E0  39 75 6B 7E BE 57 7B 4A  -
...%.b`.  9uk~.W{J

2016-02-29T13:51:13.049173|0x00000070     30 DB 07 70 14 D5 DB C1  30 27 50 4B 07 08 E1 1B  -
0..p....  0'PK....

2016-02-29T13:51:13.049173|0x00000080     DF 65 2C 00 00 00 20 00  00 00 50 4B 01 02 1E 03  -
.e,... .  ..PK....

2016-02-29T13:51:13.049173|0x00000090     0A 00 09 00 00 00 38 76  5D 48 E1 1B DF 65 2C 00  -
......8v  ]H...e,.

2016-02-29T13:51:13.049173|0x000000a0     00 00 20 00 00 00 0C 00  18 00 00 00 00 00 01 00  -
.. .....  ........

2016-02-29T13:51:13.049173|0x000000b0     00 00 A4 81 00 00 00 00  73 6F 6C 75 74 69 6F 6E  -
........  solution

2016-02-29T13:51:13.049173|0x000000c0     2E 74 78 74 55 54 05 00  03 7B 4C D4 56 75 78 0B  -
.txtUT..  .{L.Vux.

2016-02-29T13:51:13.049173|0x000000d0     00 01 04 E8 03 00 00 04  E8 03 00 00 50 4B 05 06  -
........  ....PK..

2016-02-29T13:51:13.049173|0x000000e0     00 00 00 00 01 00 01 00  52 00 00 00 82 00 00 00  -
........  R.......

2016-02-29T13:51:13.049173|0x000000f0     00 00                                            -
..
```

Une fois procédé à son extraction, son ouverture ne pose pas de problème connaissant la formule « Cyb3rSSTIC_2016 ». La clé apparait finalement :

368BE8C1CC7CC70C22450309343O1C15

La quête Texane

Les populations Texane vivant sur l'autre rive de la grande mer des Atlantes disposent d'objets technologiques avancés fournis par la guilde des magiciens locale, dont le fameux Tambour Incongru, modèle 83, dans sa variante avancée (aussi nommé TI83+).

Un chevalier digne de ce nom dispose de solides alliances avec toutes sortes de créatures. L'une d'entre elles est la sorcière Ida, mais notre ami décida de ne pas faire appel à elle immédiatement, et utilisa plutôt l'un des nombreux outils de la guilde les mages, accessibles par invocation directe à distance en tous points du globe.

Cela lui permit d'obtenir la formule originale qui avait permis la création du Tambour Incongru remis par le villageois.

Ladite formule était rédigée dans un langage magique basique, que notre héros pu déchiffrer seul. Ses maigres connaissances de magie des arcanes lui permirent de reconnaitre

la formule du Cycle des Rameaux Canoniques de la 32e lune (aussi appelé, parfois, par sa version courte, CRC32). Le sésame à présenter à l'objet devait, après transformation par la formule, correspondre à la rune C49ABZ57.

Notre chevalier, dont la force n'a d'égale que la détermination, décida alors de tenter toutes les combinaisons possibles lui permettant d'arriver au résultat escompté, ce qui lui donna rapidement (car il est rapide !) la solution 8959902.

Le Tambour Incongru libéra donc sa clé, concluant la quête.

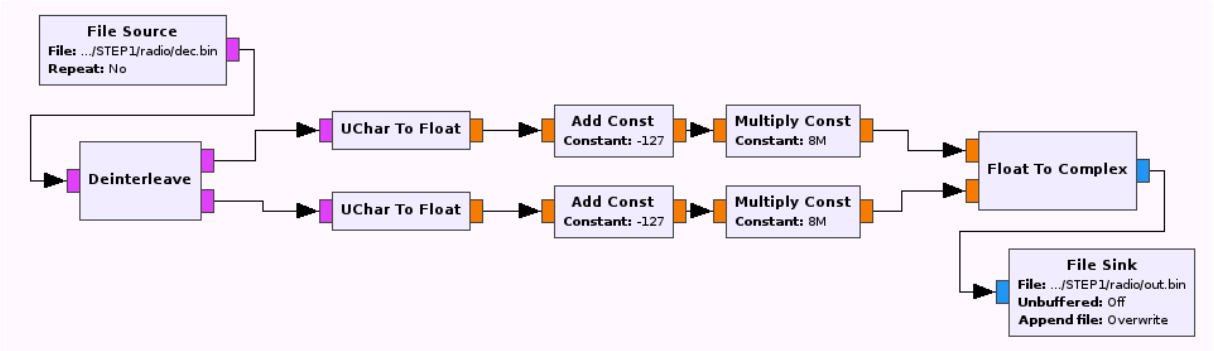
57D9F82B49C1EB3993CB82D26E37F69C

La quête du Golem Supérieur Magistral (GSM)

Repensant alors aux propositions indécentes du villageois précédemment rencontré, le chevalier se dit qu'il avait bien quelques minutes devant lui pour aller s'amuser un peu !

L'homme à la grosse antenne remis à notre héros un document donc les marques indiquaient qu'il s'agissait d'une qûete impliquant un GSM.

Après décompression et modification runique, le document était prêt à être analysé !



Dans sa grande besace, le chevalier a toujours à sa disposition un Gros Réducteur pour Golem Supérieurs Magistraux (soit GRGSM), lui permettant aisément de venir à bout de ce genre de créatures.

Un premier passage par le grand Requin des Câbles lui révéla la vraie nature du Golem ! Celui-ci agissait principalement sur le 4e plan Supérieur des Dryades Cochonnes et Cachées dans un Hêtre (SDCCH4). Cette information en main, le Requin des Câbles ne fit qu'une bouchée du golem, libérant la clé.

```
▾ TP-User-Data
    SMS text: Bonjour, votre cle est 1ac3d8c409e656380a06f6f2c6de6b4a
```

1AC3D8C409E656380A06F6F2C6DE6B4A

En possession des deux clés simples et de la double, notre héros fringuant pris la route de l'Est après que le garde lui ait laissé le passage...

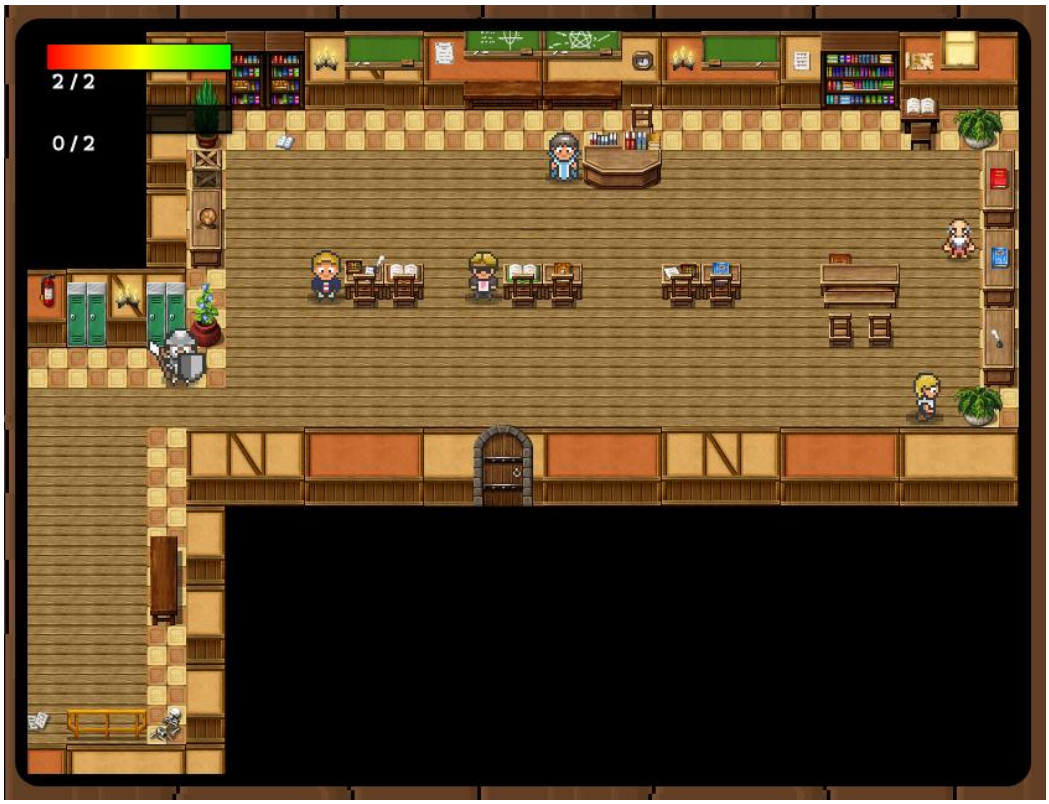

2 / 2
0 / 2
Super ! Tu peux passer.
57D9F82B49C1EB3993CB82D26E37F69C

Chapitre 3 – Les neiges enneigées

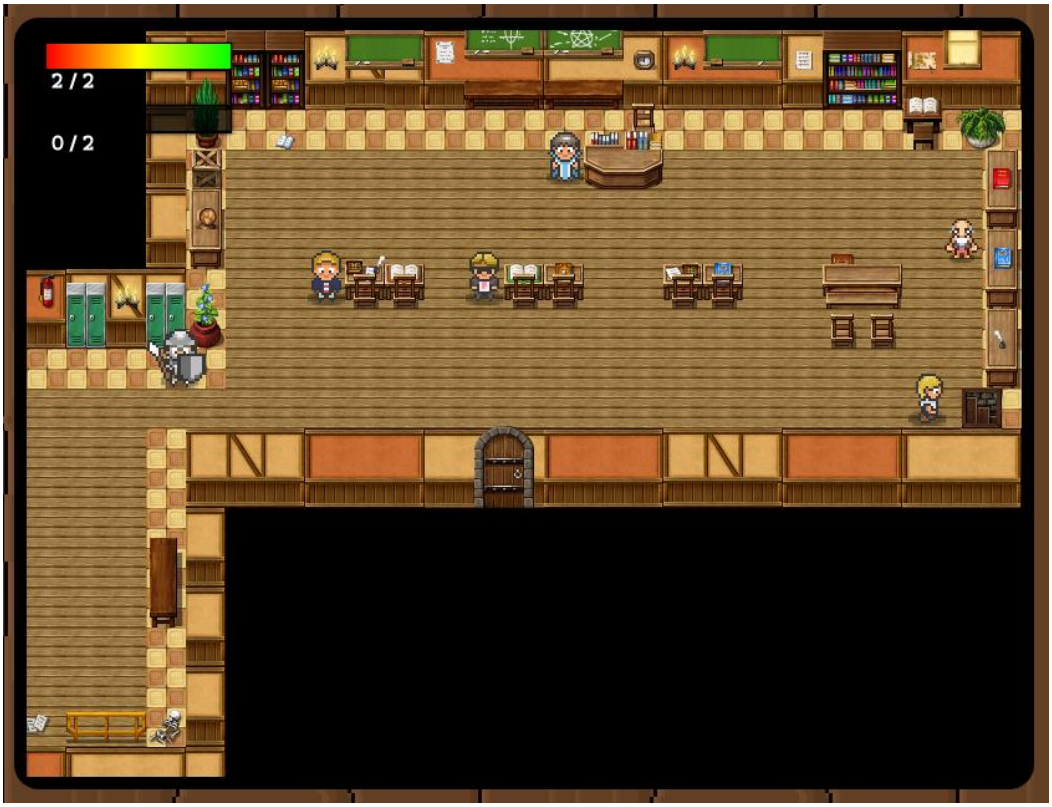
Notre héros poursuivit sa route dans les terres enneigées de l'Ouest, et ne tarda pas à trouver une demeure peuplée de quelques villageois.

Passé la discussion avec un lanceur d'alerte, les trois villageois restant proposèrent chacun une quête, dans le but de récupérer à nouveaux des clés permettant à notre chevalier de continuer sa route.

Mais une étrange plante verte attira rapidement l'attention du chevalier, de par sa chevelure étincelante…



Une rapide négociation avec elle libère alors un passage secret, sous forme d'un escalier, menant directement dans une nouvelle zone, sans aucun besoin de réaliser les quêtes proposées !



Oh oh oh ! Il ne s'agissait là que d'un calembour ! Nulle magie ne permet d'outrepasser l'un des gardes de ce monde !

Reprenons donc nos quêtes…

Deux clés étaient nécessaires pour passer le garde, mais trois quêtes furent disponibles :

- Le combat avec un ELF (Elementaire Légendaire Faramineux) géant ;
- L'extraction et l'analyse minutieuse d'une table de caractères runiques ;
- L'affrontement avec un EFI (Elementaire Foncièrement Imbuvable).

Notre chevalier n'ayant aucune velléité à affronter une créature imbuvable, il choisit rapidement les deux premières missions proposées…

La quête de l'ELF géant

L'ELF géant fut fourni à notre héros enfermé dans une lampe magique.

Malheureusement, le plan actuel (nous nous trouvons actuellement dans la 4e galaxie Elementaire du Xylophone Termitophage, aussi nommée EXT4) ne permettant pas la libération d'un ELF aussi imposant, notre chevalier utilisa son parchemin de téléportation majeure afin de se rendre dans la galaxie du Xylophone Faramineux Stoïque (parfois nommée XFS dans les cartes spatiales), et la lampe libéra alors son hôte.

Il fallait maintenant étudier l'ELF, afin de savoir quel mot runique prononcer pour qu'il libère la clé qu'il renfermait. Notre vaillant héros appela alors la sorcière IDA pour lui demander son aide.

Celle-ci, une fois arrivée, ne put que constater que l'élémentaire était bien trop imposant pour être directement étudié sous sa forme libérée. Elle le renferma alors dans sa lampe, et en extrait sa version diminuée, bien plus facile à observer.

La sorcière commença à rédiger une carte sur un parchemin, afin d'établir la correspondance entre les diverses parties du corps de l'ELF libéré et celles de sa variante atrophiée.

Dès lors, l'étude des mécanismes de défense de l'ELF peut être effectuée directement sur la version atrophiée. La majeure partie du mot runique à trouver est rapidement identifiée, mais deux vérifications sont plus complexes que les autres.

29..7ED1.....D54EC08C89341BEC

La première utilise une variation de la pile, positionnant celle-ci de telle sorte qu'elle termine par 03, et saute ensuite dans des instructions nulles en fonction de la valeur de l'octet 1. Ces instructions ont pour effet d'incrémenter une zone de 0x3 à chaque instruction. La valeur finale devant être de 0x65, la seule valeur multiple de 3 correspondant est 0x165, ce qui implique 0x165/3, soit 0x77. La valeur finale doit donc être 0x100-0x77, soit 0x89.

La seconde vérification consiste à résoudre la formule magique $\cos(x) = x$, x étant un flottant sur 10 octets. 6 octets étant fournis, il suffit alors de trouver les 4 octets manquants, ce qui se résout aisément par force brute. La valeur à trouver est alors 0x998CD6D4.

29897ED1D4D68C99D54EC08C89341BEC

Une fois fournie cette entrée à l'ELF, celui-ci libère alors sa clé :

E574B514667F6AB2D83047BB871A54F5

La quête de la table de caractères

Une table de caractères runiques est remise à notre héros, ainsi qu'un emplacement dans lequel écrire une série de runes. Si la bonne série est entrée, la tablette le fait savoir, et la clé peut être dérivée de la série.

A l'aide de la sorcière, celui-ci pu rapidement identifier qu'une formule magique spécifique était associée à chaque rune. Pour la plupart des runes, il s'agissait d'associer un nombre entre 0 et 63 à la rune, et d'affecter ce nombre à la position de la rune. Mais l'une d'entre elles cachait une formule bien plus conséquente.

Répondant aux règles édifiées par les mages TrueType, il fallut alors que notre héros se plonge dans les anciens écrits d'A-doh-bee le Grand, afin de se forger un outil spécial permettant de décoder la formule.

Une série d'équations lui apparut alors, permettant de définir quelle rune devait se trouver à quel emplacement. Voici quelques exemples de telles équations :

19 == RS[2]

5 == (RS[16] - 2))

90 == (320/64 * (64/64 * (7 + (2 + RS[9]))))

1970 == (320/64 * ((256/64 * (3 + (256/64 * ((RS[20] - 7) - 1)))) - 2))

Ces 4 équations permettent respectivement de trouver les runes pour les emplacements 2, 16, 9 et 20.

La résolution donne rapidement la bonne série de runes, produisant la suite de caractères suivante :

4ft/0AdJNjHBFW5ch+vVGw

Celle-ci ressemble aux anciennes incantations basiques de 64e niveau (aussi appelées base64), se décodant en une clé :

E1FB7FD007493631C1156E5C87EBD51B

La quête de l'EFI

Soudain pris d'un terrible remord, notre chevalier décida tout de même de s'occuper de l'EFI, ne voulant pas laisser de quête inachevée.

L'EFI était particulier, sa composition n'étant pas semblable à celle des autres créatures dites de la branche x86. La lecture de ses entrailles se faisait toutefois sans difficulté.

Passée la récupération d'une clé, sa première action est le déchiffrement basique par sortilège XOR d'une zone de données. Celle-ci correspond à la GUID (Glorieuse et Unique Incantation Difficile) d'un sortilège de décompression.

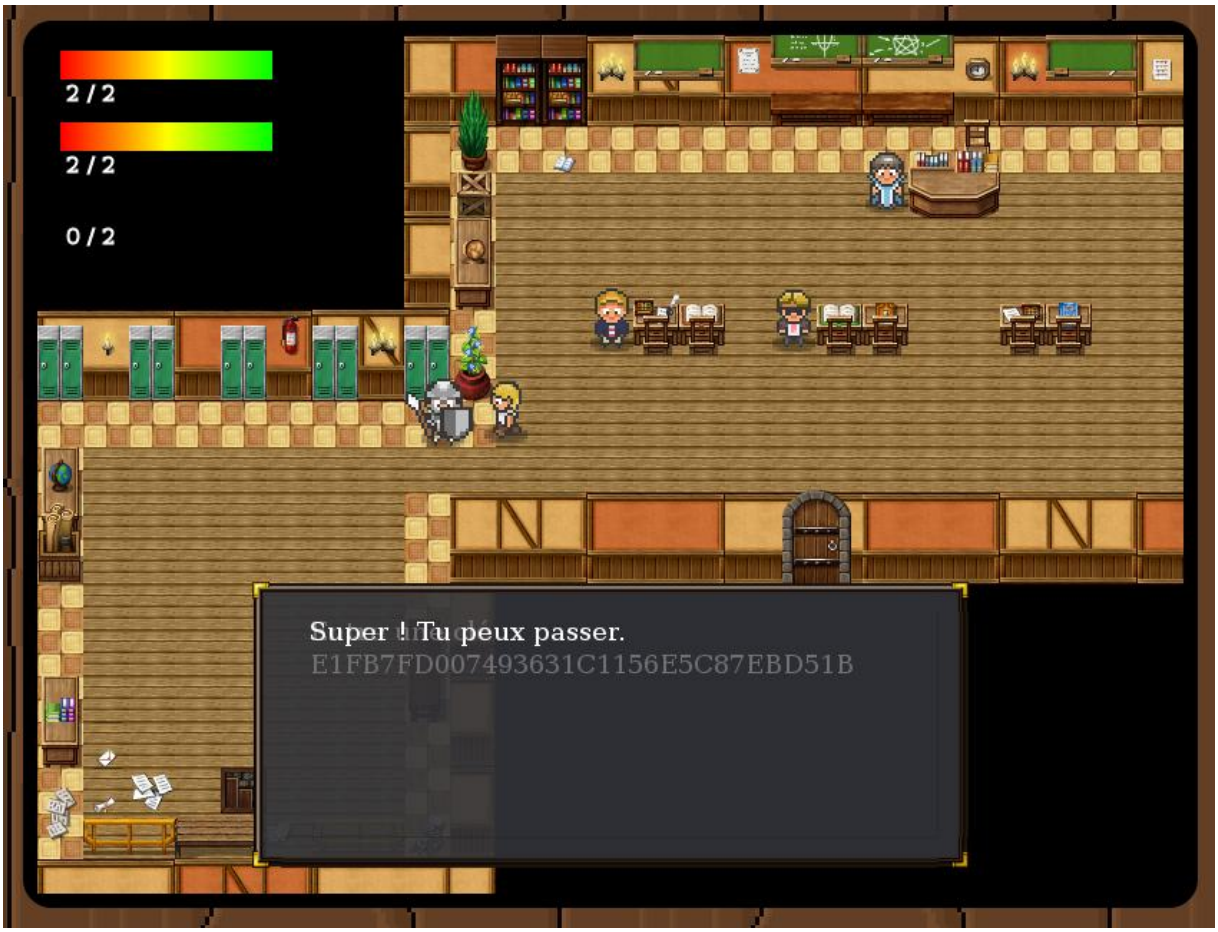
L'application de ce sortilège à une seconde zone permet l'extraction d'un indice capital :

secret data: cb41dcb1d89746705a7fe998f11acce7

La donnée secrète est ensuite récupérée, et subit les affres d'une boucle nonifiante rotationnelle par indice relatif (autrement dit, un NOT et un ROL dépendant de l'indice par octet). Une fois cette opération terminée, la clé tant attendue apparaît :

347D8C72720D6EC7A501583BE0BCCC0C

Avec 3 clés en sa possession, notre héros peut alors passer le garde, qui n'en attendait que deux…


2 / 2
2 / 2
0 / 2
Super ! Tu peux passer.
E1FB7FD007493631C1156E5C87EBD51B

Chapitre 4 – La Cave du Temps

L'escalier derrière le garde descendait vers une étrange cave abritant des technologies du futur. Finis les chevaliers, épées, dragons et autres sortilèges, place à la magie des ordinateurs, leurs processeurs et les langages assembleur exotiques, Windows et ses subtilités, et les ingénieurs docteurs en XSS !

N.B. Le lecteur déçu du changement de style comprendra bien vite que certaines solutions ne pourraient être facilement décrites en restant dans le champ lexical de la chevalerie. Le retour à une période plus médiévale se fera dans le chapitre 5.

Cette fois-ci, 4 épreuves sont proposées, 2 d'entre elles libérant directement une double clé permettant de passer au niveau suivant :

- Un Driver Windows et son client userland (« usb ») ;
- Un screensaver Windows et une vidéo (« video ») ;
- Un ELF IA-64 (« strange », clé double) ;
- Un binaire Windows x64 (« ring », clé double).

Le chemin le plus rapide vers le niveau suivant passe par les deux épreuves donnant une clé simple (« usb » et « video »).

USB

Les fichiers fournis pour ce challenge sont :

- userSSTIC.exe : un PE 64 bits
- img.bz2 : une image disque compressée

Une analyse rapide du code nous montre qu'un driver est embarqué, et chargé sur le système à l'exécution. Ensuite, le contenu du répertoire « %SystemDrive%\SSTIC\ » est listé, chaque fichier est lu, et le contenu du fichier ainsi que son hash MD5 sont copiés dans une structure. Un thread est également chargé de détecter l'insertion d'un périphérique type disque, la lettre de ce nouveau étant sauvegardée via ZwSetInformationProcess(ProcessDefaultHardErrorMode).

Curieusement, aucune IOCTL n'est effectuée vers le driver, la communication passant par un autre canal. Le binaire userland attend un changement de valeur à l'adresse 0x7FFE02C8. Cette adresse fait partie de la structure KUSER_SHARED_DATA, mappée en userland en 0x7FFE0000. Le champ en 0x2C8 est le SystemExpirationDate, et est utilisé par le driver pour passer au userland un pointeur vers une zone mémoire partagée entre user et kernelland.

Une première analyse du driver nous montre que les actions suivantes sont effectuées :

- Récupération de la lettre du disque via NtQueryInformationProcess(ProcessDefaultHardErrorMode) ;
- Création de la zone mémoire partagée via MDL, et écriture de l'adresse de cette zone dans le champ SystemExpirationDate de la structure KUSER_SHARED_DATA (mappée en 0xFFFFF78000000000) ;
- Récupération du contenu de chaque fichier lu par le userland, via un système de mutex pour que le processus userland et le driver sachent quand effectuer de nouvelles lectures/ecritures. La structure contenant le contenu de chaque fichier est chiffrée en RC4 avec une clé générée aléatoirement. Une structure contenant la clé et le contenu chiffré est ensuite allouée et ajoutée à une liste chainée ;

- Une fois tous les fichiers lus, l'intégralité de la liste est chiffrée en RC6 (clé en dur : « 551C2016B00B5F00 »), puis écrite sur le disque dont la lettre a été précédemment récupérée.

Le mécanisme de sélection des blocks dans lesquels écrire est assez long à analyser. Une observation de l'image disque montre que les partitions ne sont pas contiguës, et que des données à forte entropie se trouvent entre elles. Enfin, le code du driver nous indique que l'écriture sur le disque commencera au premier bloc composée uniquement de zéros.

La dernière difficulté est de trouver une implémentation de RC6 fonctionnelle, l'auteur ayant en plus confondu RC5 et RC6 lors de sa résolution…

La solution a donc été de « ripper » la fonction de déchiffrement, et de la réutiliser dans du code C avec quelques ajouts pour rendre compatible la convention d'appel avec Linux ☺

Dès lors, il est possible de lire les fichiers chiffrés sur le disque, et de rapidement trouver un ZIP :

```
$ ./rc5 img

$ ./rc4.py out.bin

$ file outfile*
outfile.0: ASCII text
outfile.1: ASCII text, with very long lines
outfile.2: PDF document, version 1.5
outfile.3: JPEG image data, Exif standard: [TIFF image data, big-endian,
direntries=7, orientation=upper-left, xresolution=98, yresolution=106,
resolutionunit=2, software=Adobe Photoshop CS5 Windows, datetime=2011:06:13
13:36:51], baseline, precision 8, 640x360, frames 3
outfile.4: Zip archive data, at least v2.0 to extract

$ cat outfile.0
password for the zip file : !WooYouAreSuchAnAwesomeGuy!

$ unzip outfile.4
Archive:  outfile.4
[outfile.4] 4.jpg password:
 inflating: 4.jpg
 extracting: key

$ xxd key
00000000: 0928 bde1 e3ed 8969 8632 dbff 4a23 1138
```

La clé finale est alors :

0928bde1e3ed89698632dbff4a231138

Video

Deux fichiers sont fournis pour ce challenge :

- Airlhes_screensaver_setup.exe
- Airlhes_CYBER_SECRET_possible_exfiltration.mp4

Dans la vidéo, on peut voir l'arrière d'un PC sur lequel tourne un screensaver, qui projette différentes couleurs sur le mur derrière lui.

L'exécutable est un installeur qui se charge d'installer le screensaver « Airlhes_screensaver.scr » sur la machine. Celui-ci est packé avec UPX.

Une fois dépacké, l'analyse nous montre les points suivants :

- Le contenu de la clé de registre HKCU\Software\Airlhes\ScreenSaver\config\trajectories est récupéré et stocké précédée d'un Dword correspondant à sa taille ;
- Son contenu est ensuite utilisé pour calculer les codes de couleur à afficher à l'écran.

Le code est assez pénible à analyser, la partie intéressante étant noyée autour de nombreuses routines toutes sales étant inutiles pour comprendre le challenge et des boucles de timing pour gérer le temps d'affichage à l'écran.

Pour chaque byte de la clé de registre, des opérations sont effectuées, et finalement une valeur est extraire d'un tableau de 8 entrées (en 0x405020) xorées avec la constante 0x 4FB78EB3.

Une fois déxorées, on obtient les valeurs : 0xFF0000, 0x00FF00, 0xFFFF00, 0x0000FF, 0xFF00FF, 0x00FFFF, 0xFFFFFF, 0x000000, correspondant aux couleurs Rouge, Vert, Jaune, Bleu, Violet, Bleu Clair, Blanc et Noir.

On constate ensuite que pour chaque byte d'entrée, trois couleurs sont produites en sortie. L'algorithme est le suivant :

```
Xor_byte = in_byte ^ key[i]

offset_color_1 = (Xor_byte%7 + offset_previous_color + 1)&7

color1 = colors[offset_color_1]

offset_color_2 = ((Xor_byte/7)%7 + offset_color_1 + 1)&7

color2 = colors[offset_color_2]

offset_color_3 = ((Xor_byte/49)%7 + offset_color_2 + 1)&7

color3 = colors[offset_color_3]

offset_previous_color = offset_color_3
```

La clé utilisée pour Xorer la valeur de la clé de registre est présente en 0x405040.

La variable offset_previous_color est initialisée à 6.

Il suffit alors de récupérer la séquence complète des couleurs de la vidéo, et d'inverser l'algorithme précédent.

Bien que la récupération des couleurs puisse être automatisée, l'auteur a jugé plus rapide de les récupérer à la main.

```
$ python decode.py

1800000078da0b1634172bdac558f9cb6e6257f3be7b5b003250077e
```

Les 4 premiers octets correspondant bien à la taille du reste de la chaine, le décodage semble correct.

Le début de la chaine ressemble étrangement à une en-tête zlib.

```
>>>
zlib.decompress("78da0b1634172bdac558f9cb6e6257f3be7b5b003250077e".decode("
hex")).encode("hex")

'5311371672ba0179fa3e918a83bedeb4'
```

La clé finale est alors :

5311371672ba0179fa3e918a83bedeb4

Strange

Deux fichiers sont fournis pour le challenge Strange :

- Un ELF compilé pour architecture Itanium

- Un fichier « 196 », contenant des données pour l'instant inconnues

Débuter dans ce challenge nécessite de lire un peu de documentation concernant l'architecture Itanium ! Quelques points intéressants :

- De très nombreux registres (128 « généraux », 128 contenant des flottants, etc.)
- Les fonctions récupèrent leurs arguments dans r32, r33, r34, etc.
- Les arguments passés aux sous-fonctions passent par les registres commençant après les registres utilisés localement (si une fonction utilise des registres jusqu'à r51, elle passe les arguments aux sous fonctions en commençant à r52)

De là, il est déjà possible d'identifier la fonction main en 0x400000000019C700.

Celle-ci effectue les actions suivantes :

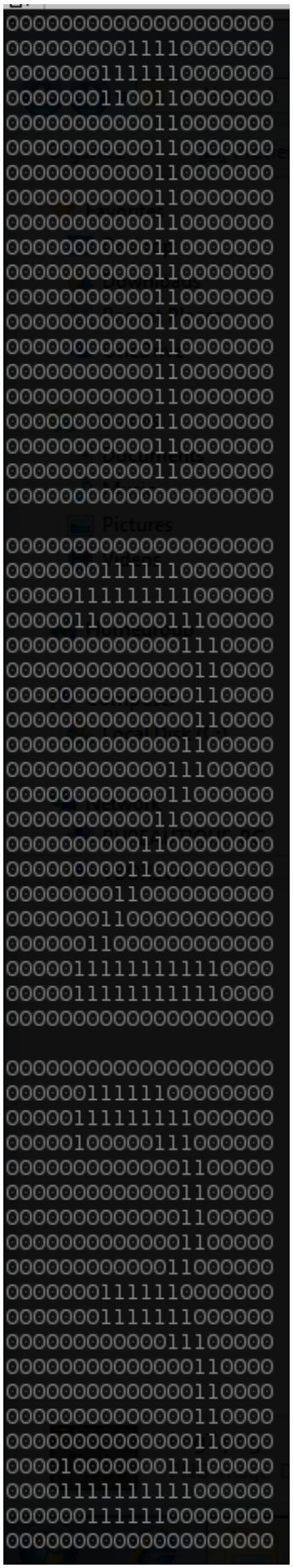
- Ouvre le fichier « 196 » en lecture binaire ;
- Alloue un buffer de 526880 octets ;
- Ouvre le fichier passé en argument en lecture ;
- Vérifie via quelques lectures que celui-ci est une image au format PGM Ascii en niveaux de gris, de 12800 pixels ;
- Lit l'image ligne par ligne (l'image fait 640x20 pixels) et vérifie que chaque pixel est soit à 0, soit à 255 (image en noir et blanc) ;
- Entre dans une boucle de 32 itérations qui va :
    - Lire 526880 octets du fichier « 196 » ;
    - Lire l'image d'entrée par carrés de 20x20 pixels dans les 3200 premiers octets du buffer ;
    - Effectue quelques vérifications basiques sur le carré de 20x20 pixels dans une première fonction ;
    - Appelle une seconde fonction prenant en paramètre le buffer ;
        - Cette fonction appelle 161 sous fonctions : les 160 premières semblent faire la même opération sur des parties différentes du buffer ;
        - La dernière fonction semble mixer les résultats des précédentes.
    - Les 4 valeurs retournées par la fonction doivent être inférieures à 0.15.

De là, plusieurs observations :

- L'image d'entrée contient 32 carrés de 20x20 pixels => cela correspond à la longueur de la clé, et chaque carré pourrait contenir un caractère ;
- Les 160 sous-fonctions de la fonction 2 effectuent la somme d'un produit de matrices, la premières matrice étant le carré de 20x20 de l'image d'entrée, et la seconde étant contenue à l'offset (3200 + k*3240 + 24) du gros buffer, pour chaque fonction k. De cette somme sont ensuite calculées 2 valeurs :
  - 1/(1-exp(x))
  - (1/(1-exp(x))) * (1 – (1/(1-exp(x)))

Quelques recherches sur ces valeurs nous amènent à des implémentations de réseaux de neurones pour la reconnaissance de caractères, ce qui semble confirmer notre première hypothèse ! Il ne reste qu'à comprendre le format du fichier 196 pour extraire l'ensemble des paramètres, et trouver les caractères à passer au réseau de neurones.

Un point reste à éclaircir, les 3200 premiers octets lus dans le fichier « 196 » à chaque itération de la boucle sont ensuite écrasés par le carré de 20x20. En extrayant ce contenu et en interprétant les valeurs comme des flottants, il est possible de redessiner des images de 20x20, qui cachent en fait les différents caractères (chiffres de 0 à 9) qu'il est possible de passer en entrée du réseau de neurones.



Une implémentation d'un tel réseau de neurones en Python est rapidement trouvée, et une fois les bons paramètres extraits du fichier, la solution tombe :

```
$ python parse_196.py 196
…
```
23425038472508287335772085544035

Ring

Un seul fichier est fourni, il s'agit d'un PE 64-bits de 3.4Mo.

Une fois lancé, le programme va créer une nouvelle instance de lui-même en passant comme paramètres de ligne de commande son PID et le chiffre 0. Chaque nouvelle instance du programme va répéter cette opération en passant le PID du premier processus, et en incrémentant le second argument, jusqu'à ce que 6 processus fils aient été créés.

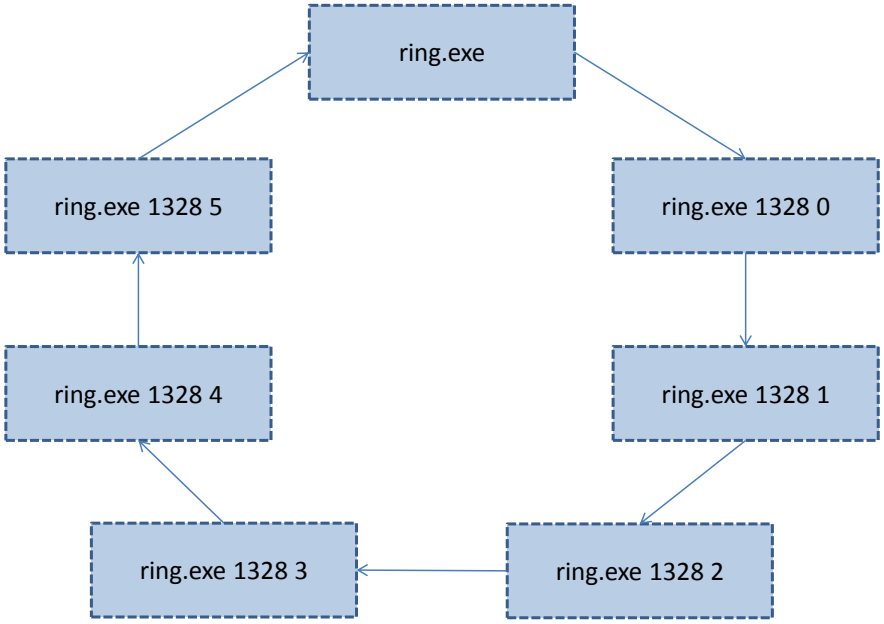
De plus, chaque parent est considéré comme debugger de son fils, le processus étant créé avec le flag DEBUG_PROCESS. Enfin, le dernier processus créé se définit comme debugger du premier processus (celui dont le PID est passé en argument) via DebugActiveProcess().

On se retrouve alors avec un anneau de processus, chacun débuggant le processus suivant dans la boucle, tel un man train[1].

On comprend alors d'où vient le nom de l'épreuve !

Cette mécanique complique alors l'analyse, car il est impossible de simplement s'attacher à l'un des processus à l'aide d'un débugger userland.

[1] Ne pas googler.



Une fois cette architecture mise en place, le 7e processus va lever une exception 0xC0000005 (ACCESS_VIOLATION), qui sera alors récupérée par le 1er processus.

Les levées d'exceptions constituent le mécanisme de communication entre les processus, les données étant passées par le biais du paramètre lpArguments de RaiseException(). Ces données contiennent, entre autres :

- L'id du processus destination
- L'id du processus source
- Des flags
- Un numéro de commande
- Des paramètres optionnels pour la commande

La première commande envoyée par le 7e processus est la 0x1337.

Voici quelques exemples de commandes :

- 0x1337 : initialise une zone de mémoire partagée et lance la demande de mot de passe ;
- 0xeeee : lance un nouveau thread, l'adresse est passée en paramètre, xorée avec 0x47ABC348712F980A ;
- 0x6666 : keylogging / gestion des touches interceptées => permet de lire le mot de passe entré par l'utilisateur ;

- 0x2016 / 0x2048 : résolution dynamique de DLL et d'exports ;
- 0x1212 / 0x1213 / 0x1214 : allocation, copie et manipulation de données chiffrées ;
- 0x4242 : anti-debug.

La zone de mémoire partagée entre les processus est un FileMapping nommé « SSTIC_ss » et mappé en 0x5571C000.

L'anti-debug est effectué par appels à GetTickCount() et vérification que le temps écoulé depuis le précédent appel n'a pas dépassé une certaine valeur. D'autres appels de ce type sont effectués aléatoirement à différents endroits dans le code.

A chaque interception d'une touche via la fonctionnalité de keylogging, celle-ci est transmise à un autre processus, ainsi que le résultat de la fonction GetTickCount(). Le processus récupérant ces informations va stocker le quartet de poids faible du caractère reçu et la valeur de GetTickCount() dans deux tableaux.

Une fois que 16 caractères de clé ont été lus, le 1er processus envoie la commande 0xeeee (création de thread) avec en paramètre la fonction 0x747C00, qui sera donc appelée par le 7e processus.

Cette fonction écrit à l'adresse 0x10 de la mémoire partagée 16 octets, en commençant par la fin. Ces octets sont composés de :

- La valeur dérivée de la touche interceptée (quartet de poids faible) ;
- Le temps mis entre chaque frappe de touche (quartet de poids fort).

S'en suis un déroulement de très nombreuses fonctions contenant majoritairement des instructions SSE, pour finir sur la fonction vérifiant si la clé est valide. Celle-ci fait les vérifications suivantes :

- L'octet 0 de la mémoire partagée est à 1 ;
- Les octets 0x1f, 0x1e et 0x1d sont à 0 ;
- L'octet 0x1c a ses 2 bits de poids fort à 0.

Appelons « clé » la valeur située en 0x10 de la mémoire partagée. Selon la fonction de vérification, on constate que nous n'avons que 102 bits utiles.

Reste maintenant à analyser environ 3Mo de code obfusqué composé majoritairement d'instructions SSE de type :

```
…
movdqa  xmm3, xmmword ptr [rsi+10h]
pinsrw  xmm6, edx, 4
push    r13
push    r14
push    r15
pshufb  xmm3, xmm6
sub     rsp, 80h
mov     rdi, 0D8C2706054441009h
mov     rdx, [rsi+10h]
movdqa  [rsp+0C0h+var_90], xmm11
movq    xmm4, rdi
movdqa  xmm0, xmm4
psllq   xmm3, 3
movdqa  xmm7, xmm2
pand    xmm2, xmm3
ror     rdx, 3Eh
mov     rdi, 67AEFF3BDFDF797Eh
and     r11, rdi
mov     rdi, 8D421420421h
pandn   xmm7, xmm11
psllq   xmm1, 6
pandn   xmm0, xmm13
por     xmm7, xmm2
movdqa  [rsp+0C0h+var_70], xmm13
…
```

On constate qu'il n'y a pas à proprement parler d'appels de fonctions, mais que chaque « bloc » d'instructions termine par une instruction « int 3 », potentiellement précédée d'un push d'adresse sur la pile.

Ces instructions servent à « passer la main » au processus suivant. Lorsque l'exception est récupérée, le processus va récupérer le contexte du thread l'ayant levé, créer un nouveau thread, y copier le contexte, terminer l'ancien thread, et positionner RIP en récupérant sa valeur dans un tableau de fonction, indexé par la valeur du registre R15.

Si une adresse a été poussée sur la pile avant le « int 3 », il s'agit d'un appel de fonction. Dans le cas contraire, on se trouve toujours dans la même fonction.

Par exécution symbolique du premier bloc de la première fonction (0x54625F), on constate que celui-ci récupère simplement **les 102 bits d'entrée**, et fait appel à une fonction (0x541112) en les passant en paramètres, en les ayant préalablement mélangés.

```
$ python read.py func_start | grep WRITE

WRITE [rsi+0x10] [70];[rsi+0x18] [13];[rsi+0x18] [23];[rsi+0x10] [54];[rsi+0x10]
[81];[rsi+0x10] [60];[rsi+0x10] [10];[rsi+0x10] [17];[rsi+0x18] [24];[rsi+0x10]
[42];[rsi+0x10] [30];[rsi+0x18] [33];[rsi+0x18] [11];[rsi+0x10] [16];[rsi+0x10] [0];[rsi+0x10]
[80];[rsi+0x10] [8];[rsi+0x10] [12];[rsi+0x10] [56];[rsi+0x10] [73];[rsi+0x10] [21];[rsi+0x10]
[3];[rsi+0x10] [28];[rsi+0x10] [51];[rsi+0x10] [53];[rsi+0x10] [45];[rsi+0x10] [9];[rsi+0x10]
[4];[rsi+0x10] [29];[rsi+0x10] [35];[rsi+0x10] [59];[rsi+0x10] [52];[rsi+0x10] [63];[rsi+0x10]
[41];[rsi+0x10] [58];[rsi+0x10] [64];[rsi+0x10] [15];[rsi+0x18]
[10];0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0

WRITE [rsi+0x10] [24];[rsi+0x10] [82];[rsi+0x10] [20];[rsi+0x10] [23];[rsi+0x10]
[84];[rsi+0x10] [40];[rsi+0x10] [7];[rsi+0x10] [31];[rsi+0x18] [32];[rsi+0x10] [98];[rsi+0x10]
[99];[rsi+0x10] [6];[rsi+0x10] [69];[rsi+0x10] [50];[rsi+0x10] [47];[rsi+0x10] [48];[rsi+0x10]
[66];[rsi+0x10] [38];[rsi+0x10] [39];[rsi+0x10] [13];[rsi+0x18] [25];[rsi+0x10]
[90];[rsi+0x10] [2];[rsi+0x10] [92];[rsi+0x10] [36];[rsi+0x10] [91];[rsi+0x10] [44];[rsi+0x10]
[61];[rsi+0x10] [33];[rsi+0x10] [43];[rsi+0x18] [8];[rsi+0x10] [67];[rsi+0x10] [94];[rsi+0x18]
[19];[rsi+0x10] [46];[rsi+0x10] [57];[rsi+0x18] [31];[rsi+0x10] [25];[rsi+0x10]
[19];[rsi+0x10] [27];[rsi+0x10] [49];[rsi+0x10] [18];[rsi+0x10] [26];[rsi+0x18]
[14];[rsi+0x10] [79];[rsi+0x18] [1];[rsi+0x10] [11];[rsi+0x10] [1];[rsi+0x10] [100];[rsi+0x10]
[71];[rsi+0x10] [62];[rsi+0x10] [5];[rsi+0x10] [85];[rsi+0x10] [34];[rsi+0x10] [86];[rsi+0x18]
[12];[rsi+0x10] [14];[rsi+0x10] [22];[rsi+0x10] [101];[rsi+0x10] [37];[rsi+0x10]
[93];[rsi+0x10] [55];[rsi+0x10] [32];[rsi+0x10] [68]
```

On peut voir sur la trace précédente 2 écritures sur la pile des différents bits d'entrée mélangés. L'exécution symbolique d'autres parties du code tend à montrer que chaque fonction à son propre « mélange » de bits d'entrée. On peut également s'attendre à ce que les sorties soient mélangées de la sorte.

Il n'est dès lors pas possible de simplement étudier les fonctions en boîte noire, les entrées et les sorties n'étant pas directement interprétables.

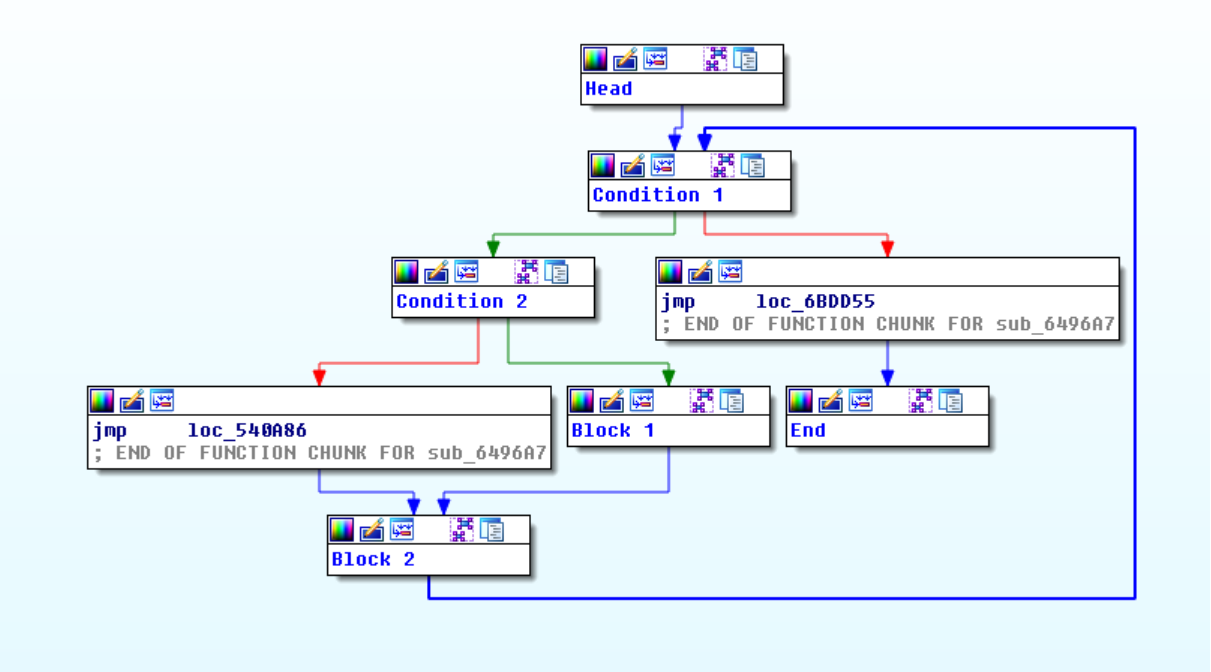
Pour étudier le code, il serait plus simple de se débarrasser de la contrainte du ring de processus. Il suffit pour cela de « ripper » la section .text du PE pour la réutiliser tranquillement dans un ELF sous Linux ☺ L'utilisation de ptrace() permet d'intercepter les appels aux « int 3 » pour rediriger le flux d'exécution au bon endroit selon le tableau de fonctions que l'on aura également préalablement rippé (à l'aide d'une injection de DLL). Nous sommes alors en mesure de facilement examiner les entrées de chaque fonction. Pour pouvoir en examiner les sorties, il suffit de corrompre l'adresse de retour stockée sur la pile, et de la restaurer lorsque l'on aura intercepté le SIGSEGV avec ptrace().

Ce bout de code que l'on qualifiera volontiers de dégueulasse a pour avantage de permettre d'instrumenter le code obfusqué à notre guise : nous pouvons dérouler l'intégralité de la vérification de la clé en observant les différents appels de fonctions, ou choisir de n'exécuter qu'une fonction en modifiant les entrées pour observer les sorties.

Une première étape sera de représenter l'arbre des appels, pour voir si une forme connue s'en dégage (cf. fichier « forme » de l'archive).

On observe alors un certain schéma, avec de très nombreux appels à la fonction 0x6496a7 (fonction_1), qui est une fonction finale (i.e. sans appels de fonctions).

Cette fonction est l'une des rares comportant des sauts conditionnels et une boucle. Sa structure est la suivante :



Pour chacune des conditions, un bit est testé via l'instruction « test ax, 0x8000 ».

L'exécution symbolique de la fonction va nous permettre, une fois de plus, de voir quels bits sont utilisés dans les conditions (sortie modifié pour une meilleure lecture) :

```
TESTED rax PMAXUB(PMAXUB(PMAXUB(PMAXUB(PMAXUB(PMAXUB(OR(PMAXUB(PMAXUB(OR([rbp+0x10]
[6],[rbp+0x10] [14]),[rbp+0x10] [18]),[rbp+0x10] [0]),[rbp+0x10] [4]),[rbp+0x10]
[21]),[rbp+0x10] [2]),[rbp+0x10] [11]),[rbp+0x10] [10]),[rbp+0x10] [19]),[rbp+0x10] [15])
TESTED rax [rbp+0x10] [6]
```

```
TESTED rax PMAXUB(PMAXUB(PMAXUB(PMAXUB(PMAXUB(PMAXUB(OR(PMAXUB(PMAXUB(OR([rbp+0x10]
[14],[rbp+0x10] [18]),[rbp+0x10] [0]),[rbp+0x10] [4]),[rbp+0x10] [21]),[rbp+0x10]
[2]),[rbp+0x10] [11]),[rbp+0x10] [10]),[rbp+0x10] [19]),[rbp+0x10] [15]),0)
TESTED rax [rbp+0x10] [14]

TESTED rax PMAXUB(PMAXUB(PMAXUB(PMAXUB(PMAXUB(OR(PMAXUB(PMAXUB(OR([rbp+0x10] [18],[rbp+0x10]
[0]),[rbp+0x10] [4]),[rbp+0x10] [21]),[rbp+0x10] [2]),[rbp+0x10] [11]),[rbp+0x10]
[10]),[rbp+0x10] [19]),[rbp+0x10] [15]),0)
TESTED rax [rbp+0x10] [18]

TESTED rax PMAXUB(PMAXUB(PMAXUB(PMAXUB(OR(PMAXUB(PMAXUB(OR([rbp+0x10] [0],[rbp+0x10]
[4]),[rbp+0x10] [21]),[rbp+0x10] [2]),[rbp+0x10] [11]),[rbp+0x10] [10]),[rbp+0x10]
[19]),[rbp+0x10] [15]),0)
TESTED rax [rbp+0x10] [0]

TESTED rax PMAXUB(PMAXUB(PMAXUB(PMAXUB(OR(PMAXUB(PMAXUB(OR([rbp+0x10] [4],[rbp+0x10]
[21]),[rbp+0x10] [2]),[rbp+0x10] [11]),[rbp+0x10] [10]),[rbp+0x10] [19]),[rbp+0x10]
[15]),0),0)
TESTED rax [rbp+0x10] [4]

TESTED rax PMAXUB(PMAXUB(PMAXUB(OR(PMAXUB(PMAXUB(OR([rbp+0x10] [21],[rbp+0x10] [2]),[rbp+0x10]
[11]),[rbp+0x10] [10]),[rbp+0x10] [19]),[rbp+0x10] [15]),0),0)
TESTED rax [rbp+0x10] [21]

TESTED rax PMAXUB(PMAXUB(OR(PMAXUB(PMAXUB(OR([rbp+0x10] [2],[rbp+0x10] [11]),[rbp+0x10]
[10]),[rbp+0x10] [19]),[rbp+0x10] [15]),0),0)
TESTED rax [rbp+0x10] [2]

TESTED rax PMAXUB(PMAXUB(PMAXUB(PMAXUB(OR([rbp+0x10] [11],[rbp+0x10] [10]),[rbp+0x10]
[19]),[rbp+0x10] [15]),0),0)
TESTED rax [rbp+0x10] [11]

TESTED rax PMAXUB(PMAXUB(OR([rbp+0x10] [10],[rbp+0x10] [19]),[rbp+0x10] [15]),0)
TESTED rax [rbp+0x10] [10]

TESTED rax PMAXUB(PMAXUB(OR([rbp+0x10] [19],[rbp+0x10] [15]),0),0)
TESTED rax [rbp+0x10] [19]

TESTED rax PMAXUB(PMAXUB([rbp+0x10] [15],0),0)
TESTED rax [rbp+0x10] [15]
```

Malgré la sortie non-intuitive de l'outil, on connait maintenant le but des deux tests :

- Le premier vérifie qu'un des arguments ne vaut pas 0 ;
- Le second vérifie les bits de cet argument 1 à 1.

On peut voir alors que cet argument contient 11 bits, représentés par les bits 6, 14, 18, 0, 4, 21, 2, 11, 10, 19 et 15 de l'entrée.

L'exécution symbolique montre également que seuls 22 bits de l'entrée sont utilisés.

La structure de la fonction rappelle une multiplication ou une exponentiation rapide. Au vu du nombre d'instructions, nous partons de l'hypothèse qu'il s'agit d'une multiplication. Connaissant les bits représentant le multiplicateur, nous pouvons simplement mettre tous les bits de l'autre facteur à

1, et faire varier le multiplicateur tout en observant la sortie. Ainsi, si l'on fait passer le multiplicateur de 1 à 2, on pourra alors identifier le bit 0 de la sortie comme étant celui passé de 1 à 0, et le bit 11 comme celui passé de 0 à 1. En répétant l'opération jusqu'à 2^11, on identifie tous les bits de la sortie.
Une fois la sortie connue, on en déduit les bits du facteur d'entrée, et sommes en mesure de « tracer » les entrées/sorties de la fonction_1.

Type	Bits
Entrée 1	7, 8, 16, 9, 12, 17, 5, 13, 20, 3, 1
Entrée 2 (multiplicateur)	6, 14, 18, 0, 4, 21, 2, 11, 10, 19, 15
Sortie	7, 16, 6, 1, 21, 18, 5, 12, 15, 13, 10, 8, 20, 14, 9, 19, 3, 0, 4, 11, 2, 17

La suite consiste à « remonter » dans l'arbre d'appel des fonctions pour tenter d'en comprendre le sens. La fonction_1 est appelée 3 fois par la fonction 0x407d02 (fonction_2). Etant en mesure d'observer les entrées/sorties de la fonction_1, on constate empiriquement que les trois appels fonctionnent comme suit :

- Mul(A,X)
- Mul(B,Y)
- Mul(A+B,X+Y)

A, B, X et Y étant des nombres représentés sur 10 bits. Cette structure d'appel semble permettre la multiplication de 2 nombres de 20 bits « AB » et « XY », en passant tour à tour les 10 bits de poids faible, les 10 bits de poids fort, et la somme des deux à la fonction_1, dans le but de combiner les résultats pour obtenir AB*XY.
Connaissant les entrées de fonction_1, on retrouve les entrées de fonction_2, puis on calcule alors les sorties de fonction_2 selon l'hypothèse formulée. Une fois cela fait, on peut tracer les entrées/sorties de fonction_2, et vérifier empiriquement l'hypothèse ! Fonction_2 permet donc la multiplication de 2 nombres de 20 bits.

Si l'on continue à remonter, on constate que fonction_2 est appelée 3 fois par la fonction 0x466a23 (fonction_3), et on retrouve les mêmes types d'appels que pour fonction_2. On

prend alors l'hypothèse qu'il s'agit du même mécanisme, et on calcule entrées et sorties pour vérifier empiriquement. L'hypothèse se validant, on sait alors que fonction_3 permet de multiplier 2 nombres de 40 bits.

On continue à remonter …
On arrive sur la fonction 0x4a5c37 (fonction_4), appelant 5 fois fonction_3 ! La structure des appels est la suivante, toujours définie empiriquement :

- Mul(A, X)
- Mul(A + B*2 + C*4, X + Y*2 + Z*4)
- Mul(A + B*3 + C*9, X + Y*3 + Z*9)
- Mul(C, Z)
- Mul(A + B + C, X + Y + Z)

Il semble s'agir cette fois de multiplier « ABC » par « XYZ ». En observant les entrées/sorties de fonction_3, on retrouve l'ordre des 204 bits d'entrée, puis celui des bits de sortie, et validons l'hypothèse.

Nous avons alors identifié une fonction permettant de multiplier 2 nombres de 102 bits, ce qui est la taille de la clé prise en entrée des fonctions SSE !

En continuant à remonter, on arrive sur la fonction 0x4ef832 (fonction_5), faisant d'abord appel à fonction_4, puis à la fonction 0x58989a (fonction_6), appelant également fonction_4 avec un multiplicateur fixe de 0x374255b3818bbba1418ab90ad5 (produit de 2 nombres premiers, 1982112667691789 et 2208793993743593). Quelques observations nous montrent que fonction_6 multiplie son entrée par une constante (0x3df98dab28e68aba94e6422183) avant de passer le résultat à fonction_4. Ces opérations font penser à un modulo (ce qui rappellerait également le texte de l'épreuve !).
En effet, A mod K ⇔ A-(A/K)*K ⇔ A-((A*C)>>B)*K.
Fonction_6 serait alors un modulo 0x374255b3818bbba1418ab90ad5 (que nous nommerons X).

Nous sommes alors remontés suffisamment haut dans l'arbre des appels pour tenter d'avoir une vue globale de l'algorithme. Son observation fait penser à une élévation modulaire à une petite puissance. Afin de déterminer la puissance utilisée (et ne connaissant que l'ordre des bits d'entrée), on peut prendre une hypothèse sur l'exposant, entrer une valeur aléatoire K en

entrée, et vérifier si le nombre de bits à « 1 » en sortie est cohérent avec K^e mod X.

Ce test n’est pas concluant pour un exposant 3 (bye bye RSA ?), mais fonctionne pour un exposant 2 ! On peut alors calculer les bits de sortie selon cette hypothèse, et en faire la vérification empirique, qui colle parfaitement !

Nous sommes donc en présence d’un simple carré modulaire, codé sur plus de 3Mo d’instructions SSE… (merci les gars !).

La seule étape restante est de comprendre la condition finale (le fameux octet 0 de la zone partagée positionné à 1).
Pour cela, nous pouvons à nouveau faire une exécution symbolique du code commençant à l’adresse 0x54cb64, et observer la valeur du bit 0 de la zone mémoire partagée. Celui est représenté par une équation mettant en jeu l’ensemble des bits de la sortie : il s’agit du NOT d’un ensemble de OR sur tous les bits, certains subissant un NOT.

```
NOT(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(
OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(O
R(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR(OR
(OR(OR(OR(OR(OR(OR(OR(OR([rsp+0x10] [23],NOT([rsp+0x10] [16])),NOT([rsp+0x18]
[17])),[rsp+0x10] [49]),NOT([rsp+0x10] [3])),[rsp+0x10] [28]),[rsp+0x10] [36]),[rsp+0x10]
[52]),[rsp+0x10] [62]),[rsp+0x10] [54]),[rsp+0x18] [13]),[rsp+0x18] [1]),[rsp+0x10]
[9]),NOT([rsp+0x10] [24])),[rsp+0x10] [35]),[rsp+0x10] [17]),[rsp+0x10] [33]),[rsp+0x18]
[26]),NOT([rsp+0x10] [61])),NOT([rsp+0x18] [19])),NOT([rsp+0x18] [35])),[rsp+0x18]
[4]),[rsp+0x18] [8]),[rsp+0x10] [34]),NOT([rsp+0x10] [51])),NOT([rsp+0x10]
[31])),NOT([rsp+0x18] [30])),[rsp+0x18] [21]),NOT([rsp+0x10] [11])),[rsp+0x18]
[14]),NOT([rsp+0x10] [14])),[rsp+0x10] [45]),NOT([rsp+0x10] [39])),[rsp+0x10]
[59]),NOT([rsp+0x10] [18])),[rsp+0x18] [23]),NOT([rsp+0x10] [46])),NOT([rsp+0x18]
[34])),NOT([rsp+0x18] [25])),[rsp+0x10] [38]),NOT([rsp+0x18] [3])),NOT([rsp+0x10]
[26])),[rsp+0x10] [56]),[rsp+0x10] [32]),NOT([rsp+0x18] [29])),NOT([rsp+0x10] [8])),[rsp+0x18]
[36]),[rsp+0x10] [19]),NOT([rsp+0x10] [57])),[rsp+0x10] [37]),[rsp+0x10] [41]),[rsp+0x10]
[7]),[rsp+0x10] [5]),[rsp+0x18] [9]),[rsp+0x10] [0]),[rsp+0x10] [20]),[rsp+0x18]
[11]),[rsp+0x10] [1]),[rsp+0x10] [27]),[rsp+0x10] [42]),[rsp+0x18] [15]),[rsp+0x10]
[21]),[rsp+0x10] [12]),[rsp+0x10] [15]),[rsp+0x18] [37]),[rsp+0x18] [20]),[rsp+0x10]
[13]),[rsp+0x18] [32]),[rsp+0x10] [6]),[rsp+0x18] [18]),[rsp+0x10] [10]),[rsp+0x10]
[48]),[rsp+0x18] [22]),[rsp+0x10] [25]),[rsp+0x18] [0]),[rsp+0x18] [33]),[rsp+0x18]
[16]),[rsp+0x18] [2]),[rsp+0x10] [55]),[rsp+0x10] [47]),[rsp+0x18] [6]),[rsp+0x18]
[7]),[rsp+0x10] [44]),[rsp+0x18] [28]),[rsp+0x10] [2]),[rsp+0x10] [58]),[rsp+0x10]
[4]),[rsp+0x10] [53]),[rsp+0x18] [31]),[rsp+0x18] [12]),[rsp+0x18] [5]),[rsp+0x10]
[40]),[rsp+0x10] [30]),[rsp+0x10] [63]),[rsp+0x18] [10]),[rsp+0x10] [29]),[rsp+0x10]
[22]),[rsp+0x18] [24]),[rsp+0x10] [43]),[rsp+0x10] [50]),[rsp+0x18] [27]),[rsp+0x10] [60]))
```

On comprend alors que les bits subissant un NOT doivent être à 1. La valeur finale doit alors être **0x13375571c2016** (soit LEETSSTIC2016 en leet-speak).

Reste à résoudre l’équation :
X^2 mod 0x374255b3818bbba1418ab90ad5 == 0x13375571c2016

La bibliothèque libnum permet de résoudre le problème en Python :

```
>>> import libnum
>>> for k in libnum.sqrtmod(0x13375571c2016, {1982112667691789:1, 2208793993743593:1}):
...  print hex(k)
...
325304773f41a3b50bd8a2d545
e3aee80814502fd956f9123a0
290767330046b8a3ac1b27e735
4ef513c424a17ec35b2163590
```

Il y a 4 solutions. Il ne reste qu'à trouver la clé attendue en essayant les 4 dans le RPG, sachant que la clé est inversée.

La bonne clé est alors :
35e7271baca3b84600336707290000000

Chapitre 5 – The End

De retour dans un monde plus normal, notre héros accomplit enfin sa quête…

Le monarque siégeant en haut de la pièce lui remit l'ultime parchemin, contenant la formule suivante :

J01p1 y'4qe3553 z41y :

8Y6d5j9Vy88HUGHfGSSKsJvqA@ffgvp.bet

Il s'agissait là de l'ancien dialecte du 13e Rang des Orcs Terrifiants (ROT13), dont la traduction dans une langue plus moderne serait :

V01c1 l'4dr3553 m41l :

8L6q5w9Il88UHTUsTFXfWidN@sstic.org