Solution du challenge SSTIC 2021

Mathieu Dechambe

Sommaire

1 Introduction

Cette année, j'ai participé pour la première fois au challenge du SSTIC. Je ne savais pas du tout dans quoi j'allais m'embarquer avant de commencer et ce fut une excellente surprise ! La difficulté est au rendez-vous mais la qualité des problèmes à résoudre est vraiment excellente d'où la satisfaction d'en arriver à bout. J'en ressort avec un bagage de connaissances que j'aurais probablement mis beaucoup plus de temps à acquérir.

Le challenge démarre avec cet énoncé :

> Suite à la situation sanitaire mondiale, le SSTIC se déroule pour la deuxième année consécutive en ligne. Une des conséquences principales est que cette année encore, aucun billet ne sera vendu. Devant cette impossibilité à s'enrichir grassement sur le travail de la communauté infosec et voulant faire l'acquisition d'une nouvelle Mercedes et de 100g de poudre, le Comité d'Organisation a décidé de réagir ! Une solution de DRM a été développée spécifiquement pour protéger les vidéos des présentations du SSTIC qui seront désormais payantes.
> En tant que membre de la communauté infosec, impossible de laisser passer ça. Il faut absolument analyser cette solution de DRM afin de trouver un moyen de récupérer les vidéos protégées et de les partager librement pour diffuser la connaissance au plus grand nombre (ou donner les détails au CO du SSTIC contre un gros bounty).
> Heureusement, il a été possible d'infiltrer le CO et de récupérer une capture effectuée lors d'un transfert de données sur une clé USB. Avec un peu de chance, elle devrait permettre de mettre la main sur la solution de DRM et l'analyser.
> Bon courage!

L'objectif sera donc d'analyser une solution de DRM (Management des Droits Numériques en français) afin de récupérer une vidéo particulière protégée par ce système dans laquelle se trouve une adresse email qui permettra de terminer le challenge.

2 Etape 1

2.1 Confirmer les informations

La première étape commence par l'analyse d'un fichier de type **pcapng** qui contient la capture d'un transfert de données entre un ordinateur et une clé USB.

```
> file usb_capture_C0.pcapng
usb_capture_C0.pcapng: pcapng capture file - version 1.0
```

Mon premier réflexe a été d'ouvrir le fichier avec **Wireshark**[1] qui permet la réalisation et l'analyse de ce genre de capture. Avant de commencer à analyser le fichier directement qui comporte un certain nombre de paquets (1441), j'ai regardé les propriétés du fichier de capture grâce à Wireshark. Cela m'a permis de confirmer qu'il s'agissait bien d'un transfert usb : Le champ type de lien indique *USB packets with Linux header and padding* (on le voit aussi au champ protocole des paquets qui est soit **USB**, **USBMS** ou **USBHUB**). J'apprends aussi que la capture a été réalisée avec **Dumpcap** [2] qui est un utilitaire permettant de capturer du flux réseau.

Le paquet 35 de la capture permet d'identifier le type d'appareil avec lequel l'ordinateur est connecté.

```
DEVICE DESCRIPTOR
   bLength: 18
   bDescriptorType: 0x01 (DEVICE)
   bcdUSB: 0x0320
   bDeviceClass: Device (0x00)
   bDeviceSubClass: 0
   bDeviceProtocol: 0 (Use class code info from Interface Descriptors)
   bMaxPacketSize0: 9
   idVendor: Kingston Technology (0x0951)
   idProduct: DataTraveler 100 G3/G4/SE9 G2/50 (0x1666)
   bcdDevice: 0x0001
   iManufacturer: 1
   iProduct: 2
   iSerialNumber: 3
   bNumConfigurations: 1
```

Figure 1: Le paquet GET DESCRIPTOR Response DEVICE

Nous sommes donc en présence d'un périphérique de stockage. Les informations données dans l'énoncé se confirment.

2.2 Extraire les données

La difficulté est maintenant de déterminer quel type de données ont été transmises et de les extraires du fichier de capture.

Avant d'aller plus loin, j'ai utilisé **binwalk**[3] sur le fichier de capture qui permet entre autres de repérer les headers de types de fichiers connus dans d'autres fichiers.

[1] https://www.wireshark.org/
[2] https://manpages.ubuntu.com/manpages/xenial/man1/dumpcap.1.html
[3] https://github.com/ReFirmLabs/binwalk

```
> binwalk usb_capture_C0.pcapng

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------
3083492       0x2F0CE4        7-zip archive data, version 0.4
```

Binwalk détecte le header d'un fichier 7-zip à l'offset 3083492 du fichier de capture. Un fichier 7-zip a un header de 6 octets : **"37 7A BC AF 27 1C"** ce qui rend la probabilité de le trouver par hasard dans un autre fichier assez faible. On peut additionnellement utiliser binwalk avec l'option -E pour tracer l'entropie du fichier :

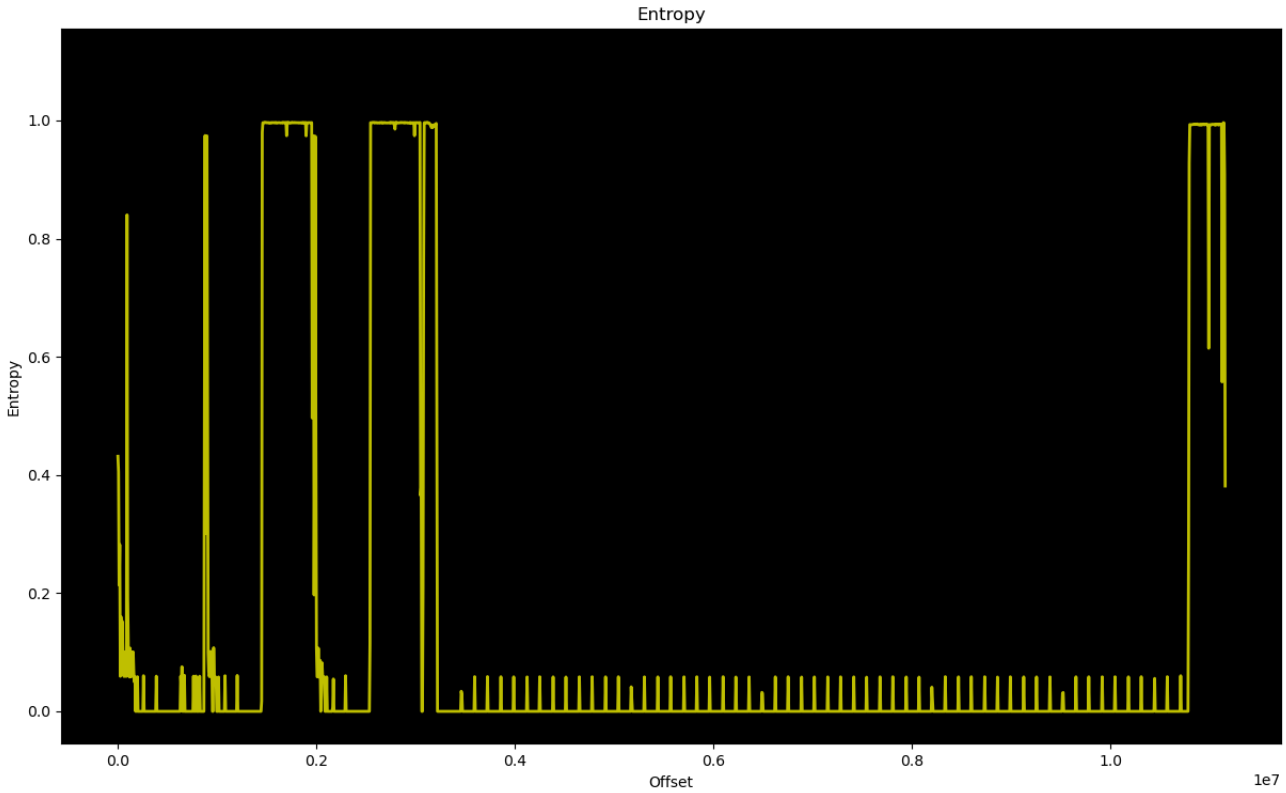

Figure 2: Tracé de l'entropie du fichier

Les pics qui montent aux alentours de 1.0 d'entropie sont caractéristiques de données chiffrées ou **compressées**. L'hypothèse d'un transfert de fichier 7-zip dans la capture semble se confirmer.

Pour continuer à explorer cette hypothèse, on va utiliser quelques lignes de python pour identifier à quel paquet démarre le transfert du fichier 7-zip. Lors d'un transfert de données via USB, la charge utile (les données du fichier effectivement transmis) est située à un offset de 64 octets par rapport au début du paquet, dans le champ **SCSI Payload**.

```python
#!/usr/bin/env python3
from scapy.all import *

packets = rdpcap("usb_capture_C0.pcapng")

for i in range(len(packets)):
    # On regarde si l'un des paquets a une charge utile SCSI
    # qui commence par le header d'un fichier 7-zip
    if b"7z\xbc\xaf\x27\x1c" == packets[i].load[0x40:0x46]:
        print("Found beginning of 7z file at packet %d" % (i + 1))
```

```
> ./extract.py
Found beginning of 7z file at packet 942
```

En regardant dans Wireshark le paquet numéro 942, on se rend compte que le paquet est à destination du périphérique de stockage et en provenance de l'hôte.

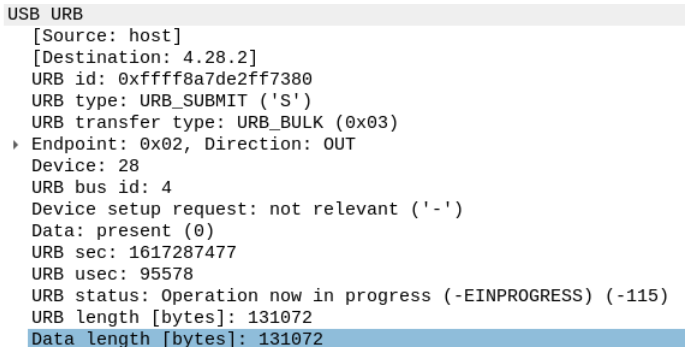

Figure 3: Une partie des métadonnées du paquet 942

Malheureusement, extraire les données ce paquet seul ne suffit pas. Le fichier 7-zip n'est pas complet, il faut donc retrouver la suite dans le reste du fichier de capture.

Un transfert de données via USB se déroule de la façon suivante (en ne regardant que le protocole USBMS, USB Mass Storage):

- L'initiateur du transfert donne l'adresse du bloc logique à laquelle écrire ainsi que la longueur des données qu'il va transmettre : **SCSI Write**
- Les données sont ensuite effectivement envoyées dans un paquet de grande taille : **SCSI Data Out**
- A la fin du transfert, le récepteur renvoie un acquittement : **SCSI Response**

Pour récupérer la suite du fichier 7-zip, on va donc regarder l'adresse du bloc logique (**LBA**, qui permet d'adresser les secteurs d'un disque dur, en général d'une taille de 512 octets) à laquelle l'hôte demande à écrire dans le Write ainsi que le nombre de blocs pour calculer l'adresse du prochain Write.

Si on regarde la figure 3 on voit que le nombre d'octets envoyés dans le Data Out est de 131072 / 512 = 256 blocs écrits ce qu'on confirme en regardant le Write associé au Data Out.

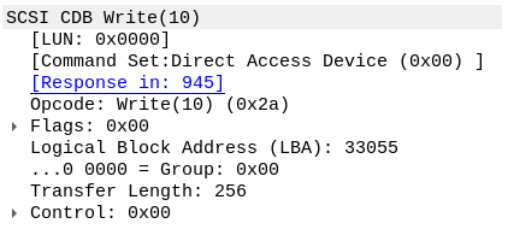

Figure 4: Le paquet SCSI Write correspondant au Data Out de la figure 3

On peut observer la longueur du transfert : 256 blocs et l'adresse du bloc à laquelle écrire : 33055. On peut en déduire l'adresse de la prochaine demande d'écriture pour ce fichier : $33055 + 256 = 33311$.

Plus qu'à itérer sur le reste des paquets pour récupérer les données qui correspondent à la suite du transfert.

```python
#!/usr/bin/env python3
from scapy.all import *
import struct

def u32(a):
    return struct.unpack(">I", a)[0]

def u16(a):
    return struct.unpack(">H", a)[0]

packet_reader = RawPcapNgReader("usb_capture_C0.pcapng")
packets = []
for i in range(1441):
    packets.append(packet_reader.read_packet(size=5000000)[0])

nblocks = u16(packets[939][86:88])
lba = u32(packets[939][81:85])
next_lba = lba + nblocks

data = packets[941][64:]

for i in range(942, len(packets)):
    if len(packets[i]) < 88 : continue
    current_lba = u32(packets[i][81:85])
    if current_lba == next_lba:
        print("Found next packet %d" % (i+2))
        nblocks = u16(packets[i][86:88])
        next_lba = current_lba + nblocks
        data += packets[i+2][64:]

open("out", "wb").write(data)
```

La fonction rdpcap de scapy tronque les paquets dont la longueur est supérieure à 65536 octets, ce qui est le cas des paquets Data Out qui contiennent de gros volumes de données. J'ai choisi de passer par la fonction RawPcapNgReader pour pallier à ce problème.

On se retrouve donc avec un beau fichier 7-zip reconstitué qu'on peut maintenant extraire sans problème! Dans le dossier extrait on trouve un binaire Windows **A..Mazing.exe**, un fichier Readme, un fichier env.txt et le premier flag dans une image :

SSTIC{c426baf3470c7ffbea05a5320d1d2b74}

3 Etape 2

3.1 Trouver le bug

Le fichier Readme contient le message suivant :

> Hey Trou,
> Do you remember the discussion we had last year at the secret SSTIC party? We planned to create the next SSTIC challenge to prove that we are still skilled enough to be trusted by the community.
> I attached the alpha version of my amazing challenge based on maze solving. You can play with it in order to hunt some remaining bugs. It's hosted on my workstation at home, you can reach it at challenge2021.sstic.org:4577.
> I've written in the env.txt file all the information about the remote configuration if needed.
> Have Fun,

et env.txt :

```
OS Name : Microsoft windows 10 Pro
Version : 20H2 => 10.0.19042 Build 19042
Seems facultative but updated until 22/03/2021 :
Last installed KBs:
Quality Updates : KB500802, KB 46031319, KB4023057 /
Other Updates : KB5001649, KB4589212
Network : No outbound connections
Process limits per jail: 2
Memory Allocation limit per jail : 100Mb
Time Limit : 2 min cpu user time
```

D'après les informations dans le fichier Readme, cette étape consistera donc à analyser le binaire Windows trouvé dans l'étape 1 pour y trouver des bugs.

Pour commencer à me familiariser avec le programme à analyser, je l'ai lancé dans une machine virtuelle Windows 10.

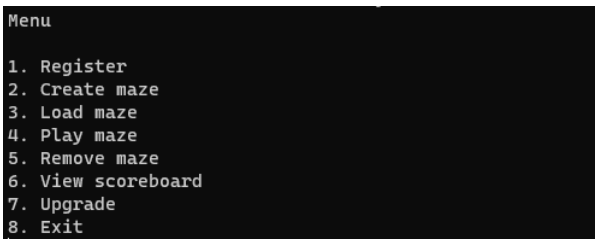

Figure 5: Menu de A..Mazing.exe

C'est un jeu dans le terminal qui permet de générer des labyrinthes et de les résoudre. Il est accessible sur une machine distante à l'adresse donnée dans le fichier Readme. Le but sera donc d'exploiter un ou plusieurs bugs dans ce programme pour gagner un accès à la machine distante et accéder à la suite du challenge.

Après avoir manipulé un peu le programme, on se rend compte de plusieurs choses intéressantes :

- Il faut s'enregistrer en entrant un nom avant de pouvoir effectuer n'importe quelle action
- A chaque labyrinthe est associé un tableau de scores indiquant les noms des joueurs qui ont terminé le labyrinthe et leur score (le nombre de coups mis pour arriver à la fin)
- On peut créer 3 types de labyrinthes différents :
  - Labyrinthe classique qu'on appellera **Type 1** dans la suite
  - Labyrinthe avec plusieurs passages (**Type 2**)
  - Labyrinthe avec plusieurs passages et des pièges (**Type 3**)
- Il est possible de sauvegarder un labyrinthe (**.maze**) et son fichier de scores associé (**.rank**) sur le système de fichier
- Il est possible de charger un labyrinthe et son fichier de scores associé depuis le système de fichier

Le dernier point est particulièrement intéressant car il implique que le programme implémente une logique de **parsing de fichier**, c'est à dire lire les données sur le disque et les charger en mémoire dans des structures de données utilisables par le programme. Ce genre de fonctionnalité est souvent source de bugs[4].

De plus, la fonction de chargement d'un labyrinthe est très permissive sur la façon d'identifier le fichier à charger. Il est possible de donner l'identifiant du fichier ou son nom avec ou sans extension. Ma première idée a été d'essayer de charger un fichier qui n'a pas la bonne extension et qui ne représente pas du tout un labyrinthe.

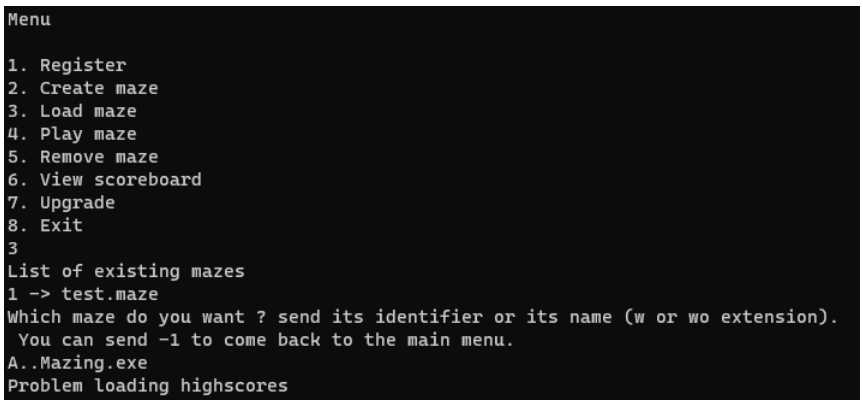

Figure 6: Chargement d'un faux labyrinthe

Comme on peut le voir ici, il est possible de charger le binaire lui-même en tant que labyrinthe. Une erreur est affichée par rapport au chargement du fichier de scores (car le fichier A..Mazing.rank n'existe pas) mais il est possible de lancer une partie sur le fichier chargé, ce qui fait planter le programme.

On peut ainsi observer un premier bug dans ce programme que l'on va essayer de rendre exploitable.

[4]Voir AFL (American Fuzzy Lop), un fuzzer particulièrement efficace dans la recherche de bugs au niveau du parsing de fichiers

3.2 Comprendre le programme

Pour mieux comprendre le bug et déterminer s'il est exploitable, il va falloir faire de la rétro-ingénierie sur le programme pour comprendre comment sont représenté les labyrinthes et les tableau des scores :

- En mémoire
- Sur le disque

Pour ce processus de rétro-ingénierie, j'ai utilisé **IDA** et son décompilateur qui s'est avéré être très utile. Ma méthodologie a été la suivante : tout d'abord trouver la fonction qui affiche le menu et qui demande à l'utilisateur de choisir l'action qu'il veut effectuer.

```
while ( 1 )
{
  printf(
    "Menu\n"
    "\n"
    "1. Register\n"
    "2. Create maze\n"
    "3. Load maze\n"
    "4. Play maze\n"
    "5. Remove maze\n"
    "6. View scoreboard\n"
    "7. Upgrade\n"
    "8. Exit\n");
  if ( get_number((__int64)user_choice) )
    break;
  switch ( user_choice[0] )
  {
    case 1:
      if ( register(&player) )
      {
        registered = 1;
      }
      else
      {
        printf("Problem while registering. Please retry\n");
        registered = 0;
      }
      break;
```

Figure 7: Affichage du menu et début du switch

J'ai pu déterminer grâce au contexte d'utilisation des différentes fonctions quel était leur rôle et quel type de données étaient manipulées. Par exemple, on voit ici le début du switch qui détermine ce que doit faire le programme en fonction du choix de l'utilisateur. L'option 1 permettant de s'enregistrer comme affiché dans le menu, il est facile de déterminer que la fonction appelée dans le case 1 du switch est la fonction qui permet au joueur de s'enregistrer. Les messages d'erreurs affichés en cas d'échec des fonctions sont aussi particulièrement utiles durant la rétro-ingénierie (comme *Problem while registering. Please retry* ici).

Un première itération m'a permis de déterminer l'utilité de chaque fonction en fonction du contexte et de leur donner un nom. Ensuite, il est plus facile d'analyser en détail le fonctionnement précis d'une fonction si besoin.

De fil en aiguille, j'ai pu déterminer la forme des structures de données manipulées par le programme que j'ai retranscrit en C de la façon suivante :

```
// un piège
struct trap {
    long score;     // valeur du piège
    short pos;      // position du piège dans le labyrinthe
    char symbol;    // représentation du piège dans le labyrinthe
    int active;     // le piège a-t-il été déclenché ?
};

// tous les pièges
struct traps {
    char number_traps;
    struct trap traps[256];
};

// un score
struct score_entry {
    long score;
    char player_name[128];
};

// tous les scores
struct scores {
    char number_players;
    struct score_entry entries[128];
};

// un labyrinthe
struct maze {
    char width;         // largeur du labyrinthe
    char height;        // hauteur du labyrinthe
    char type;          // type du labyrinthe
    char name[128];     // nom du labyrinthe
    char creator[128];  // nom du créateur du labyrinthe
    union {
        // pièges, pour le labyrinthe de type 3
        struct traps maze_traps;
        // représentation ascii du labyrinthe pour le type 1
        char * type1_ascii_rep;
    };
    // représentation ascii du labyrinthe pour les types 2 et 3
    char * type23_ascii_rep;
    char nwalls_to_remove;
    struct scores player_scores;  // tableau des scores
};
```

On peut noter une chose très intéressante dans la structure du labyrinthe : il y a un champ `type` qui est utilisé pour déterminer comment interpréter le reste de la structure dans d'autres fonctions, notamment celle qui affiche le labyrinthe lorsque

que le joueur est en train de jouer.

```
void print_maze(struct maze * maze, int width) {
    if (maze->type == 3 || maze->type == 2) {
        print_with_traps(maze, width);
    } else {
        print_without_traps(maze, width);
    }
}
```

print_without_traps() utilise l'adresse contenue dans `maze->type1_ascii_rep` pour afficher le labyrinthe tandis que **print_with_traps()** utilise l'adresse contenue dans `maze->type23_ascii_rep`.

Cette logique de type est importante pour la suite.

Pour comprendre les formats de fichier utilisés pour stocker labyrinthes et tableau de scores, j'ai regardé les fonctions utilisées pour sauvegarder sur le disque.

La rétro-ingénierie de ces fonctions est très facile une fois les structures entrées dans **IDA**. Il s'agit simplement d'une suite d'appel à la fonction **WriteFile()** avec différents champs de la structure représentant le labyrinthe en arguments.

On peut ainsi déterminer les formats de fichier:

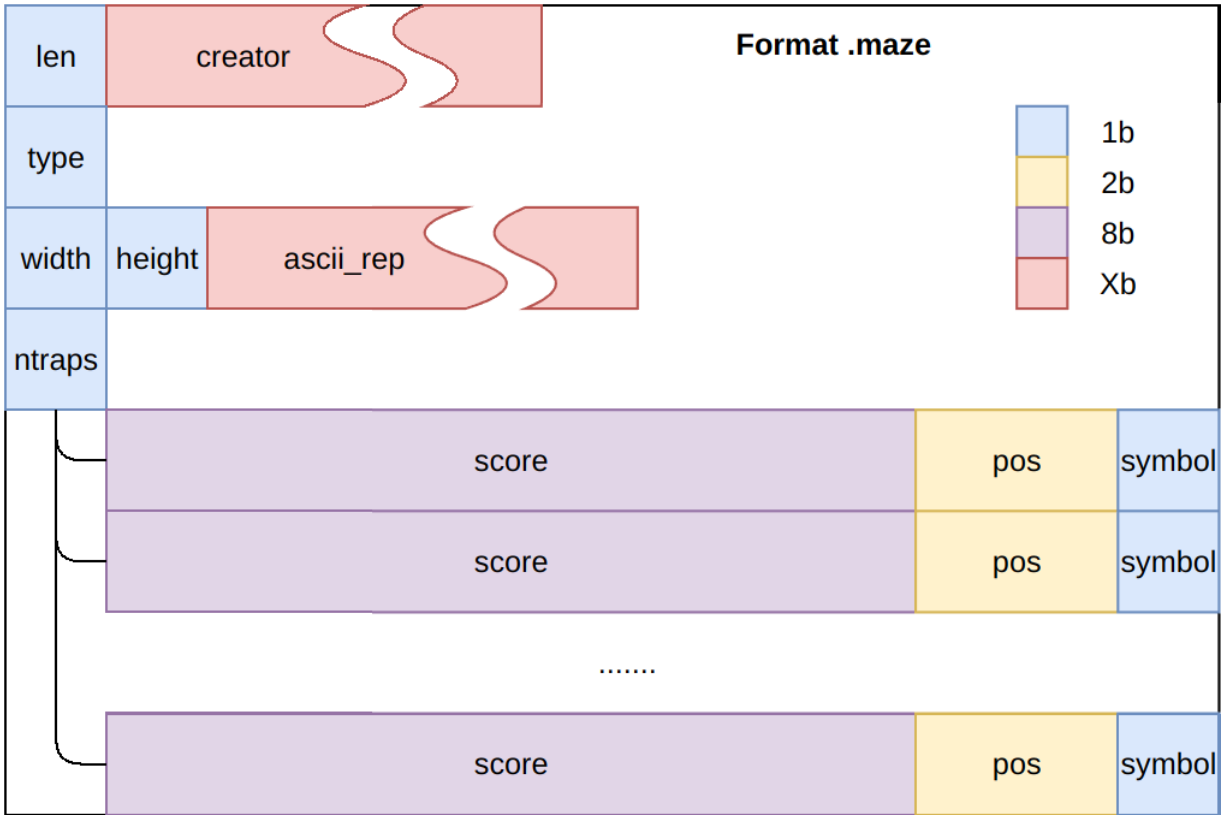

Figure 8: Format de fichier maze

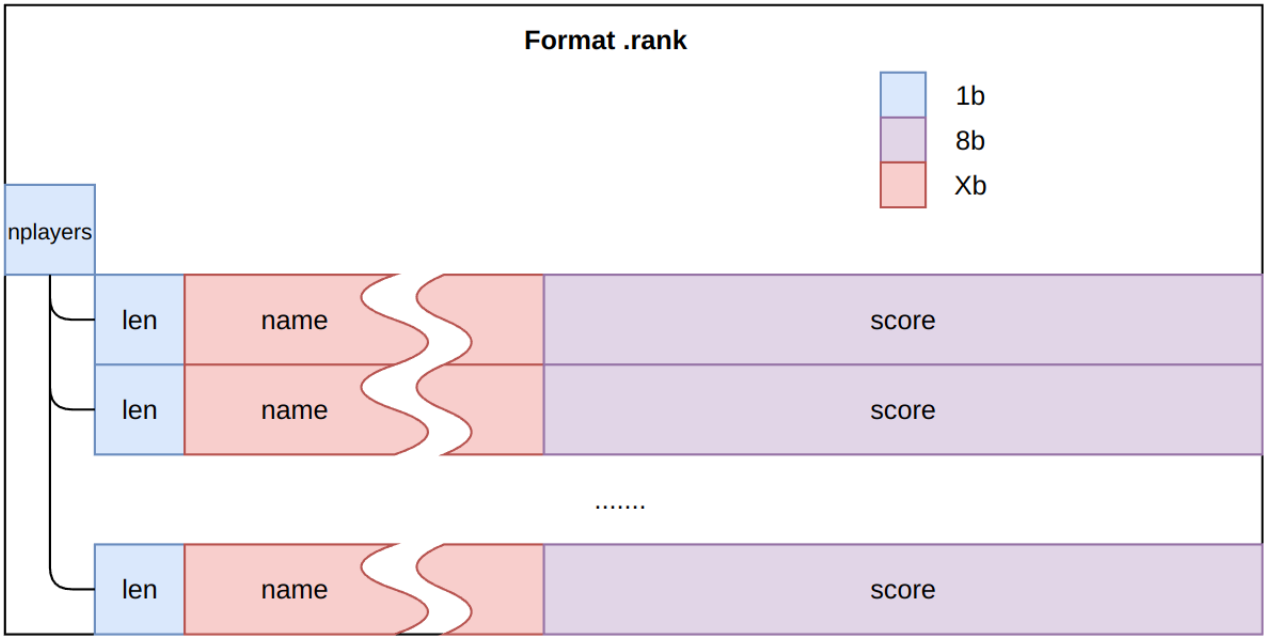

Figure 9: Format de fichier rank

Les figures se lisent de haut en bas et de gauche à droite. Les champs `len` indiquent la longueur en octets du champ suivant. Le champ `ascii_rep` dans le format maze a une longueur de `width * height` octets.

La fonction qui permet de charger un fichier maze est assez simple à comprendre. Elle enchaine des **read** sur le fichier que l'on demande à charger. Il n'y a quasiment pas de vérifications. Il est ainsi possible de charger des pièges même si le labyrinthe est de type 1 ou 2 alors qu'ils ne sont pas censé en posséder.

Un particularité de cette fonction est le chargement de la représentation ascii du labyrinthe dont le code est le suivant :

```
int len = maze->width * maze->height;
char * loaded_ascii = calloc(len + 1, 1);
if (maze->type < 2) {
    maze->type1_ascii_rep = loaded_ascii;
} else {
    maze->type23_ascii_rep = loaded_ascii;
}
loaded_ascii[len] = 0;
read(file, loaded_ascii, len);
```

En fonction du type, le pointeur vers la zone mémoire allouée sur le tas pour la représentation ascii sera stocké dans un champ différent dans la structure du labyrinthe.

Nous pouvons fabriquer quasi-arbitrairement des structures maze en mémoire en chargeant des fichiers forgés.

3.3 Exploiter le bug

Un problème se pose cependant : on peut créer les fichiers que l'on veut localement mais ce n'est pas le cas sur la cible distante qui nous permet seulement d'intéragir avec le programme qui est exposé sur le port 4577. Les seuls fichiers que nous pouvons créer à distance sont les sauvegardes de labyrinthes et de tableau des scores.

Le but sera donc de créer un fichier de scores forgé que l'on demandera à charger comme un labyrinthe pour créer des structures de données intéressantes du point de vue d'un attaquant dans la mémoire du programme.

Le format rank est simple, nous avons le contrôle sur tous les champs:

- **nplayers** : Nous pouvons enregistrer jusqu'à 128 joueurs sur le labyrinthe
- **len** et **name** : Pour chaque joueur créé nous contrôlons son nom (max : 128 octets), donc la longueur
- **score** : Possible de le contrôler en mettant des pièges sur le chemin du joueur avec des scores choisis spécifiquement

En enregistrant un nombre choisi de joueur et en choisissant leur nom avec attention, il est possible de créer des fichier maze valides.

Maintenant que nous pouvons forger des structures maze, la question à se poser est: **Quelle forme de structure est intéressante pour nous ?**

Le but est d'exécuter du code sur la machine distante.

3.3.1 Primitive de lecture arbitraire

On rappelle le code de la fonction d'affichage d'un labyrinthe :

```
void print_maze(struct maze * maze, int width) {
    if (maze->type == 3 || maze->type == 2) {
        print_with_traps(maze, width);
    } else {
        print_without_traps(maze, width);
    }
}
```

Pour un labyrinthe avec un type différent de 2 et 3, la chaine de caractères à l'adresse `maze->type1_ascii_rep` sera affichée. Or, ce pointeur est dans une **union** avec la structure représentant les pièges dans le labyrinthe. En créant un labyrinthe avec un type différent de 2 et 3 et en lui attribuant des pièges, il est possible de fabriquer une fausse adresse mémoire à laquelle le programme essaiera de lire lors de l'affichage du labyrinthe. On rappelle la forme de la structure stockant les pièges dans le labyrinthe :

```
struct trap {
    long score;
    short pos;
    char symbol;
    int active;
};
// Dans l'union avec type1_ascii_rep
struct traps {
    char number_traps;
    struct trap traps[256];
};
```

Pour créer l'adresse de lecture (8 octets) avec cette structure, il suffit de stocker l'octet de poids faible dans `traps.number_traps` (1 octet) et le reste dans le score du premier piège (8 octets). Si on essaye d'afficher ce labyrinthe (par exemple en lançant une partie dessus) on peut afficher la mémoire à l'adresse fabriquée dans la struct `traps`.

Ce comportement donne une **primitive de lecture arbitraire** dans la mémoire virtuelle du processus.

3.3.2 Primitive d'écriture arbitraire

Le principe est similaire à celui de la primitive de lecture. Pour un labyrinthe de type 3, il est possible de mettre à jour la position des pièges dans le labyrinthe.

Cette fonction de mise à jour affiche la chaine de caractères à l'adresse **maze->type23_ascii_rep** puis fait

```
fgets(maze->type23_ascii_rep, maze->width * maze->height + 1, stdin);
```

Il est donc possible d'écrire des données arbitraires à l'adresse pointée par `maze->type23_ascii_rep` grâce à ce fgets qui écrit directement à cette adresse et pas dans un buffer temporaire. Tous les checks qui peuvent avoir lieu sur l'entrée après sont anecdotiques étant données que les données sont déjà écrites.

Dans la primitive de lecture, nous avons vu qu'il était possible de corrompre le pointeur `maze->type1_ascii_rep` en créant une `struct traps` particulière. Or ici, c'est le pointeur `maze->type23_ascii_rep` qui est utilisé.

Il est possible de corrompre ce pointeur aussi grâce aux fonctions de mise à jour du labyrinthe. L'option 7 du menu (upgrade) amène au code suivant (simplifié):

```
if (maze->type == 3) {
    update_traps_position(maze);
} else if (maze->type == 2) {
    upgrade_to_type3(maze);
} else {
    upgrade_to_type2(maze);
    upgrade_to_type3(maze);
}
```

```
Choose the level of your maze :
        1. Classic maze
        2. Maze multipass
        3. Maze multipass with traps
3
Random maze or custom maze ? r/c c
Width odd and greater than 3: 5
Height odd and greater than 3: 5
Enter the percentage of wall to remove, default is 5%: 5
Number of traps between 0 and 4: 3
Score value for traps: 25
Will destroy 0 walls
#####
#x^ #
# ###
#^^ o
#####
-*-*-*-*-*-*-*
Do you want to save this maze ? y/n y
What the name of the maze to save ?example3
Maze saved as example3
Menu

1. Register
2. Create maze
3. Load maze
4. Play maze
5. Remove maze
6. View scoreboard
7. Upgrade
8. Exit
7
Do you want to update traps positions y/n?y
Please enter new data respecting the format, for example, current data maze is :
###### ^ ## ####^^ o#####
-*-*-*-*-*-*-*
######^^^## ####   o#####
Trap positions have been updated
```

Figure 10: Mise à jour de la position des pièges

La fonction **upgrade_to_type2()** copie le pointeur dans `maze->type1_ascii_rep` à `maze->type23_ascii_rep`. Sachant qu'on peut contrôler le contenu de `maze->type1_ascii_rep` grâce aux chargement des pièges depuis le fichier, on contrôle l'adresse à laquelle peut écrire des données arbitraires ce qui nous amène à une **primitive d'écriture arbitraire**.

3.3.3 Fuite de mémoire

Depuis 2007 et l'arrivée de Windows Vista, l'ASLR (Address Space Layout Randomization) est activé sur Windows. Cette protection rend aléatoire le placement de certaines zones mémoire (le tas, la pile, les bibliothèques, la base de l'exécutable) dans l'espace d'adressage virtuel des processus. La machine cible ne fait pas exception et une fuite de mémoire sera nécessaire pour exploiter les primitives de lecture/écriture arbitraire.

On rappelle la structure du labyrinthe :

```
struct maze {
    [...]
    char creator[128];  // nom du créateur du labyrinthe
    union {
        // pièges, pour le labyrinthe de type 3
        struct traps maze_traps;
        // représentation ascii du labyrinthe pour le type 1
        char * type1_ascii_rep;
    };
    [...]
};
```

Nous avons une chaine de caractères suivie d'un pointeur, ce qui est un pattern intéressant pour les fuites de mémoire. La chaine de caractères **creator** a une taille de 128 octets, le dernier octet étant un octet nul pour indiquer la fin de la chaine. En enregistrant un joueur, il est possible de rentrer un nom de 127 octets maximum, le dernier octet étant réservé pour la fin de chaine.

En regardant les figures 8 et 9, on voit que le premier octet du fichier maze indique la longueur du nom du créateur (la chaine qui nous intéresse) et que le premier octet du fichier rank indique le nombre de joueurs dans le tableau des scores.

Hors, il est possible d'avoir jusqu'à 128 joueurs dans un tableau des scores, ce qui nous permet de charger un fichier rank comme un labyrinthe ayant un nom de créateur de 127 octets + 1. Ainsi, en affichant le nom du créateur grâce à la fonction *View Scoreboard* du menu, nous pouvons faire fuiter le contenu de **maze->type1_ascii_rep** qui est une adresse du tas si le labyrinthe est de type 1 comme expliqué à la page 14.

3.3.4 Exécuter du code

Nous avons maintenant tous les éléments pour exécuter du code sur la machine distante. Le plus simple est pour moi de faire du ROP (Return Oriented Programming) pour démarrer un shell. Pour prendre le contrôle du flux d'exécution, j'ai choisi de trouver l'adresse de retour de la fonction main sur la pile du programme et d'écrire ma ropchain à cet emplacement qui exécutera le code suivant :

```
WinExec("cmd.exe", 1);
```

Pour parvenir à cet objectif, il faut plusieurs choses:

- L'adresse d'une zone mémoire R/W pour écrire "cmd.exe"
- L'adresse de base de la bibliothèque **ntdll.dll** dans laquelle on va pouvoir trouver des gadgets pour contrôler les arguments de WinExec
- L'adresse de base de la bibliothèque **kernel32.dll** dans laquelle on trouvera la fonction WinExec qui permettra d'exécuter un programme
- L'adresse de base de la pile qu'on va scanner pour trouver l'adresse de retour de la fonction main

Etant novice en terme d'exploitation Windows, j'ai procédé de la manière suivante. A partir de ma fuite mémoire sur le tas, j'ai observé l'état du tas grâce à **Process Hacker**[5] et je me suis rendu compte qu'il y avait l'adresse de la base de l'exécutable à un offset de 216 octets par rapport à la fuite. On peut observer sur la figure 11

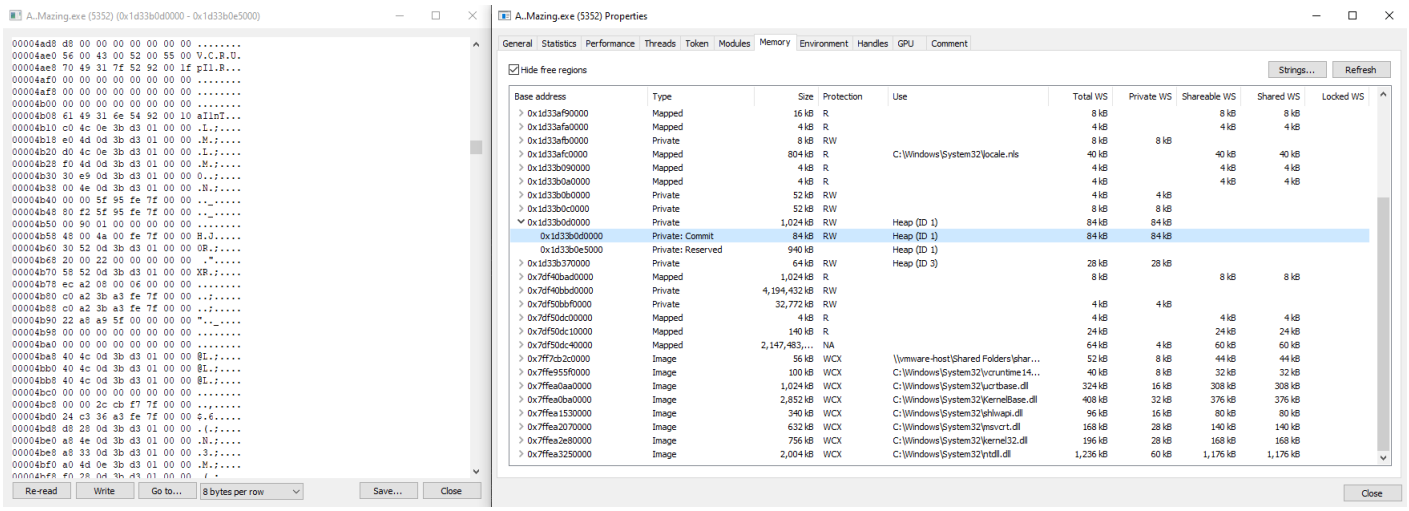

Figure 11: Etat du tas dans Process Hacker

l'adresse de base du programme à l'offset 0x4bc8 dans le tas, l'adresse qui fuite est celle à l'offset 0x4af0. Cet offset entre la fuite et l'adresse de base de l'exécutable est constant entre les exécutions et identique sur la cible distante. Cet ensemble de conditions est un coup de chance.

A partir de l'adresse de base de l'exécutable, on peut trouver sa section **.data** qui est R/W et dans laquelle on pourra écrire "cmd.exe".

On peut aussi trouver l'**IAT**[6] du programme qui contient les adresses résolues des fonctions importées par l'exécutable depuis les différentes bibliothèques. Beaucoup de fonctions sont importées et on peut trouver des adresses de fonctions appartenant à **kernel32.dll** et à **ntdll.dll**. A partir de là, il est facile de calculer les adresses de base de ces deux bibliothèques.

Il ne manque plus que l'adresse de base de la pile du processus (c'est la partie qui m'a pris le plus de temps).

J'ai fini par découvrir qu'il existe un symbole dans **ntdll.dll** nommé **TlsBitMap** qui contient l'adresse du **PEB**[7] (+0x80) à un offset de +8 par rapport au symbole.

Dans Windows, à chaque processus est associé un PEB. Cette structure de données décrit le processus auquelle elle est associée. Un processus est composé de plusieurs Thread qui s'exécutent en parallèle et possèdent chacun leur propre pile. A chaque Thread est associé un **TEB**[8] qui est une structure de données décrivant le Thread auquel il est associé. Le TEB du Thread numéro i se trouve à l'adresse **PEB + i * 0x1000**.

On trouve donc l'adresse du Thread principal (numéro 1) du processus à l'adresse PEB + 0x1000 que l'on peut afficher avec WinDbg :

[5] `https://processhacker.sourceforge.io/`
[6] Import Address Table
[7] Process Environment Block
[8] Thread Environment Block

```
0:004> x ntdll!TlsBitMap
00007ffe`a483a440 ntdll!TlsBitMap = <no type information>
0:004> dq 00007ffe`a483a440
00007ffe`a483a440  00000000`00000040 000000a5`61fe8080
00007ffe`a483a450  5560000f`ffc6abb9 00000000`00000000
00007ffe`a483a460  00000000`00000000 00000000`00000000
00007ffe`a483a470  00000000`00000000 00000000`00000004
00007ffe`a483a480  00000000`00000000 00000000`00000000
00007ffe`a483a490  00000000`00000000 00000000`00000000
00007ffe`a483a4a0  0000022e`8a402aa0 00007ffe`a46d0000
00007ffe`a483a4b0  00000000`00000000 00000000`00000000
0:004> !peb
PEB at 000000a561fe8000
```

Figure 12: Retrouver l'adresse du PEB grâce à TlsBitMap dans WinDbg

```
0:000> !teb
TEB at 000000a561fe9000
    ExceptionList:        0000000000000000
    StackBase:            000000a561d10000
    StackLimit:           000000a561d0d000
    SubSystemTib:         0000000000000000
    FiberData:            0000000000001e00
    ArbitraryUserPointer: 0000000000000000
    Self:                 000000a561fe9000
    EnvironmentPointer:   0000000000000000
    ClientId:             0000000000001d44 . 0000000000001d78
    RpcHandle:            0000000000000000
    Tls Storage:          0000022e8a405cf0
    PEB Address:          000000a561fe8000
    LastErrorValue:       0
    LastStatusValue:      c000000d
    Count Owned Locks:    0
    HardErrorMode:        0
```

Figure 13: Contenu de la structure TEB

On voit que le TEB contient l'adresse de la base de la pile à l'offset +8.

Maintenant que nous avons tous les éléments nécessaire à la création de notre ropchain, nous pouvons mettre en application.

J'ai utilisé **rp++**[9] pour trouver les gadgets nécessaires à l'exploitation dans ntdll. Nous avons besoins de gadgets permettant de contrôler **rcx** et **rdx** qui sont les registres permettant de passer les deux premiers arguments aux fonctions dans la convention d'appel de Windows x64. Nous avons également besoin d'un gadget **ret** simple pour aligner le pointeur de pile **RSP** sur une adresse multiple de 16 pour éviter que le programme plante.

Les étapes pour l'exploitation complète sont les suivantes :

1. Faire fuiter une adresse du tas
2. Utiliser la primitive de lecture pour lire à fuite + 216 et déterminer la base de l'exécutable en mémoire
3. En déduire l'adresse du .data de l'exécutable qui est R/W
4. Utiliser la primitive d'écriture pour écrire cmd.exe dans .data

[9] https://github.com/0vercl0k/rp

5. Utiliser la primitive de lecture pour lire certaines entrées de l'IAT du programme pour déduire les adresses de base de ntdll.dll et kernel32.dll
6. En déduire les adresses des gadgets nécessaires et de WinExec
7. Utiliser la primitive de lecture pour déterminer l'adresse du PEB puis du TEB puis de la pile.
8. Scanner la pile jusqu'à trouver l'adresse de retour calculable grâce à l'adresse de base du programme
9. Utiliser la primitive d'écriture pour écrire notre ropchain à la place de l'adresse de retour pour prendre le contrôle du flux d'exécution et exécuter un shell

Plus qu'à exécuter notre exploit :

```
> python3 exploit.py REMOTE
[+] Opening connection to challenge2021.sstic.org on port 4577: Done
[*] Leaking...
[+] Leak : 0x24598b14810
[+] exec base : 0x7ff76b3e0000
[+] ntdll base : 0x7fffeca70000
[+] kernel32 base : 0x7fffeb1f0000
[+] Peb base : 0x748dd7a000
[+] Stack base : 0x748db90000
[*] Ret addr : 0x7ff76b3e5e58
FOUUUUUUUUUUUUUUUND!!!
[+] Ret addr @0x748db8fa28
[*] Writing cmd.exe @0x7ff76b3ea070
[*] Writing 8 bytes to 0x7ff76b3ea070
[*] Done
[*] Writing ropchain
[*] Writing 24 bytes to 0x748db8fa28
[*] Done
[*] Writing 24 bytes to 0x748db8fa40
[*] Done
[*] Writing 24 bytes to 0x748db8fa58
[*] Done
[*] Writing 24 bytes to 0x748db8fa70
[*] Done
[*] Got shell
[*] Switching to interactive mode
C:\users\challenge\mazes\7a8ca45f5176928919f526588b768e49>$ whoami
whoami
desktop-qte4bqs\challenge
```

Le scan de la pile prend un certain temps mais l'exploit finit par aboutir et on a accès à la machine. cmd.exe est restreint, on ne peut pas se déplacer dans le système de fichiers mais en lançant powershell ça marche. On trouve un fichier **DRM.zip** sur le bureau de la machine qu'on télécharge avec quelques lignes de python en plus dans notre exploit :

```
prompt="C:\\users\\challenge\\mazes\\"
r.recvuntil(">")
print("[*] Got shell")
while True:
    r.sendline("type C:\\users\\challenge\\Desktop\\DRM.zip")
    data = r.recvuntil(prompt)
    if len(data) > 800:
        print("[+] Downloaded file")
        open("dumped", "wb").write(data)
        break
```

Cette archive dézippée, on obtient 3 fichiers :

- DRM_server.tar.gz
- libchall_plugin.so
- Readme

Voici le contenu du fichier Readme :

> Here is a prototype of the DRM solution we plan to use for SSTIC 2021. It's 100% secure, because keys are stored on a device specifically designed for this. It uses a custom architecture which garantee even more security! In any case, the device is configured in debug mode so production keys can't be accessed.
> The file DRM_server.tar.gz is the remote part of the solution, but for now we can't emulate the device, so some feature are only available remotely. The file libchall_plugin.so is a VLC plugin that will allow you to test the solution, if you ever decide to install Linux :)
> Trou

libchall_plugin.so est donc un plugin VLC linux qui permet de tester la solution de DRM à étudier pour la suite.

J'ai donc installé une machine virtuelle Ubuntu 20.04, chargé le plugin en le copiant dans **/usr/lib/x86_64-linux-gnu/vlc/plugins** et lancé VLC.

En allant dans **View** > **Playlist**, on découvre une entrée étrange : **Chall Media Services**. On peut également voir les capacités du plugin dans **Tools** > **Plugins and Extensions**. On découvre que le plugin permet d'accéder à un service multimédia distant, ce qui se confirme en cliquant sur Chall Media Services :

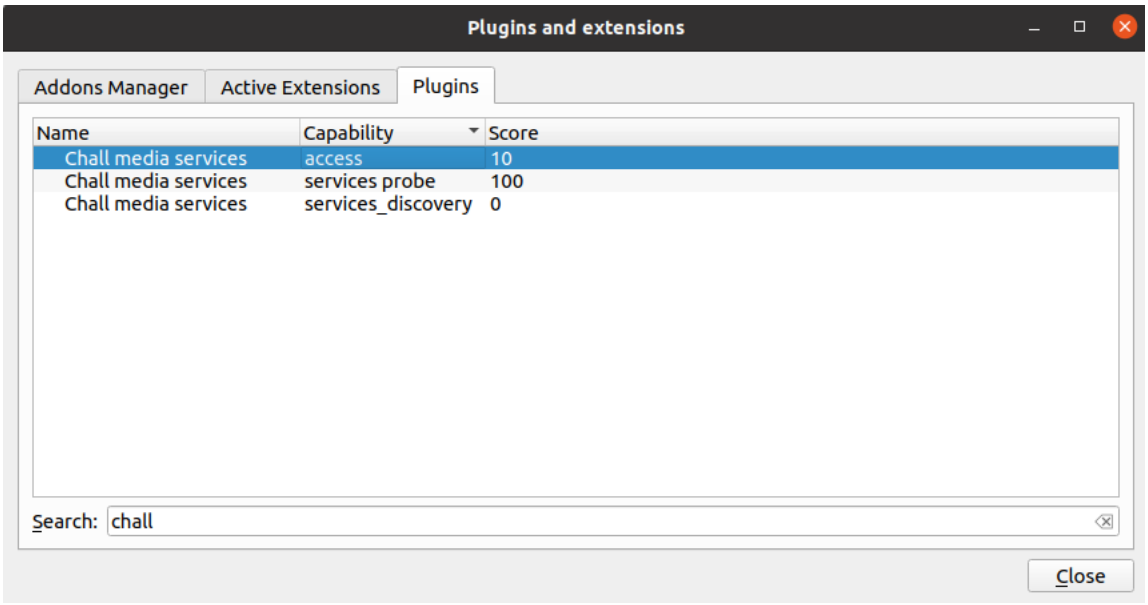

Figure 14: Les capacités du plugin

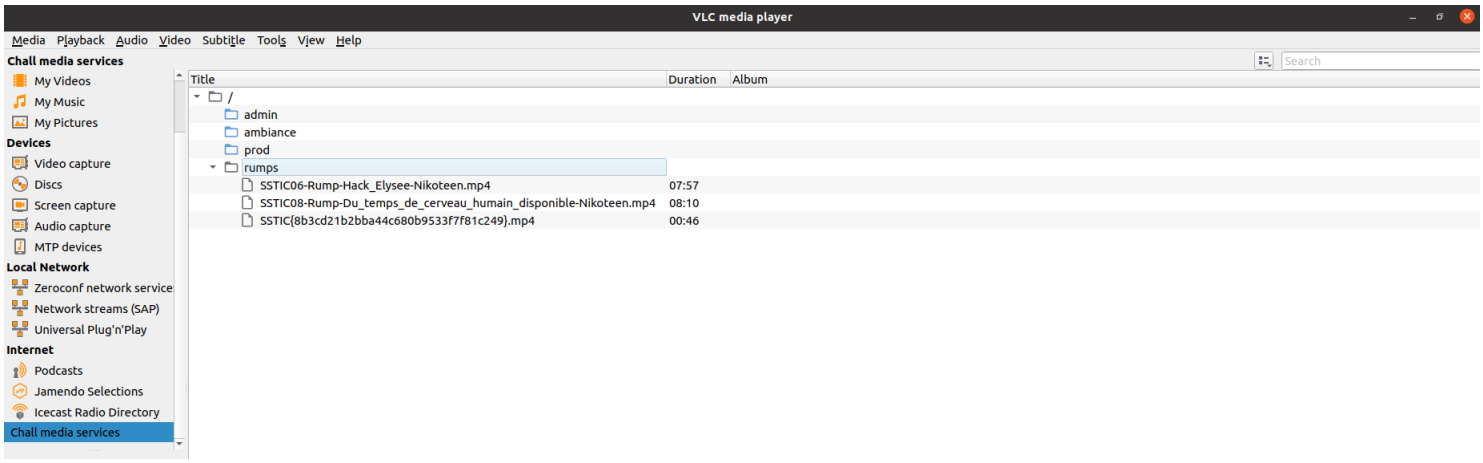

Figure 15: Le service multimédia distant

Le seul dossier auquel on peut accéder est **rumps** (les autres renvoient un message d'erreur indiquant que l'accès n'est pas autorisé) et on y trouve le flag de l'étape 2 dans le nom de l'une des vidéos :

SSTIC{8b3cd21b2bba44c680b9533f7f81c249}

4 Etape 3

4.1 Découvrir l'environnement, par le réseau

L'archive **DRM_server.tar.gz** contient 3 fichiers : un système de fichiers, un noyau linux et un script qui permet de lancer une machine virtuelle qemu à partir du système de fichiers et du noyau.

Avant de m'attaquer à cette partie de l'environnement, j'ai choisi de comprendre le fonctionnement du plugin VLC.

Ma première approche a été d'utiliser Wireshark pour voir les connexions effectuées par le plugin. J'ai donc appliqué un filtre sur Wireshark pour ne garder que les paquets ayant pour adresse source ou destination celle de ma machine virtuelle puis j'ai accédé au serveur multimédia et au dossier rumps. On peut analyser les différentes conversations TCP du plugin avec **Statistiques** > **Conversations** > **TCP**.

Ethernet · 3 | IPv4 · 4 | IPv6 | TCP · 7 | UDP · 10

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration
192.168.1.36	60890	62.210.125.243	8080	300	3 926k	190	12k	110	3 913k	4.579830	0.8525
192.168.1.36	59202	62.210.125.243	1337	51	3 689	36	2 602	15	1 087	5.487773	21.8581
192.168.1.36	60894	62.210.125.243	8080	19	2 616	14	1 276	5	1 340	13.515784	0.0987
192.168.1.36	60896	62.210.125.243	8080	19	2 665	14	1 392	5	1 273	13.710319	0.2527
192.168.1.36	60898	62.210.125.243	8080	231	2 645k	154	10k	77	2 634k	26.574137	0.3837
192.168.1.36	60900	62.210.125.243	8080	178	1 622k	118	8 028	60	1 614k	26.997086	0.3385
192.168.1.36	60902	62.210.125.243	8080	69	261k	48	3 528	21	257k	27.381263	0.2233

Figure 16: Les conversations TCP du plugin VLC

Un échange DNS au début de la capture nous apprend que l'adresse 62.210.125.243 correspond au nom d'hôte **challenge2021.sstic.org** .

On remarque que la conversation sur le port 1337 reste ouverte pour plusieurs échanges sur le port 8080.

Wireshark permet de suivre individuellement chacune de ces conversations grâce à la fonction **Follow Stream** dans la liste des conversations et nous pouvons ainsi déterminer plus précisément le contenu de chaque échange :

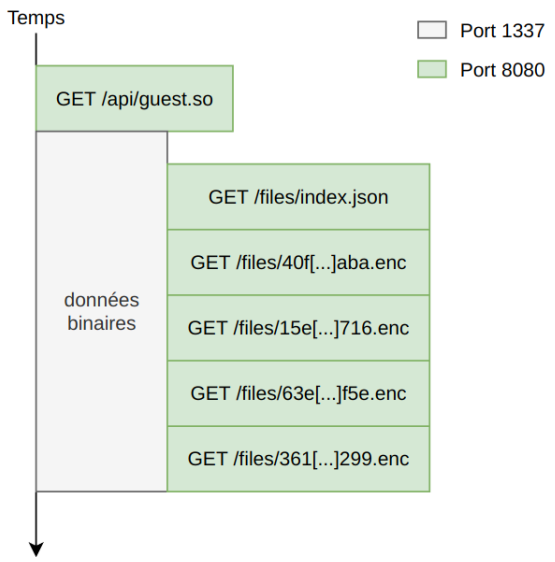

Figure 17: Un chronogramme des conversations TCP entre le plugin et le serveur

Les conversations sur le port 8080 sont des échanges HTTP pour télécharger des fichiers et la conversation sur le port 1337 semble être un protocole inconnu échangeant des données binaires auxquelles on ne peut pas encore donner du sens.

On peut télécharger avec **wget** les différents fichiers récupérés par le plugin pour les analyser. **guest.so** est un ELF shared object, c'est-à-dire une bibliothèque. **index.json** a le contenu suivant:


```
[
    {
     "name": "930[...]08c.enc",
     "real_name": "admin",
     "type": "dir_index",
     "perms": "0000000000000000",
     "ident": "75edff360609c9f7"
    },
    {
     "name": "4e4[...]1af.enc",
     "real_name": "ambiance",
     "type": "dir_index",
     "perms": "00000000cc90ebfe",
     "ident": "6811af029018505f"
    },
    {
     "name": "e14[...]328.enc",
     "real_name": "prod",
     "type": "dir_index",
     "perms": "0000000000001000",
     "ident": "d603c7e177f13c40"
    },
    {
     "name": "40f[...]aba.enc",
     "real_name": "rumps",
     "type": "dir_index",
     "perms": "ffffffffffffffff",
     "ident": "68963b6c026c3642"
    }
]
```


J'ai tronqué les noms en **.enc** pour la visibilité (ils font normalement 68 caractères avec l'extension). On retrouve dans ce fichier json les noms des 4 dossiers visibles depuis VLC, associés à ce qui semblent être des permissions, un identifiant, un nom de fichier et un type.

Les fichiers **.enc** suggèrent par leur extension qu'ils sont chiffrés, ce qu'on peut facilement vérifier en faisant un calcul d'entropie avec binwalk. L'entropie plafonne aux alentours de 1.0, ce qui est caractéristique des fichiers chiffrés.

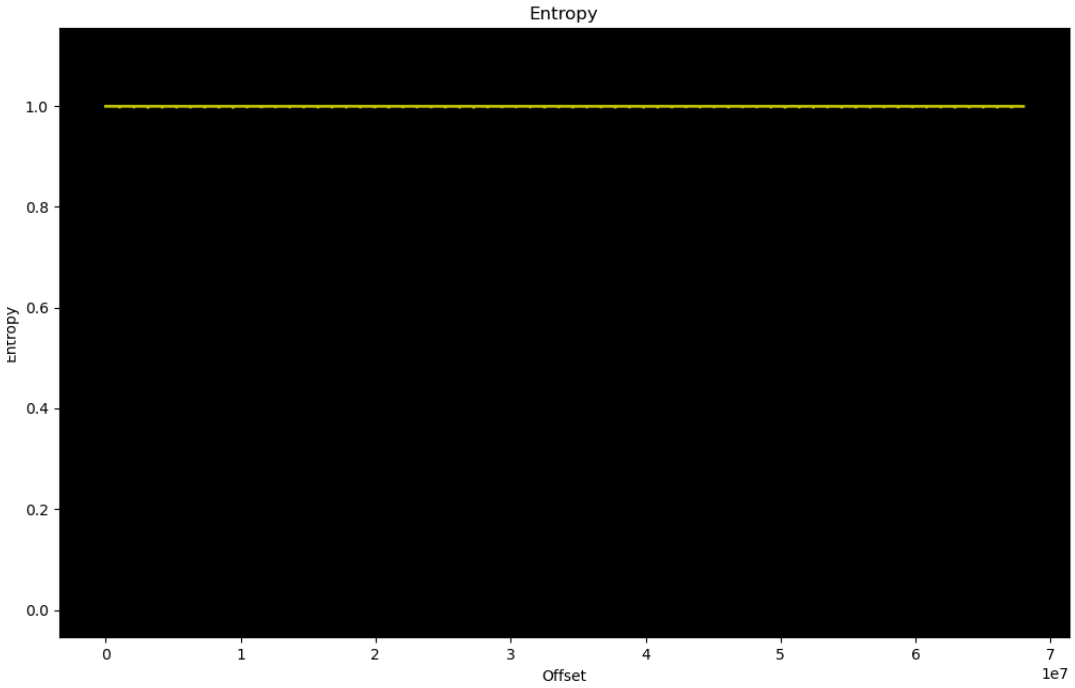

Figure 18: Entropie du fichier 15e[...]716.enc

Le fichier json nous donne beaucoup d'informations. On voit que le dossier rumps auquel on demande à accéder est associé à un nom de fichier **.enc**, c'est ce fichier qui est téléchargé juste après **index.json** (voir figure 17). 3 autres fichiers enc sont téléchargés après rumps, ce qui correspond au nombre de vidéos qu'on peut voir dans le dossier rumps dans VLC.

De plus, en regardant plus attentivement les échanges sur le port 1337, on se rend compte de la chose suivante : avant chaque échange http pour télécharger un fichier, il y a une requête du plugin vers le serveur (21 octets) suivie d'une réponse du serveur (17 octets). C'est le seul moment où des données passent sur cette connexion, hormis à l'ouverture où le serveur envoie **STIC**.

Par cette rétro-ingénierie des données qui passent sur le réseau, on déduit les choses suivantes :

- Le plugin télécharge d'abord une bibliothèque **guest.so**, on ne sait pas encore comment elle est utilisée
- Il ouvre ensuite une connexion avec le serveur sur le port 1337 qui reste ouverte tout le long des échanges
- Il télécharge ensuite 2 fichiers via HTTP : **index.json** qui indique les dossiers accessibles sur le serveur ainsi que leurs permissions puis 40f[...]aba.enc qui correspond à **rumps**, on suppose que c'est aussi un .json de la même forme qu'index.json indiquant les informations des 3 vidéos qu'il contient
- 3 fichiers .enc sont téléchargés, on suppose qu'ils correspondent aux 3 vidéos contenues dans le dossier **rumps**

- Avant chaque échange HTTP pour télécharger un fichier .enc, on peut observer une requête du plugin vers le serveur suivie d'une réponse sur le port 1337. D'après le fichier Readme, on sait que des clés sont stockées sur un appareil dédié à cette fonction. On peut donc supposer que la connexion sur le port 1337 sert à interagir avec ce fameux appareil et demander la clé de déchiffrement correspondant au fichier .enc qui va être téléchargé

4.2 Aller plus loin dans la rétro-ingénierie

4.2.1 Proxy, Hooking, disséquer le plugin

Wireshark a permis d'obtenir une première intuition sur le fonctionnement du système. Maintenant, il faut regarder plus en détail comment le plugin fonctionne. On ouvre IDA et on commence la rétro-ingénierie.

Dans notre première analyse via le réseau, le plugin commence par télécharger la librairie **guest.so**. On va donc regarder de ce côté-là.

On recherche dans le plugin la chaine de caractères *guest.so*. On y trouve une référence dans la fonction **open_state_internal**. Le plugin n'est pas strippé, c'est-à-dire que les symboles n'ont pas été retirés ce qui facilite grandement la rétro-ingénierie.

Cette fonction a le comportement suivant :

1. Crée un fichier temporaire via la fonction **mkstemp64** avec pour template **/tmp/libVMXXXXXX.so**. Les X seront remplacés par des caractères aléatoires.

2. Télécharge soit **/api/auth.so** ou **/api/guest.so** en fonction d'une condition sur le premier argument de la fonction qui semble être une structure et écrit le contenu dans le fichier temporaire précédemment créé

3. Ouvre le fichier temporaire avec **dlopen**, une fonction qui permet de charger en mémoire une bibliothèque pendant l'exécution (dl = dynamic loader)

4. Cherche les adresses des fonctions **useVM**, **getPerms**, **getIdent** dans la bibliothèque précédemment chargée grâce à la fonction **dlsym** qui utilise le handler renvoyé par la fonction **dlopen**

5. Stocke les adresses de ces 3 fonctions dans la structure en argument

La bibliothèque **guest.so** est donc chargée dans la mémoire de VLC grâce au dynamic loader de Linux pour utiliser les fonctions **useVM**, **getPerms** et **getIdent**. Avant de reverser **guest.so**, on va regarder dans quel contexte sont utilisées ces fonctions et si possible découvrir leur utilité en boîte noire.

VLC possède un code d'exemple pour l'écriture de modules[10] qui nous apprend que la première fonction appelée lors du chargement du module est la fonction **Open** qui permet d'initialiser les structures de données du module, les devices et I/O. Le **plugin** ne possède pas de fonction **Open** mais la fonction **OpenAccess** semble avoir ce rôle.

[10] `https://wiki.videolan.org/Hacker_Guide/How_To_Write_a_Module/`

Cette fonction **OpenAccess** effectue les actions suivantes :

1. Extraire les arguments :
   - `--media-server`
   - `--media-server-login`
   - `--media-server-pass`
   - `--media-server-permcheck`
   - `--key-server-addr`
   - `--key-server-port`
2. Initialiser la structure **global_ctx** avec ces arguments. Si ces arguments sont nuls, ils sont initialisés à des valeurs par défaut pour certains :
   - `http://challenge2021.sstic.org:8080` pour media-server
   - `62.210.125.243` pour key-server-addr
   - 1337 pour key-server-port
3. Appeler la fonction **remote_login** qui appelle **open_state** > **open_state_internal**. On découvre que la structure passée en argument de **open_state_internal** est **global_ctx** et que le check au début de la fonction vérifie si un login/password ont été rentrés en arguments pour télécharger **auth.so** au lieu de **guest.so**. Les adresse des fonctions **getPerms**, **getIdent** et **useVM** sont donc stockées dans des champs de la structure **global_ctx**.
4. Appeler **open_index** qui télécharge **index.json**.

Cette fonction effectue encore d'autres actions mais on peut d'ores et déjà valider l'hypothèse du plugin qui demande des clés de déchiffrement via la connexion sur le port 1337 qui correspond à l'échange avec le serveur de clés.

On peut utiliser la fonction de références croisées d'IDA pour remonter la chaîne d'appel de la fonction **open_state_internal**. Grâce aux références croisées, on remonte jusqu'à la fonction **download_file** qui a un nom pour le moins intéressant et qui effectue les actions suivantes :

1. Appelle **get_current_permissions**
2. Appelle **get_file_key_part_5**
3. Appelle **download_file_with_key**

La fonction **get_current_permissions** appelle **open_state**. En regardant le code d'**open_state**, on découvre que les 3 fonctions chargées depuis la bibliothèque sont appelées. Pour découvrir quels types d'arguments sont passés dans ces fonctions sans reverser tout le contexte, on va faire du **hooking**[11] pour afficher les données manipulées par ces fonctions.

[11] Technique permettant de poser des "hook" qui vont réaliser des actions supplémentaires à des moments déterminés dans l'exécution d'un programme existant

Une technique pour faire du hooking de fonctions sous Linux se base sur la génération d'une bibliothèque redéfinissant les fonctions à hooker pour résoudre les symboles correspondant à ces fonctions avant de charger les véritables bibliothèques contenant ces fonctions. Cette méthode utilise la variable d'environnement **LD_PRELOAD** qui a pour valeur le chemin vers notre bibliothèque redéfinissant les fonctions à hooker. Cette bibliothèque sera chargée avant toutes les autres ce qui nous permet d'exécuter notre code au lieu de celui des fonctions hookées.

En général, quand on utilise cette technique pour faire du logging[12] (ce qu'on cherche à faire ici), on veut quand même exécuter la vraie fonction appelée et afficher certaines informations au moment de l'appel ou du retour. Pour cela, on utilise le loader dynamique avec la fonction **dlsym** pour retrouver l'adresse de la vraie fonction que l'on appelle grâce à ce pointeur de fonction à l'intérieur de notre hook. Exemple pour hooker la fonction **puts** :

```c
#include <stdio.h>
// gcc test.c -o test

void main() {
        puts("Hello World !");
}
```

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>

// gcc -shared -fPIC -o hook hook.c -ldl

typedef int (*real_puts_t)(const char *str);

int real_puts(const char *str) {
    return ((real_puts_t)dlsym(RTLD_NEXT, "puts"))(str);
}

int puts(const char *str) {
    printf("Puts arg : %s\n", str);
    return real_puts(str);
}
```

```
> LD_PRELOAD=$(pwd)/hook ./test
Puts arg : Hello World !
Hello World !
```

Le problème ici est qu'on veut hooker des fonctions dont les symboles sont résolus pendant l'exécution avec le loader dynamique. Donc même si on override les symboles **getPerms**, **getIdent** et **useVM** avec LD_PRELOAD ça n'aura aucun effet.

Il faut donc hooker **dlsym** pour pouvoir récupérer les adresses résolues au moment de l'appel à **dlsym** par le plugin et renvoyer les adresses de nos propres fonctions.

[12] Afficher des informations pendant l'exécution d'un programme

```c
// dlsym de la libc
extern void *_dl_sym(void *, const char *, void *);

static void * (*real_dlsym)(void *, const char *)=NULL;
static void * (*real_getPerms)(void *a1);
static void * (*real_useVM)(void *a1, void *a2);
static void * (*real_getIdent)(void *a1);

// Redéfinition du dlsym du loader dynamique
extern void *dlsym(void *handle, const char *name)
{
    // Retrouver l'adresse du véritable dlsym du loader dynamique
    // en utilisant le dlsym de la libc
    if (real_dlsym == NULL)
        real_dlsym=_dl_sym(RTLD_NEXT, "dlsym", dlsym);

    // Résoudre le symbole avec le véritable dlsym
    void * ret = real_dlsym(handle,name);

    // Si le symbole à résoudre est celui d'une des fonctions à
    // hooker alors on renvoie l'adresse de notre propre fonction
    // et on stocke l'adresse de la vraie fonction pour pouvoir
    // l'appeler dans notre hook
    if (!strcmp(name, "getPerms")) {
        real_getPerms = ret;
        return getPerms;
    }
    if (!strcmp(name, "useVM")) {
        real_useVM = ret;
        return useVM;
    }
    if (!strcmp(name, "getIdent")) {
        real_getIdent = ret;
        return getIdent;
    }
    return ret;
}
```

La libc possède sa propre implémentation de **dlsym** qui n'a pas exactement le même comportement que celui du loader dynamique pour certains cas limites mais c'est suffisant pour retrouver l'adresse du **dlsym** du loader dynamique.

On est maintenant prêts à logger des informations sur les appels aux fonctions **getPerms**, **getIdent** et **useVM**.

```
[GETIDENT] a1=0x7fcf24855850
[GETPERMS] a1=0x7fcf24855838
[useVM] a1=0x7fcf24855830, a2=0x7fcf24855840
```

Figure 19: Le hook fonctionne

Les arguments sont des pointeurs donc les fonctions utilisent probablement ces adresses pour recevoir des arguments ou renvoyer une valeur. En affichant la mémoire aux adresses des arguments, avant et après les appels de fonctions, on se rend compte des choses suivantes :

- **getIdent** modifie 4 octets à l'adresse pointée par a1
- **getPerms** modifie 8 octets à l'adresse pointée par a1 pour les mettre à `ffffffffffffffff`
- **useVM** modifie 16 octets à l'adresse pointée par a2

On en déduit que **getIdent** renvoie un identifiant de 4 octets dans l'adresse passée en argument, **getPerms** renvoie les permissions du client à l'adresse passée en argument, **useVM** prend en argument un nombre d'octets entre 1 et 16 stockés à l'adresse dans le premier argument (in) et renvoie une valeur de 16 octets à l'adresse dans le second argument (out).

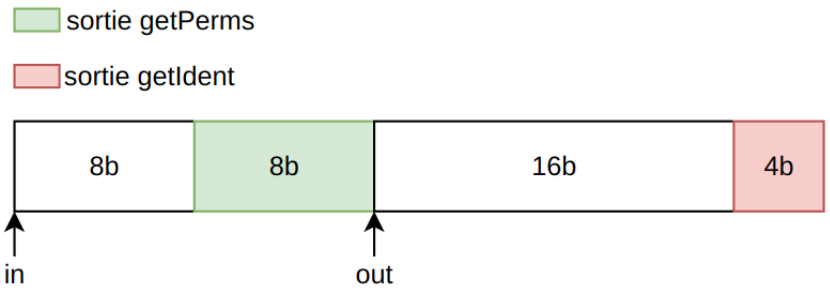

Figure 20: Disposition de la mémoire lors de l'appel à useVM

On remarque que **getPerms** renvoie toujours `ffffffffffffffff`, ce qui correspond au champ **perms** du dossier rumps, le seul auquel on peut accéder via VLC.

A ce moment, j'ai émis l'hypothèse que les données passant dans ces fonctions avaient un lien avec les requêtes pour demander les clés au serveur de clés. Pour faciliter la visualisation des données transitant sur le réseau, j'ai utilisé un proxy simple en python qui intercepte le trafic entre le plugin et le serveur de clés. La mise en place est simple grâce aux arguments qu'on peut passer au plugin pour changer l'adresse du serveur de clés.

On peut maintenant afficher les données qui sont échangées avec le serveur de clés en parallèle du logging qu'on fait sur les données en entrées sorties des fonctions **getPerms**, **getIdent** et **useVM** et un pattern se dessine immédiatement.

```
[FROM SERVER] :  53544943
[FROM CLIENT] :  0077dee446a5976a0762cff0ccd7e297c988bd9760
[FROM SERVER] :  01
[FROM SERVER] :  00000000000000ffffffffffffffff
[FROM CLIENT] :  01b0f70e870519ce024782fc2b268f5b5488bd9760
[FROM SERVER] :  03
[FROM SERVER] :  696464b99ff1e025105f6235fa67c91d
```

Figure 21: Logs du proxy

Les données envoyées sont la concaténation de la sortie de useVM et la sortie de getIdent préfixée d'un octet à 0 pour le premier échange avant de télécharger

```
[GETIDENT] a1=0x7f9694307850
        Client id = 88 bd 97 60
[GETPERMS] a1=0x7f9694307838
        Current perms = ffffffffffffffff
[useVM] a1=0x7f9694307830, a2=0x7f9694307840
        Asking for res : 0
        With permissions : ffffffffffffffff
        Token : 77 de e4 46 a5 97 6a 07 62 cf f0 cc d7 e2 97 c9 88 bd 97 60
[GETPERMS] a1=0x7f96943077f8
        Current perms = ffffffffffffffff
[GETPERMS] a1=0x7f96943077b8
        Current perms = ffffffffffffffff
[GETPERMS] a1=0x7f9694b8a7b8
        Current perms = ffffffffffffffff
[GETPERMS] a1=0x7f9694b8a7b8
        Current perms = ffffffffffffffff
[GETPERMS] a1=0x7f9694b8a7b8
        Current perms = ffffffffffffffff
[GETIDENT] a1=0x7f9694b8a780
        Client id = 88 bd 97 60
[GETPERMS] a1=0x7f9694b8a728
        Current perms = ffffffffffffffff
[useVM] a1=0x7f9694b8a720, a2=0x7f9694b8a770
        Asking for res : 68963b6c026c3642
        With permissions : ffffffffffffffff
        Token : b0 f7 0e 87 05 19 ce 02 47 82 fc 2b 26 8f 5b 54 88 bd 97 60
```

Figure 22: Données récupérées grâce au hooking

index.json ou d'un octet à 1 pour tous les autres échanges. On détermine aussi le sens de l'entrée de la fonction **useVM**. Avec un schéma :

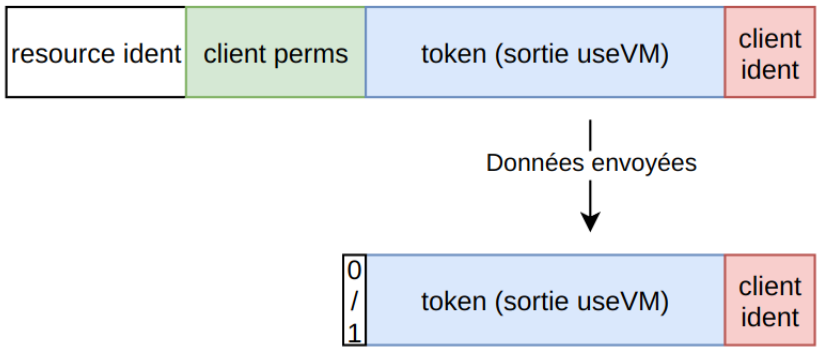

Figure 23: Données utilisées pour demander une clé

En cliquant sur le dossier **rumps** dans VLC, on voit un appel à **useVM** avec pour argument l'ident de **rumps** (sur la figure 22) visible dans index.json. Le premier appel à **useVM** correspond au téléchargement de index.json.

Le premier octet des données envoyées peut faire penser à une commande envoyée au serveur de clés; le reste des données à des arguments. index.json n'étant pas chiffré, il n'y a pas besoin de demander une clé de déchiffrement pour ce fichier.

Ainsi, le premier octet à 1 indiquerait une demande de clé tandis que le premier octet à 0 demanderait autre chose. On remarque cependant que les données renvoyées par le serveur pour la commande 0 sont `0000000000000000ffffffffffffffff` soit les arguments de l'appel à **useVM** qui ont généré la requête précédant cette réponse.

Regardons un peu plus en détail la fonction **download_file_with_key** appelée par **download_file**. La fonction initialise un contexte gcrypt[13] avec **gcry_cipher_open** puis **gcry_cipher_setkey** et **gcry_cipher_setctr**. Le fichier demandé est ensuite téléchargé puis déchiffré via gcrypt.

L'initialisation du contexte nous renseigne sur le type d'algorithme de chiffrement utilisé. Nous sommes en présence d'un chiffrement via **AES-128** en mode **CTR**. AES est un algorithme de chiffrement symétrique, la clé utilisée est la même au chiffrement et au déchiffrement. On rajoute les fonction **gcry_cipher_setkey** et **gcry_cipher_setctr** dans notre librairie de hooking pour logger les arguments et on demande à ouvrir le dossier rumps dans VLC.

```
[useVM] a1=0x7fb3f0396720, a2=0x7fb3f0396770
        Asking for res : 68963b6c026c3642
        With permissions : ffffffffffffffff
        Token : 20 6a 89 d0 be 34 49 5e aa c0 70 34 df 57 e9 f4 b8 df 97 60
[OPEN_CIPHER] cipher=0x7fb3f0386710
[SETKEY] cipher=0x7fb3d4002450, key= 69 64 64 b9 9f f1 e0 25 10 5f 62 35 fa 67 c9 1d, size=16
[SETCTR] cipher=0x7fb3d4002450, ctr= 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01, size=16
```

Figure 24: Calcul du token suivi de l'initialisation du contexte gcrypt

```
[FROM CLIENT] :  01206a89d0be34495eaac07034df57e9f4b8df9760
[FROM SERVER] :  03
[FROM SERVER] :  696464b99ff1e025105f6235fa67c91d
```

Figure 25: Les requêtes correspondantes vues par le proxy

On voit bien le token calculé par useVM envoyé au serveur de clés. Le serveur répond un premier octet (un message de statut ?) puis envoie la clé. La clé est ensuite utilisée pour initialiser le contexte gcrypt pour déchiffrer le fichier.

On a maintenant une idée plus claire du fonctionnement du système :

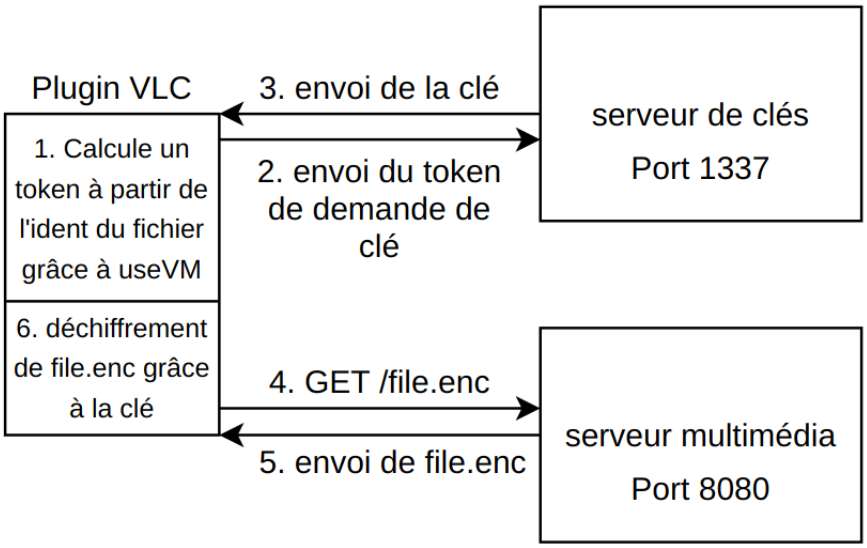

Figure 26: Schéma du fonctionnement du système pour demander l'accès à un fichier

[13] Librairie dédiée à la cryptographie implémentant de nombreux algorithmes de chiffrement.

Avec quelques lignes de python, on peut déchiffrer le fichier **rumps**.

```python
#!/usr/bin/env python3
from Crypto.Cipher import AES
from Crypto.Util import Counter

MODE = AES.MODE_CTR
counter = Counter.new(128)
key = bytes.fromhex("696464b99ff1e025105f6235fa67c91d")
data = open("rumps.enc", "rb").read()
crypto = AES.new(key, MODE, counter=counter)
print(crypto.decrypt(data))
```


```
[
    {
     "name": "15e[...]716.enc",
     "real_name": "SSTIC06-Rump-Hack_Elysee-Nikoteen.mp4",
     "type": "mp4",
     "perms": "ffffffffffffffff",
     "ident": "6fc51949a75bfa98"
    },
    {
     "name": "63e[...]f5e.enc",
     "real_name": "SSTIC08-Rump[...]Nikoteen.mp4",
     "type": "mp4",
     "perms": "ffffffffffffffff",
     "ident": "675160efed2d139b"
    },
    {
     "name": "361[...]299.enc",
     "real_name": "SSTIC{8b3cd21b2bba44c680b9533f7f81c249}.mp4",
     "type": "mp4",
     "perms": "ffffffffffffffff",
     "ident": "583c5e51d0e1ab05"
    }
]
```


On retrouve effectivement les vidéos disponibles dans le dossier rumps!

Le but va donc être de demander les clés pour accéder au reste du contenu du serveur multimédia. Ma première idée a été d'utiliser ma librairie de hooking pour patcher la valeur de retour de **getPerms** pour que le programme pense que nous ayons les permissions 0, nécessaires pour avoir accès à tout le contenu du serveur.

Malheureusement cette approche s'est avérée infructueuse car **useVM** n'écrit rien en sortie, il doit donc y avoir une forme de détection de ce genre de truanderie. On va donc s'intéresser de plus près à **guest.so**.

4.2.2 Reverser guest.so

On charge dans IDA la bibliothèque **guest.so** pour regarder le fonctionnement des trois fonctions. Les symboles **getPerms**, **getIdent** et **useVM** sont bien présents dans le binaire. Ces trois fonctions appellent toutes la même fonction (qui n'a pas de symbole associé) avec seulement le premier argument qui change.

Cette fonction est une grosse boucle dans laquelle se trouve un switch. Par expérience, vu la structure de la fonction et le **VM** dans **useVM**, on détermine que cette fonction est obfusquée via une technique de machine virtuelle. On appelle donc cette fonction **run_vm**.

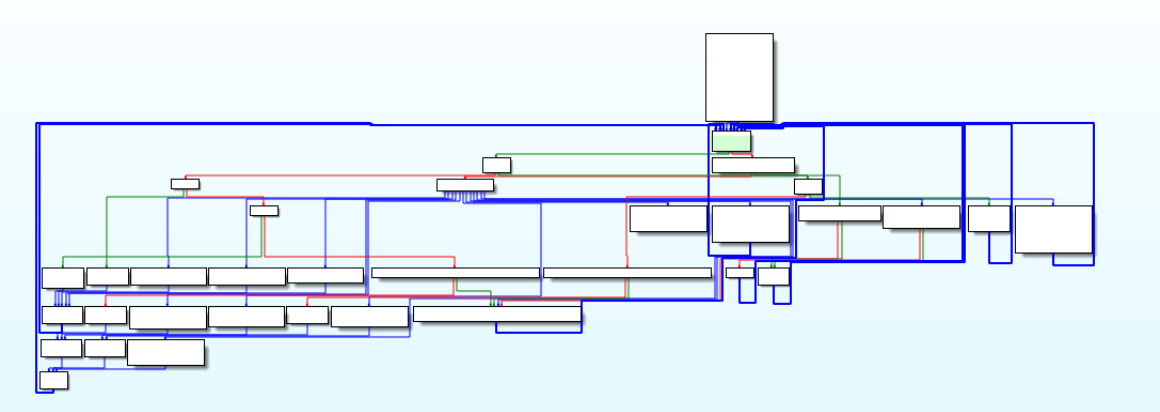

Figure 27: Graphe assembleur de run_vm, visuellement caractéristique d'une obfuscation par VM

L'obfuscation par machine virtuelle est une technique permettant de rendre plus difficile la rétro-ingénierie. Le fonctionnement est le suivant : le programmeur implémente un processeur basique qui va exécuter un bytecode[14] situé quelque part dans la mémoire de l'exécutable. Le jeu d'instruction implémenté par ce processeur virtuel peut être plus ou moins complexe. Le switch se fait sur l'opcode[15] de l'instruction à exécuter pour déterminer quelle action effectuer. La boucle tourne tant qu'il reste des instructions à exécuter dans le bytecode. La difficulté pour reverser ce genre de programme réside dans la compréhension du jeu d'instructions du processeur, de situer le bytecode en mémoire, de découvrir comment sont implémentés les registres et les instructions. Une fois ces étapes accomplies, on peut écrire un désassembleur pour reverser le code effectivement exécuté.

Heureusement, le décompilateur d'IDA facilite grandement la rétro-ingénierie de cette fonction. Pour chaque opcode, l'opération à effectuer est claire et facilement compréhensible. Ce processeur virtuel possède 256 registres, chacun d'une taille d'un octet qui sont situés sur la pile de la fonction **run_vm**.

Pour reverser cette machine virtuelle, j'ai choisi d'implémenter un émulateur en python qui affichera au fur et à mesure les instructions qu'il exécute.

J'ai d'abord codé un petit wrapper en C qui utilise le loader dynamique pour charger **guest.so** et retrouver les adresses des fonctions **useVM**, **getIdent** et **getPerms**. Maintenant qu'on connaît les types d'arguments demandés par les fonctions, on peut

[14] Le bytecode est une suite d'instructions compréhensibles par ce processeur virtuel

[15] Partie de l'instruction qui indique l'opération à effectuer

les appeler directement depuis notre wrapper pour vérifier son bon fonctionnement. On peut également débugger la fonction **run_vm** avec gdb pour s'assurer que les actions effectuées par les opcodes sont bien celles que l'on pense.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dlfcn.h>

typedef void * (*getPerms_t)(void * a1);
typedef void * (*getIdent_t)(void * a1);
typedef void * (*useVM_t)(void * a1, void * a2);

// Gen the token corresponding to the resource
void load_vm(unsigned long resource) {
    unsigned long in[2];
    unsigned long out[3];
    void * lib = dlopen("/path_to/guest.so", 1);
    getPerms_t getPerms = (getPerms_t)dlsym(lib, "getPerms");
    getIdent_t getIdent = (getIdent_t)dlsym(lib, "getIdent");
    useVM_t useVM = (useVM_t)dlsym(lib, "useVM");
    getIdent(&out[2]);
    in[0] = resource;
    getPerms(&in[1]);
    useVM(in, out);
    for (int i = 0; i < 20; i++) {
        printf("%02x", (unsigned char)((char *)out)[i]);
    }
    printf("\n");
}

int main(int argc, char **argv) {
    if (argc != 2) {
        printf("No resource name\n");
        return -1;
    }
    load_vm(atoll(argv[1]));
    return 0;
}
```

Un autre écueil nous attend cependant : si on extrait le bytecode directement depuis l'ELF pour pouvoir l'exécuter dans notre émulateur, on se rend compte qu'il n'est pas identique à celui exécuté par le processeur virtuel. Le bytecode est modifié au moment du chargement en mémoire de la bibliothèque. Pour extraire le vrai bytecode, j'ai utilisé un peu de scripting gdb. Gdb utilise le wrapper pour charger la librairie en mémoire puis extrait la portion de la mémoire du processus qui contient le bytecode décodé.

```
file wrapper
set environment LD_PRELOAD ./guest.so
b * load_vm + 52
commands
dump binary memory bytecode 0x7ffff7c2b030 0x7ffff7c2b030+3800000
quit
end
r 0
```

Une fois l'émulateur/désassembleur fonctionnel on peut commencer à reverser les fonctions ! Voici le code désassemblé de **getPerms**:

```
0x0 :    JMP 0x1f6b32
0x1f6b32 :    LOAD r0, option[0x0] (0x1)
0x1f6b35 :    STORE 0x0, r1
0x1f6b38 :    CMP r0 (0x1), r1 (0x0)
      JNE 0x1f6b44
0x1f6b44 :    STORE 0x1, r1
0x1f6b47 :    CMP r0 (0x1), r1 (0x1)
      JNE 0x1f6b53
0x1f6b4e :    JMP 0x175f99
0x175f99 :    STORE 0xff, r0
0x175f9c :    STORE r0 (0xff), out[0x0]
0x175f9f :    STORE 0xff, r0
0x175fa2 :    STORE r0 (0xff), out[0x1]
0x175fa5 :    STORE 0xff, r0
0x175fa8 :    STORE r0 (0xff), out[0x2]
0x175fab :    STORE 0xff, r0
0x175fae :    STORE r0 (0xff), out[0x3]
0x175fb1 :    STORE 0xff, r0
0x175fb4 :    STORE r0 (0xff), out[0x4]
0x175fb7 :    STORE 0xff, r0
0x175fba :    STORE r0 (0xff), out[0x5]
0x175fbd :    STORE 0xff, r0
0x175fc0 :    STORE r0 (0xff), out[0x6]
0x175fc3 :    STORE 0xff, r0
0x175fc6 :    STORE r0 (0xff), out[0x7]
0x175fc9 :    RET :  0
```

Au début, un check est fait sur l'option pour déterminer quelle fonction exécuter. Comme on l'a vu précédemment, la fonction stocke simplement la valeur `ffffffffffffffff` dans le pointeur de sortie.

Maintenant, le code de **getIdent**. On a vu grâce au hooking que cette fonction renvoie régulièrement des valeurs différentes, peut-être quelque chose basé sur le timestamp :

```
0x0 :    JMP 0x1f6b32
0x1f6b32 :    LOAD r0, option[0x0] (0x2)
0x1f6b35 :    STORE 0x0, r1
0x1f6b38 :    CMP r0 (0x2), r1 (0x0)
      JNE 0x1f6b44
0x1f6b44 :    STORE 0x1, r1
0x1f6b47 :    CMP r0 (0x2), r1 (0x1)
      JNE 0x1f6b53
0x1f6b53 :    STORE 0x2, r1
0x1f6b56 :    CMP r0 (0x2), r1 (0x2)
      JNE 0x1f6b62
0x1f6b5d :    JMP 0x2b93df
0x2b93df :    STORE 0x7e, r0
0x2b93e2 :    STORE r0 (0x7e), out[0x0]
0x2b93e5 :    STORE 0x74, r0
0x2b93e8 :    STORE r0 (0x74), out[0x1]
0x2b93eb :    STORE 0x8a, r0
0x2b93ee :    STORE r0 (0x8a), out[0x2]
0x2b93f1 :    STORE 0x60, r0
0x2b93f4 :    STORE r0 (0x60), out[0x3]
0x2b93f7 :    RET :  0
```

La fonction stocke une valeur hardcodée dans la sortie... Là, on comprend pourquoi VLC télécharge guest.so à chaque démarrage. Les valeurs hardcodées dans le bytecode changent à chaque téléchargement. La valeur hardcodée dans **getIdent** correspond effectivement au timestamp du téléchargement. Cependant, il y a encore plus fourbe. Les opcodes du processeur sont différents à chaque téléchargement... On ne peut évidemment pas reverser tous les opcodes à chaque fois qu'on télécharge guest.so, ça prendrait beaucoup trop de temps.

J'ai donc pris le pari que même si les opcodes étaient toujours différents, le code exécuté serait identique. En regardant l'ordre d'apparition des différents opcodes dans les fonctions stockées dans **run_vm**, on pourra déterminer les opcodes dynamiquement en faisant tourner l'émulateur. Cette approche s'est avérée payante et on peut retrouver les opcodes grâce à l'émulateur.

Par exemple, les 4 premières instructions de **getIdent** sont toujours JMP, LOAD, STORE, CMP. Si on émule **getIdent**, la première fois qu'on rencontre un opcode inconnu on peut déterminer que c'est l'opcode de JMP. La deuxième fois, c'est l'opcode de LOAD et ainsi de suite.

Maintenant, on peut commencer à s'intéresser à la fonction **useVM** qui génère notre token pour demander une clé liée à une ressource à partir de son ident et des permissions du client. Je ne montre volontairement pas le code désassemblé ici car il est beaucoup trop long (709 lignes).

4.3 Etudier l'algorithme de chiffrement

Cette fonction implémente un algorithme de chiffrement inconnu (pour moi en tout cas). Les premières opérations vérifient que les permissions du client dans in sont bien à `ffffffffffffffff`. Ensuite les 8 octets indiquant l'ident de la ressource dont on

veut demander la clé sont chiffrés via une suite de permutations et de xor pour générer un bloc chiffré de 16 octets.

Cet algorithme peut se découper en 3 types de blocs d'assembleur :

- Les blocs XOR, toujours ces registres et cet ordre d'instructions

```
0x2b9f64 :    XOR r16, r16, r21 (0x5d)
0x2b9f68 :    XOR r17, r17, r22 (0xe0)
0x2b9f6c :    XOR r18, r18, r23 (0x1f)
0x2b9f70 :    XOR r19, r19, r20 (0x21)
0x2b9f74 :    XOR r20, r20, r18 (0x1a)
0x2b9f78 :    XOR r21, r21, r19 (0x82)
0x2b9f7c :    XOR r22, r22, r16 (0xcb)
0x2b9f80 :    XOR r23, r23, r17 (0x3c)
0x2b9f84 :    XOR r16, r16, r23 (0x61)
0x2b9f88 :    XOR r17, r17, r20 (0xfa)
0x2b9f8c :    XOR r18, r18, r21 (0x9d)
0x2b9f90 :    XOR r19, r19, r22 (0xea)
0x2b9f94 :    XOR r20, r20, r19 (0xf0)
0x2b9f98 :    XOR r21, r21, r16 (0xe3)
0x2b9f9c :    XOR r22, r22, r17 (0x31)
0x2b9fa0 :    XOR r23, r23, r18 (0xa1)
```

- Les blocs substitution matrice de cette forme (les registres peuvent différer) :

```
0xc3fc7 :    MOV r0, data[256 * r0 + 0x37d90b + r20] (0x76)
0xc3fcf :    MOV r1, data[256 * r1 + 0x37d90b + r21] (0xac)
0xc3fd7 :    MOV r2, data[256 * r2 + 0x37d90b + r22] (0xf3)
0xc3fdf :    MOV r3, data[256 * r3 + 0x37d90b + r23] (0x8c)
0xc3fe7 :    MOV r4, data[256 * r4 + 0x37d90b + r16] (0x3)
0xc3fef :    MOV r5, data[256 * r5 + 0x37d90b + r17] (0x82)
0xc3ff7 :    MOV r6, data[256 * r6 + 0x37d90b + r18] (0x9a)
0xc3fff :    MOV r7, data[256 * r7 + 0x37d90b + r19] (0x69)
```

- Les blocs substitution array de cette forme (les registres peuvent différer) :

```
0x2b9f2c :    MOV r16, data[0x41b5e + r0] (0xfe)
0x2b9f33 :    MOV r17, data[0x41db7 + r1] (0x76)
0x2b9f3a :    MOV r18, data[0x1f710a + r2] (0xc3)
0x2b9f41 :    MOV r19, data[0x289607 + r3] (0x24)
0x2b9f48 :    MOV r20, data[0x124b38 + r4] (0x5)
0x2b9f4f :    MOV r21, data[0x30a84 + r5] (0xa3)
0x2b9f56 :    MOV r22, data[0xc379f + r6] (0x96)
0x2b9f5d :    MOV r23, data[0x207542 + r7] (0xdc)
```

Ces trois blocs "élémentaires" peuvent se rassembler en 2 types de blocs basiques qui peuvent eux-même se rassembler en 2 blocs généraux qui sont répétés pour le chiffrement.

Pseudo-code :

```
bloc_arraysub1: change regs 16 to 24 using regs 8 to 16
bloc_arraysub2: change regs 16 to 24 using regs 0 to 8
bloc_arraysub3: change regs 8 to 16  using regs 16 to 24
bloc_matsub1:   change regs 0 to 8   using regs 0 to 8  and 16 to 24
bloc_matsub2:   change regs 8 to 16  using regs 8 to 16 and 16 to 24
bloc_xor:       change regs 16 to 24 using regs 16 to 24

start_bloc:
    bloc_arraysub2
    bloc_xor
    bloc_arraysub3

basic_bloc1:
    bloc_arraysub1
    bloc_xor
    bloc_matsub1

basic_bloc2:
    bloc_arraysub2
    bloc_xor
    bloc_matsub2

bloc1:
    basic_bloc1
    basic_bloc2
    basic_bloc1
    basic_bloc2
    basic_bloc1

bloc2:
    multiple matsub
    basic_bloc2

encrypt:
    start_bloc
    bloc1
    bloc2
    bloc1
    bloc2
    bloc1
```

Le bloc chiffré retourné est `registers[0:16]`.

Pour vérifier ma bonne compréhension de l'algorithme, je l'ai ré-implémenté en python. Grâce à l'émulateur codé précédemment, je peux extraire du bytecode les array

et les matrices utilisés pour les substitutions pour les utiliser dans mon implémentation de l'algorithme. J'ai aussi implémenté l'algorithme de déchiffrement mais ça ne m'a servi à rien à part me rendre compte de ce qui suit :

On remarque que le bloc **start_bloc** a exactement la même structure que le bloc **basic_bloc2**, excepté pour la dernière partie où la substitution matrice a été remplacée par une substitution array.

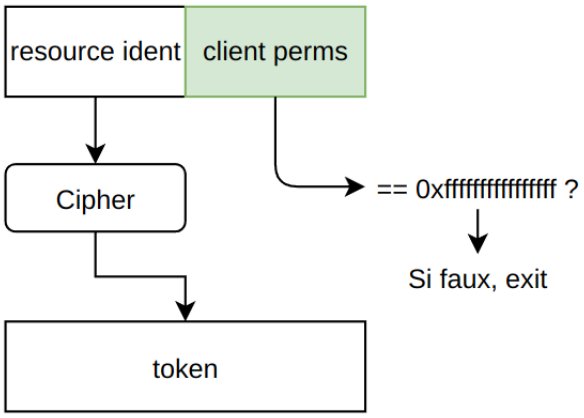

Figure 28: Utilisation des données par l'algorithme

Mon raisonnement a été le suivant :

- L'algorithme n'utilise que 8 octets de l'entrée qu'il utilise pour initialiser les registres 0 à 8
- Le token est un bloc de 16 octets qu'on extrait des registres 0 à 16 à la fin de l'algorithme
- La substitution array dans **start_bloc** modifie les même registres que la substitution matrice dans **basic_bloc2** qui a la même structure que **start_bloc**
- Le seul moment où sont utilisées les permissions du client est au tout début de **useVM** pour vérifier qu'elles ont bien la valeur `ffffffffffffffff`

Déduction : cette substitution array dans **start_bloc** est une substitution matrice dont on n'a gardé qu'une seule ligne : la dernière !

Mon hypothèse est que l'algorithme "complet" initialiserai les registres 0 à 8 avec l'ident de la ressource dont on veut demander la clé et les registres 8 à 16 avec les permissions associées. Le bloc initial serait un **basic_bloc2** pour introduire les données de l'utilisateur dans l'algorithme.

Le problème est que nous ne possédons pas la matrice originale qui permettrait de fabriquer des demandes de clés avec des permissions arbitraires. Il faut plus d'informations sur le serveur de clés.

4.4 Les composants du serveur de clés

Dans l'archive DRM.zip récupérée à la fin de l'étape 2 se trouve une machine virtuelle qemu que nous n'avons pas encore analysée. C'est ce que nous allons faire maintenant.

Le système de fichier de la machine virtuelle est une archive cpio que nous pouvons décompresser avec la commande suivante :

```
zcat rootfs.img | cpio -idmv
```

Pour se donner un shell root au démarrage de la machine virtuelle, il faut rajouter les lignes suivantes dans le script `/init` :

```
stty -F /dev/ttyS0 -icrnl -ixon -ixoff -opost -isig
setsid cttyhack setuidgid 0 sh
```

On peut ensuite recompresser l'archive cpio et éditer le script qemu pour utiliser notre nouveau système de fichiers

```
# en se plaçant à la racine du système de fichiers :
find . | cpio -o -H newc | gzip -9 > new_rootfs.img
```

On trouve deux choses intéressantes à l'intérieur du système de fichier :

- Un ELF compilé statiquement dans `/home/sstic` : **service**
- Un module noyau dans `/lib/modules` : **sstic.ko**. Le script `/init` nous apprend que ce module est chargé dès le démarrage du système.

On commence par s'intéresser à **service**. Le binaire est strippé et compilé statiquement, il y aura donc beaucoup de fonctions et aucun symbole ce qui peut rendre la rétro-ingénierie complexe.

Pour trouver la fonction **main**, on se rend au point d'entrée du programme (la fonction **_start**). Les binaires sous Linux commencent toujours par appeler `__libc_start_main` avec comme premier argument l'adresse de la fonction **main**. On retrouve très facilement ce pattern grâce à IDA, ce qui nous permet de localiser la fonction **main**.

Ce programme fait beaucoup de vérifications sur la valeur de retour des fonctions qu'il appelle et affiche des messages d'erreur en conséquence. C'est du pain béni pour la rétro-ingénierie puisque ça nous permet d'identifier très facilement les fonctions de la libc qui sont appelées.

La fonction main effectue les actions suivantes :

1. Crée un socket qui attend des connexions sur le port 1337
2. Lors de l'ouverture d'une connexion, envoie la chaîne **STIC**
3. Attend de recevoir 17 octets puis checke que le premier octet est compris entre 0 et 3 inclus
4. En fonction de la valeur du premier octet, appelle une fonction particulière. On appelera ces fonctions **treat_0** pour traiter une requête avec le premier octet valant 0, **treat_1** pour l'octet valant 1, **treat_2** et **treat_3**.
5. Pour les fonctions **treat_{1, 2, 3}**, les données passent d'abord par une autre fonction qu'on appelera **check_request**

Le premier octet de la requête détermine donc quelle fonction sera exécutée sur le serveur de clés.

Ces 4 fonctions fournissent essentiellement une interface avec le module noyau. **service** reçoit des données via le réseau, effectue des vérifications dessus puis interagit avec le module noyau via des **ioctl**[16] et une implémentation custom de **mmap** du module noyau qu'on regardera plus en détail plus tard.

On a donc :

1. **treat_0** qui est un oracle de déchiffrement pour le token envoyé (le déchiffrement est effectué sur le module noyau) et qui permet de vérifier que l'ident du client (qui est le timestamp du téléchargement de guest.so on le rappelle) date de moins d'une heure.

2. **treat_1** qui déchiffre le token envoyé, extrait les permissions et vérifie que le client a bien les bonnes permissions pour demander la clé associée à la ressource puis récupère la clé via le module noyau

3. **treat_2** qui permet d'exécuter du code envoyé par le client en mode debug (on ne sait pas encore ce que ça veut dire)

4. **treat_3** qui permet d'exécuter du code de la même façon que **treat_2** mais sans le mode debug

La fonction de vérification avant **treat_{1, 2, 3}** vérifie simplement que le client a les permissions pour effectuer l'opération correspondante. La demande de clé requiert des permissions inférieures à `ffffffffffffff`, l'exécution en mode debug : inférieures à `100` et l'exécution : inférieures à `10`.

Maintenant qu'on comprend un peu mieux le fonctionnement du serveur de clé, il va falloir s'attaquer à l'algorithme de chiffrement/déchiffrement pour forger un token qui une fois déchiffré donnera des permissions de `0`, ce qui nous permettra d'effectuer toutes les actions possibles offertes par **service** et demander les clés du dossier **ambiance**.

4.5 De retour sur l'algorithme de chiffrement

Je m'excuse d'avance pour mon manque de vocabulaire technique auprès de tous les cryptographes qui liront cette partie.

On sait maintenant que le serveur de clés nous offre un oracle de déchiffrement. La question est : **Comment tirer profit de cet oracle pour générer un token avec des permissions arbitraires ?**

On a vu dans la fin de la partie 4.3 que des informations avaient été volontairement retirées pour la dernière opération de substitution dans **start_bloc**. Ce qui aurait dû être des matrices a été transformé en arrays pour éviter à un utilisateur avec la librairie **guest.so** de se donner des permissions inférieures à `ffffffffffffff`.

Cependant, au déchiffrement, le serveur de clé n'a aucun moyen de déterminer si l'utilisateur lui envoyant le token l'a calculé depuis la librairie **guest.so** ou **auth.so**

[16] Input Output Control, un appel système pour interagir avec un device pour effectuer des opérations qui ne peuvent pas être effectuées avec un appel système classique

qui (on suppose) peut chiffrer pour des permissions plus précises puisqu'on ne peut y accéder qu'après une authentification HTTP sur le serveur multimédia et intègre donc la matrice de substitution en totalité. Le serveur utilisera donc la matrice complète pour déchiffrer le token.

Heureusement, avoir implémenté l'algorithme en python permet d'y appliquer des petites modifications! **start_bloc** n'apporte aucune modification sur les registres 0 à 8 qui contiennent l'ident de la ressource dont on veut demander la clé. Apporter des modifications sur les registres 8 à 24 dans ce bloc n'affectera donc pas le déchiffrement des 8 premiers octets. On peut agir directement sur les permissions sans toucher à l'ident de la ressource demandée.

Les substitutions dans les matrices sont indexées par des registres d'un octet qui peuvent prendre des valeurs allant jusqu'à 256. Ces matrices ont donc une dimension de 256*256 octets. Ce sont aussi des matrices "sudoku", c'est à dire que sur une même ligne tous les octets sont différents et idem sur une même colonne.

On va exploiter cette propriété "sudoku" de la matrice de **start_bloc** et le fait que les blocs ident et perms ne sont pas encore mélangés grâce à l'oracle de déchiffrement et notre algorithme implémenté en python. Pour cela, on va venir se placer entre le **bloc_xor** et le **bloc_arraysub3** dans **start_bloc**.

```
start_bloc:
    bloc_arraysub2
    bloc_xor
    hijack regs 16 to 24 value
    bloc_arraysub3
```

Comme les registres 16 à 24 sont utilisés pour indexer dans l'array de array_sub3 (qui détermine la valeur des registres 8 à 16), on va modifier leur valeur ce qui va influer uniquement l'opération d'inversion de la substitution de **start_bloc** lors du déchiffrement. Chaque registre entre 16 et 24 modifié influe uniquement sur un octet de perms (registres 8 à 16).

On peut ainsi mener un bruteforce de 8 * 256 possibilités sur l'oracle de déchiffrement pour trouver une combinaison des registres 16 à 24 qui donne un token qui, une fois déchiffré, donne des permissions de 0.

On peut maintenant bruteforcer les permissions pour demander la clé du dossier ambiance. Un check additionnel dans **service** nous empêche de demander les clés des dossiers **admin** ou **prod** quand le module noyau est configuré en mode debug, ce qui est le cas comme le Readme de cette étape l'a indiqué.

```
[+] Opening connection to challenge2021.sstic.org on port 1337: Done
[*] r16 - r23: [132, 0, 0, 0, 0, 0, 0, 0]
[*] Decrypted token : 5f50189002af116840e4075200e34bec
[*] r16 - r23: [132, 45, 0, 0, 0, 0, 0, 0]
[*] Decrypted token : 5f50189002af116840e4075200004bec
[*] r16 - r23: [132, 45, 30, 0, 0, 0, 0, 0]
[*] Decrypted token : 5f50189002af116840e40752000000ec
[*] r16 - r23: [132, 45, 30, 25, 0, 0, 0, 0]
[*] Decrypted token : 5f50189002af116840e4075200000000
[*] r16 - r23: [132, 45, 30, 25, 198, 0, 0, 0]
[*] Decrypted token : 5f50189002af116800e4075200000000
[*] r16 - r23: [132, 45, 30, 25, 198, 255, 0, 0]
[*] Decrypted token : 5f50189002af11680000075200000000
[*] r16 - r23: [132, 45, 30, 25, 198, 255, 210, 0]
[*] Decrypted token : 5f50189002af11680000005200000000
[*] r16 - r23: [132, 45, 30, 25, 198, 255, 210, 150]
[*] Decrypted token : 5f50189002af11680000000000000000
Getting key...
Code : 03
b'dg\x0b\xf9\xfe\x8a\xc6\xc48\xa9\xc1\xf5Te\x08\x8e'
```

Un petit schéma pour résumer le principe :

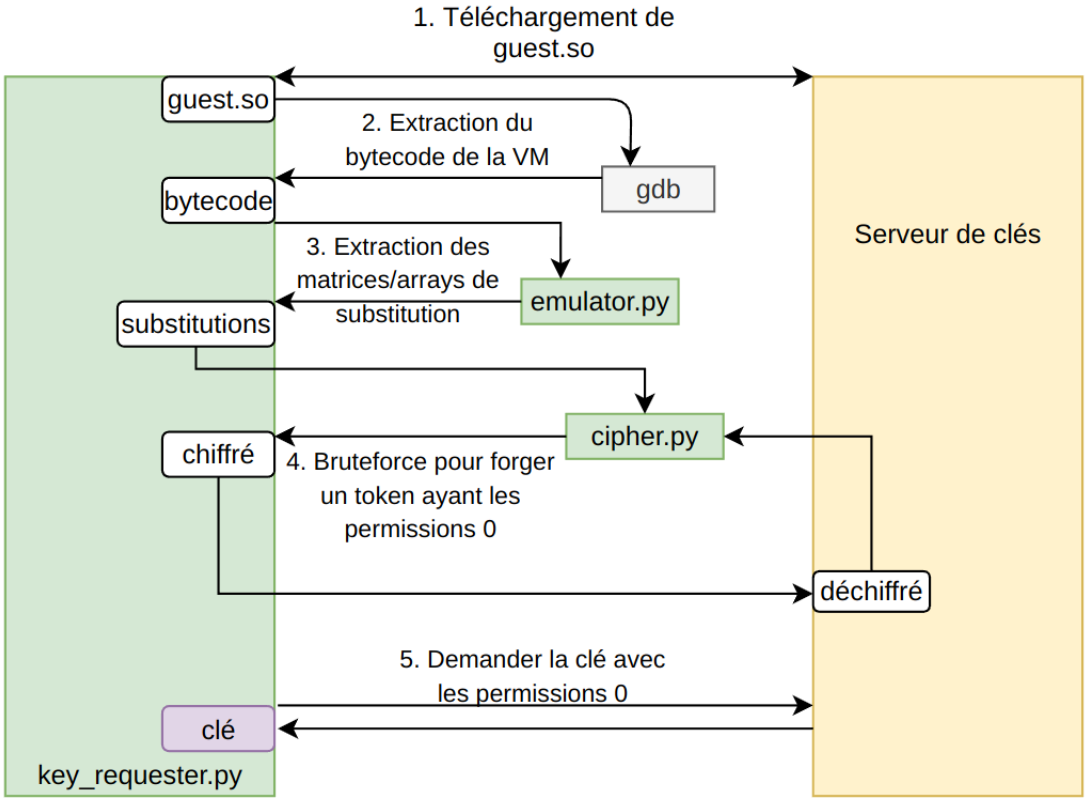

Figure 29: Résumé des opérations pour forger un token avec des permissions à 0

On peut maintenant déchiffrer **ambiance** de la même manière que **rumps**. Ce dossier contient un fichier **info.txt** dont on récupère la clé comme pour **ambiance**.

Déchiffré, ce fichier contient notre troisième flag !

SSTIC{9a5914929b7947afbef39446aafacd35}

5 Etape 4

5.1 Un processeur custom

Grâce aux outils codés dans l'étape 3, nous pouvons forger des token avec des permissions de 0 et nous avons donc accès à toutes les fonctionnalités offertes par le serveur de clés.

Il reste deux fonctionnalités que nous n'avons pas encore explorées sur le serveur : **execute_code_debug** et **execute_debug**.

execute_code_debug reçoit plusieurs données depuis le réseau :

1. Un entier de 64 bits : **code_size**
2. **code_size** octets : **code**
3. Un entier de 64 bits : **input_size**
4. **input_size** octets : **input**
5. Un entier de 64 bits : **output_size**

service transmet ensuite ces données au module noyau via son implémentation custom de mmap et déclenche une commande avec un ioctl. Une sortie de debug est ensuite renvoyée via le réseau.

execute_code fait la même chose, excepté qu'il ne reçoit depuis le réseau que 80 octets qu'il stocke dans input. Le code à envoyer au module est stocké en dur dans **service**. Un check est ensuite fait sur la sortie (de 64 octets) renvoyée par le module noyau. Si `sortie[0:48] == ff..ff` et `sortie[48:] == "EXECUTE FILE OK!"`, alors **service** se met en attente pour recevoir un ELF qu'il exécutera et nous renvoie la sortie de l'exécution de ce programme.

Testons **execute_code_debug** en envoyant quelques octets à 0 en code. Voici le résultat renvoyé :

```
debug :
 Bad instruction
regs:
PC : 1000
R0 : 00000000000000000000000000000000
R1 : 00000000000000000000000000000000
R2 : 00000000000000000000000000000000
R3 : 00000000000000000000000000000000
R4 : 00000000000000000000000000000000
R5 : 00000000000000000000000000000000
R6 : 00000000000000000000000000000000
R7 : 00000000000000000000000000000000
RC : 00000000000000000000000000000000
stack: []
```

Cette sortie nous donne beaucoup d'informations ! Visiblement, le code que nous envoyons est exécuté sur un processeur custom avec lequel on interface via le module noyau. La sortie de debug nous donne l'état des registres du processeur et de la pile à la fin de l'exécution du programme.

Ce processeur possède 8 registres généraux, **R0** à **R7** de 128 bits. On suppose que **PC** est le Program Counter = le pointeur d'instruction du processeur qui commence à 0x1000. **RC** semble être un registre spécial de 128 bits aussi dont nous ne connaissons pas encore l'utilité.

Deuxième essai, on envoie le code qui est exécuté dans **execute_code** et on obtient la sortie suivante :

```
debug :
 regs:
PC : 10ac
R0 : 00000000000000000000000000000000
R1 : 00000000000000000000000000000000
R2 : 00000000000000000000000000000000
R3 : 00000000000000000000000000000000
R4 : 00000000000000000000000000000000
R5 : 00000000000000000000000000000000
R6 : 01010101010101010101010101010101
R7 : 00000000000000000000000000000000
RC : 00000000000000000000000000000000
stack: []
```

Cette fois, pas de message d'erreur comme quoi l'instruction est invalide. Le PC s'est incrémenté de 172 et la valeur de **R6** n'est plus à 0.

Le but va donc être d'extraire et de comprendre le jeu d'instructions de ce processeur pour écrire un désassembleur et reverser le code exécuté par **execute_code** que nous appelerons **req3code**. Nous pourrons ainsi retrouver l'entrée de 80 octets qui donne la sortie attendue de 64 octets pour envoyer un ELF qui sera exécuté sur le serveur de clés !

D'abord, on va essayer de voir si on peut trouver la taille d'une instruction. En tronquant **req3code** à différentes longueurs, on détermine que les instructions sont de taille constante : 32 bits.

On peut implémenter une forme de breakpoint en remplaçant une instruction par `\x00\x00\x00\x00` qui fera planter le processeur puisque c'est une instruction invalide comme on l'a vu lors du premier essai où on a envoyé la suite de 0. Nous pouvons ainsi récupérer l'état des registres à différents points du programme pour voir l'effet de différentes instructions sur les registres.

Par essai/erreur, en regardant la représentation binaire des instructions, en flippant certains bits des instructions, on finit par déterminer le format du jeu d'instructions du processeur.

Ce processeur est un processeur SIMD[17]. Les registres peuvent être considérés par

[17] Single Instruction, Multiple Data

les instructions comme un nombre unique de 128 bits, comme deux nombres de 64 bits, 4 nombres de 32 bits, 8 nombres de 16 bits, 16 nombres de 8 bits.

Le registre **RC** est utilisé par les opérations de comparaison. Un comparaison change la valeur du registre **RC** tandis que les opérations conditionnelles lisent la valeur de ce registre pour décider d'exécuter l'instruction ou non.

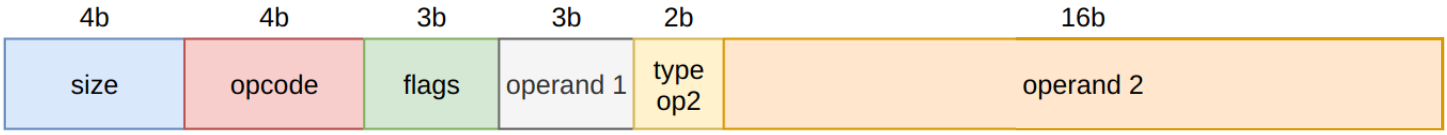

Figure 30: Le format d'une instruction du processeur custom

- **size** indique sur quelle taille de données l'opération travaille (128 bits, 64 bits, 32 bits, 16 bits ou 8 bits)
- **opcode** indique l'opération à effectuer (ADD, MOV, XOR...)
- **flags** contient la condition pour exécuter ou non l'instruction (à la manière de l'assembleur ARM)
- **operand 1** première opérande (numéro du registre, entre 0 et 7)
- **type op2** indique comment interpréter **operand 2** : valeur immédiate, registre, registre déréférencé, valeur immédiate déréférencée
- **operand 2** seconde opérande

Les données sont mappées à l'adresse `0x0000` dans l'espace d'adressage du processeur, le **code** est mappé à l'adresse `0x1000` , l'**input** à `0x2000`, l'**output** à `0x3000`.

On a maintenant toutes les informations nécessaires pour écrire un désassembleur pour ce processeur. Voici le début de **req3code** :

```
0x1000: oload       R0, [0x2040]
0x1004: omov        R6, #0x0000
0x1008: bcmp        R6, #0x0010
0x100c: bjmpeq      R0, #0x1024
0x1010: bcmp        R0, R6
0x1014: bjmpbeq     R0, #0x101c
0x1018: ojmp        R0, #0x10ac
0x101c: badd        R6, #0x0001
0x1020: ojmp        R0, #0x1008
0x1024: oload       R1, [0x0200]
0x1028: wcmpbeq     R0, R1
0x102c: wjmpb       R0, #0x10ac
0x1030: dcmpg       R0, [0x0210]
0x1034: djmpeq      R0, #0x103c
0x1038: ojmp        R0, #0x10ac
0x103c: qcmpb       R0, [0x0220]
0x1040: qjmpb       R0, #0x10ac
0x1044: oxor        R5, R5
```

Un préfixe en **o** indique qu'on travaille sur 128 bits, **q** : 64 bits, **d** : 32 bits, **w** : 16 bits, **b** : 8 bits.

5.2 Encore de la crypto

On peut maintenant commencer à reverser **req3code**.

La première partie charge les 16 derniers octets de **input** pour mener une série de vérifications dessus. Si toutes les vérifications passent, la suite du code est xorée avec cette valeur.

La suite du désassemblage est donc invalide, il faut d'abord extraire cette clé de déchiffrement pour désassembler correctement le reste du code.

Certains checks utilisent des valeurs stockées dans la section de données pour des comparaisons. Pour extraire ces valeurs, j'ai codé un assembleur qui me permet d'écrire mon propre code dans le langage du processeur pour extraire les valeurs stockées dans la section de données.

La clé xor : `0xe,0x3,0x5,0xa,0x8,0x4,0x9,0xb,0x0,0xc,0xd,0x7,0xf,0x2,0x6,0x1`
On peut xorer le reste du code avec cette clé pour obtenir le bon code à désassembler.

Le reste du code implémente un algorithme de chiffrement qui chiffre les 64 premiers octets d'**input**. Ce bloc chiffré de 64 octets est stocké dans **output**. C'est ce bloc chiffré qui est comparé à `ffff...ffffEXECUTE FILE OK!` dans **service**. Il faut donc trouver la valeur d'entrée qui une fois passée dans cet algorithme donnera la bonne valeur de sortie.

J'ai réimplémenté cet algorithme en python. Une fois que je me suis assuré que mon implémentation était correcte en comparant la sortie de mon implémentation et de celle du processeur, j'ai implémenté les opérations pour inverser l'algorithme.

On retrouve donc l'entrée correspondant à la sortie attendue. Ce bloc commence par `expand 32-byte k`. En regardant sur google, j'ai découvert que cette valeur magique m'amenait à l'algorithme de chiffrement ChaCha20 dont on peut retrouver l'implémentation sur Wikipédia[18] qui correspond effectivement aux opérations effectuées par **req3code**...

On a maintenant la valeur magique à envoyer au serveur de clés pour pouvoir ensuite envoyer notre propre exécutable ELF qui sera exécuté sur la machine !

Nous pouvons maintenant interagir directement avec le module noyau sans passer par **service** et ses vérifications sur les données envoyées. On peut mimer le comportement de **service** pour interagir avec le module noyau et essayer de récupérer les clés des dossiers **admin** et **prod** que **service** nous empêchait de récupérer.

Le module semble bloquer l'accès à la clé du dossier **prod**, probablement à cause du mode debug dont le Readme parle mais nous pouvons récupérer la clé du dossier **admin**.

[18] `https://en.wikipedia.org/wiki/Salsa20`

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>

void get_key(int fd, unsigned long res) {
    unsigned long resource[3];
    resource[0] = res;

    unsigned long ret = ioctl(fd, 0xC0185304, &resource);
    printf("Resource : %llx -> ", resource[0]);
    for (int i = 0; i < 16; i++) {
        printf("%02x", (unsigned char)((char *)resource)[i+8]);
    }
    printf("\n");
    printf("ret : %llx\n", ret);
}

int main() {
    int fd = open("/dev/sstic", 2);
    printf("admin : \n");
    get_key(fd, 0x75edff360609c9f7);

    printf("prod : \n");
    get_key(fd, 0xd603c7e177f13c40);
}
```

```
[+] Opening connection to challenge2021.sstic.org on port 1337: Done
[*] r16 - r23: [204, 0, 0, 0, 0, 0, 0, 0]
[*] Decrypted token : 00000000000000003636a9e9009591e1
            [...]
[*] r16 - r23: [204, 6, 217, 39, 159, 159, 165, 172]
[*] Decrypted token : 00000000000000000000000000000000
Executing get_keys
Code : 0a
Code 2: 0c
out :
 b'---EXEC OUTPUT START---\n'
debug :
 admin :
Resource : 75edff360609c9f7 -> 115e8adf8927887fe629bfd92829c11b
ret : 0
prod :
Resource : d603c7e177f13c40 -> b9ff400000000000e629bfd92829c11b
ret : ffffffffffffffff
---EXEC OUTPUT END---
```

Dans le dossier **admin** se trouvent plusieurs vidéos, le titre de l'une de ces vidéos est notre 4e flag.

SSTIC{377497547367490298c33a98d84b037d}

6 Etape 5

6.1 Découverte de la cible

Nous voici rendus à la cinquième et dernière étape ! Il ne reste plus qu'un composant du système que nous n'avons pas étudié en détail : le module noyau **sstic.ko**.

Pour reverser un module noyau linux, un bon point de départ est la fonction d'initialisation du module. C'est la première fonction qui est appelée lorsque le module est chargé dans le noyau avec `insmod`. Cette fonction est présente dans tous les modules et est utilisée pour préparer le terrain pour les différents services offerts par le module.

Dans sa fonction d'initialisation, le module initialise un **character device**[19] : `/dev/sstic` auquel il assigne des opérations de fichier.

Ces opérations de fichiers sont stockées dans une structure `file_operations` qui permet de fournir une implémentation pour les appels systèmes spécifiques aux fichiers pour ce **char device**. Le module implémente pour son **char device** les appels systèmes **open**, **close**, **ioctl** et **mmap**.

Le module configure ensuite un **PCI driver**[20] auquel il assigne un nom : `SSTIC Driver` et une fonction **probe**[21]. Cette fonction permet de se connecter au périphérique PCI et de créer une projection mémoire virtuelle qu'on appelera `iomem` pour le BAR[22] 0 du device PCI. Essentiellement, cette fonction initialise la communication avec ce device. Le mode **debug** est ensuite activé sur le device PCI en écrivant la valeur 1 à `iomem + 0x28`.

Enfin, la fonction d'initialisation crée deux caches `sstic_region_cache` et `sstic_session_cache` dédiés à l'allocation d'objets `region` et `session`.

On sait maintenant que le module permet de communiquer avec un périphérique PCI auquel nous n'avons pas accès. On peut communiquer avec le module via le fichier `/dev/sstic` qui implémente les appels systèmes **open**, **close**, **ioctl** et **mmap**.

Le but va donc être de trouver un moyen de désactiver le mode **debug** sur le périphérique PCI pour pouvoir récupérer les clés pour le dossier **prod**.

6.2 Les spécificités du module

6.2.1 Ioctl et structures de données

Le module implémente 6 requêtes ioctl différentes :

- `0xC0185300` : `alloc_region` pour allouer une `region`
- `0xC0185301` : `del_region` pour supprimer une `region`
- `0xC0185302` : `assoc_region` pour associer une `region` à une `session`

[19] Type spécial de fichier utilisé pour exposer une interface avec des fonctionnalité du noyau

[20] Peripheral Component Interconnect, un standard de bus pour interconnecter des composants physiques

[21] Fonction de callback appelée lors de l'enregistrement du driver

[22] Base Address Register

- 0xC0185303 : `submit_command` pour exécuter du code sur le processeur custom de l'étape 4. Au minimum deux `region` doivent êtres associées à la `session` au préalable
- 0xC0185304 : `get_key` pour récupérer une clé liée à une ressource
- 0xC0185305 : `get_debug` renvoie 1 si le périphérique PCI est configuré en mode debug

On va s'intéresser aux objets `region` et `session` pour mieux comprendre leur rôle et fonctionnement. Ces struct peuvent se représenter en C de la façon suivante :

```
// Représente une région de mémoire physique
// Cette structure de données n'a pas de taille prédéfinie
// Je l'ai représentée en C pour la lisibilité
// Elle peut posséder entre 1 et 32 pages
// L'allocation mémoire de cette struct se fait dynamiquement
// en fonction du nombre de pages allouées
struct phy_region {
    void * start;     // addresse de début de la projection en
                      // mémoire virtuelle de cette région
    void * end        // addresse de fin de la projection en
                      // mémoire virtuelle de cette région
    int npages;       // nombre de pages allouées pour cette région
    int cnt_use;      // nombre de références à cette région physique
    struct page * page1;  // pages
    ...
    struct page * pageN;
};

// Tête d'une liste doublement chainée de region
// Allouée depuis le cache sstic_session_cache
// lors de l'appel à open sur /dev/sstic
struct session {
    struct phy_region * associated[4]; // régions phy associées
    void * start;   // pointe sur la première region de la session
    void * end;     // pointe sur la dernière region de la session
};

// Element de la liste session
// Allouée depuis le cache sstic_region_cache
// à chaque appel de alloc_region
struct region {
    struct phy_region * phy_reg;    // région physique associée
    int flags;              // R/W
    int id;                 // identifiant
    void * next;            // pointe sur la region suivante
    void * prev;            // pointe sur la region précédente
};
```

alloc_region prend en argument un nombre de pages `npages`. Cet ioctl alloue une structure représentant une région physique et y stocke `npages` pages physiques **individuelles** allouées avec `alloc_pages_current` (les pages sont individuelles car le bloc alloué par `alloc_pages_current` est splitté avec `split_page` en pages individuelles). Une structure `region` est ensuite allouée depuis le cache correspondant. La `phy_region` est associée à la `region` et la `region` est insérée dans la liste chaînée `session`.

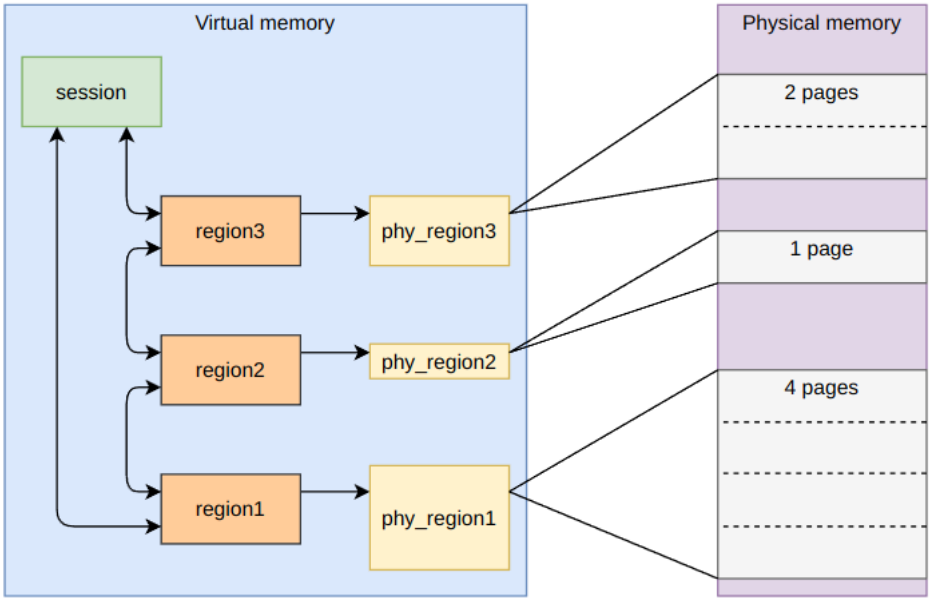

Figure 31: Un schéma de l'utilisation des structures dans le module

del_region prend l'id de la `region` à supprimer en argument. Cet ioctl libère la `region` correspondant à l'id et si le compteur de référence `cnt_use` de la `phy_region` associée vaut 1, la `phy_region` est libérée aussi et les pages associées sont retournées à l'allocateur de pages avec un appel à `put_page` pour chaque page associée.

Ces structures de données sont utilisées pour l'implémentation de l'appel système **mmap** du module. Le module étant un driver pour un périphérique PCI, il permet à un programme utilisateur de communiquer avec ce périphérique via le **character device** `/dev/sstic`. Pour éviter d'innombrables appels `ioctl + copy_from_user` pour assurer la communication entre le périphérique et le programme qui veut l'utiliser, le module implémente **mmap** pour permettre à un programme dans l'espace utilisateur de mapper des régions de mémoires dédiées à la communication avec périphérique.

6.2.2 Quelques généralités sur mmap

Le **char device** du module implémente l'appel système **mmap**. On rappelle la déclaration de **mmap** :

```
void *mmap(void *addr, size_t length, int prot, int flags,
                    int fd, off_t offset);
```

mmap permet de créer une projection mémoire dans l'espace d'adressage virtuel du processus appelant.

- `addr` précise l'adresse virtuelle à laquelle placer la projection
- `length` donne la taille de la projection
- `prot` indique les permissions d'accès (R/W/X) de la projection
- `flags` influe sur les propriétés de la projection (par exemple rendre les modifications faites dans cette projection visibles depuis les autres processus)
- `fd` spécifie le descripteur de fichier du fichier à mapper en mémoire
- `offset` spécifie l'offset auquel démarrer la projection dans le fichier mappé

L'espace d'adressage virtuel d'un processus sous linux est représenté par une struct `mm_struct` qui contient une liste de `vm_area_struct`.

Une struct `vm_area_struct` représente une région contigüe de mémoire possédant les mêmes permissions dans un espace d'adressage. Un appel à **mmap** crée donc une nouvelle `vm_area_struct` qui sera insérée dans la `mm_struct` du processus.

Lorsqu'un module implémente l'appel système **mmap**, il attribue juste un pointeur de fonction au **char device** pour l'entrée mmap dans la structure `file_operations` associée au fichier. Cette fonction a le prototype suivant :

```
int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

Cette fonction est appelée par le noyau (après qu'il ait effectué de nombreuses opérations) quand l'appel système **mmap** est lancé sur le fichier.

Dans le noyau linux, une struct `file` représente un fichier ouvert. L'argument `filp` dans cette fonction représente le fichier qu'on veut mapper en mémoire (ici `/dev/sstic`).

L'argument `vma` représente la zone mappée en mémoire. Le module qui implémente **mmap** peut ainsi effectuer des opérations sur cette struct `vm_area_struct` pour entre autres :

- Modifier son champ `vm_ops` (une struct `vm_operations_struct`) qui - de la même manière que `file_operations` - permet d'implémenter les opérations à effectuer lorsque des évènements particuliers ont lieu sur cette région de mémoire virtuelle (open, split, fault...)
- Y stocker des informations privées liées à une région de mémoire via un pointeur vers une struct dans le champ `vm_private_data`

6.2.3 L'implémentation mmap du module

Pour utiliser **mmap** sur `/dev/sstic`, il faut d'abord allouer une région via l'ioctl **alloc_region**. On peut ensuite mapper la `region` allouée en renseignant la valeur de retour de l'ioctl (qui est `region.id << 12`) dans l'argument `offset` de **mmap**.

Une nouvelle `phy_region` est alors créée. Les pages allouées pour la `region` que le programme veut mapper sont copiées dans la nouvelle `phy_region` qui est stockée dans le champ `vm_private_data` de la struct `vm_area_struct` correspondant à la région mappée.

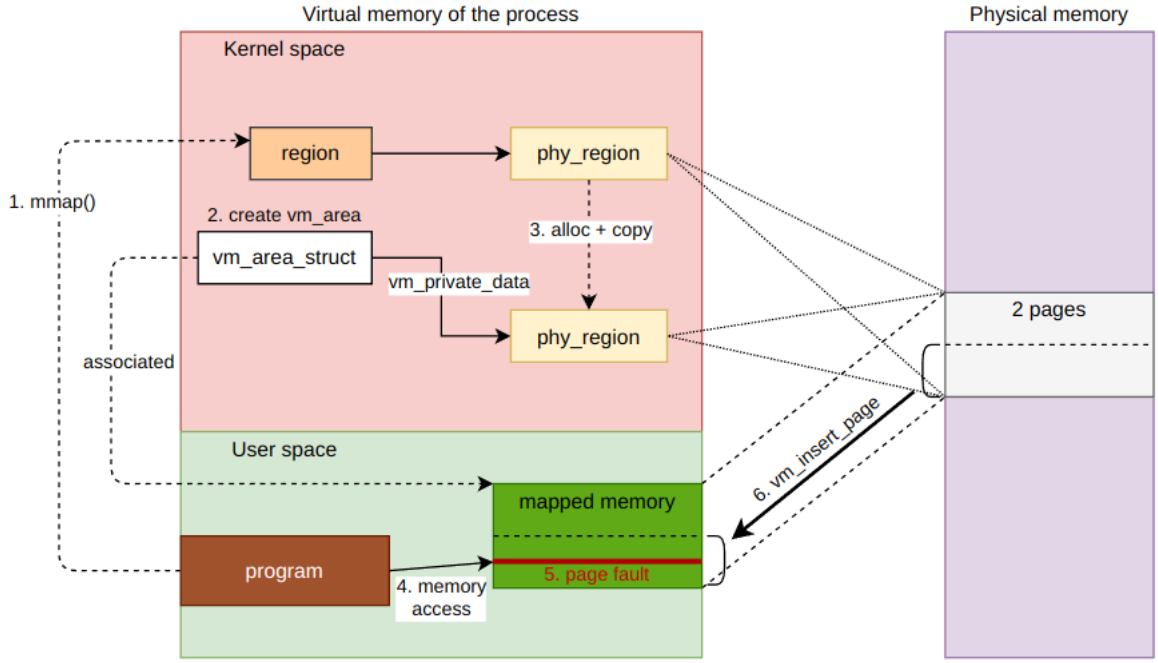

Figure 32: Le fonctionnement du handler mmap et de l'opération vm_fault

Le **mmap** du module implémente les opérations suivantes pour les régions de mémoire mappées:

- `vm_split` quand la région mappée doit être divisée en plusieurs régions (par exemple lors d'un `munmap`, `mprotect` ou `mremap` partiel)
- `vm_open` quand la région mappée est ouverte (par exemple, lors d'un `split` une nouvelle `vm_area_struct` est créée puis ouverte)
- `vm_fault` quand le processus essaye d'accéder à la région mappée mais que la page correspondante n'est pas présente en mémoire
- `vm_close` quand une région mémoire est fermée (par exemple lors d'un `munmap`)

6.3 Exploit mmap FTW

6.3.1 Le bug

Pour traquer l'utilisation des pages allouées et stockées dans les `phy_region`, le module utilise le champ `refcount` des struct `page`.

Le module incrémente ce champ de 1 lors du **mmap** des pages et le décrémente de 1 lors du free d'une `phy_region`.

Ce champ est également manipulé par les fonctions du noyau : `vm_insert_page` incrémente le `refcount` de la page qu'il insère dans la mémoire du processus. Unmapper une région mémoire décrémente le `refcount` des pages qui étaient associées au mapping.

Lors du free d'une `phy_region`, si le `refcount` de la page est à 1 (et passe à 0 lors du décrément), alors la page est retournée à l'allocateur avec `put_page`.

La fonction `vm_open` vérifie si la région mémoire qui vient d'être ouverte a été splittée et réalloue des `phy_region` en fonction. Cependant, il y a un problème dans l'une des vérifications. Pour savoir quelles struct page copier depuis la `phy_region` qui a été splittée dans les nouvelles `phy_region` résultant du split, la fonction `vm_open` compare `vm_area_struct->vm_start` et `phy_region->end` de la `phy_region` associée. `phy_region->end` a l'adresse du split. Nous pouvons contrôler `vma_area_struct->vm_start` grâce à l'appel système **mremap** qui permet de modifier un mapping mémoire précédemment créé avec **mmap**.

En créant un mapping a une adresse prédéfinie grâce au flag `MAP_FIXED` puis en faisant un **mremap** partiel de ce mapping à une adresse mémoire plus élevée grâce aux flags `MREMAP_FIXED | MREMAP_MAYMOVE`, on peut créer un décalage entre les pages qui sont censées êtres dans la `phy_region` associée au mapping et les pages qui y sont effectivement. La vérification ne prend pas en compte le fait que le mapping puisse être déplacé. Ci-dessous une illustration de ce comportement :

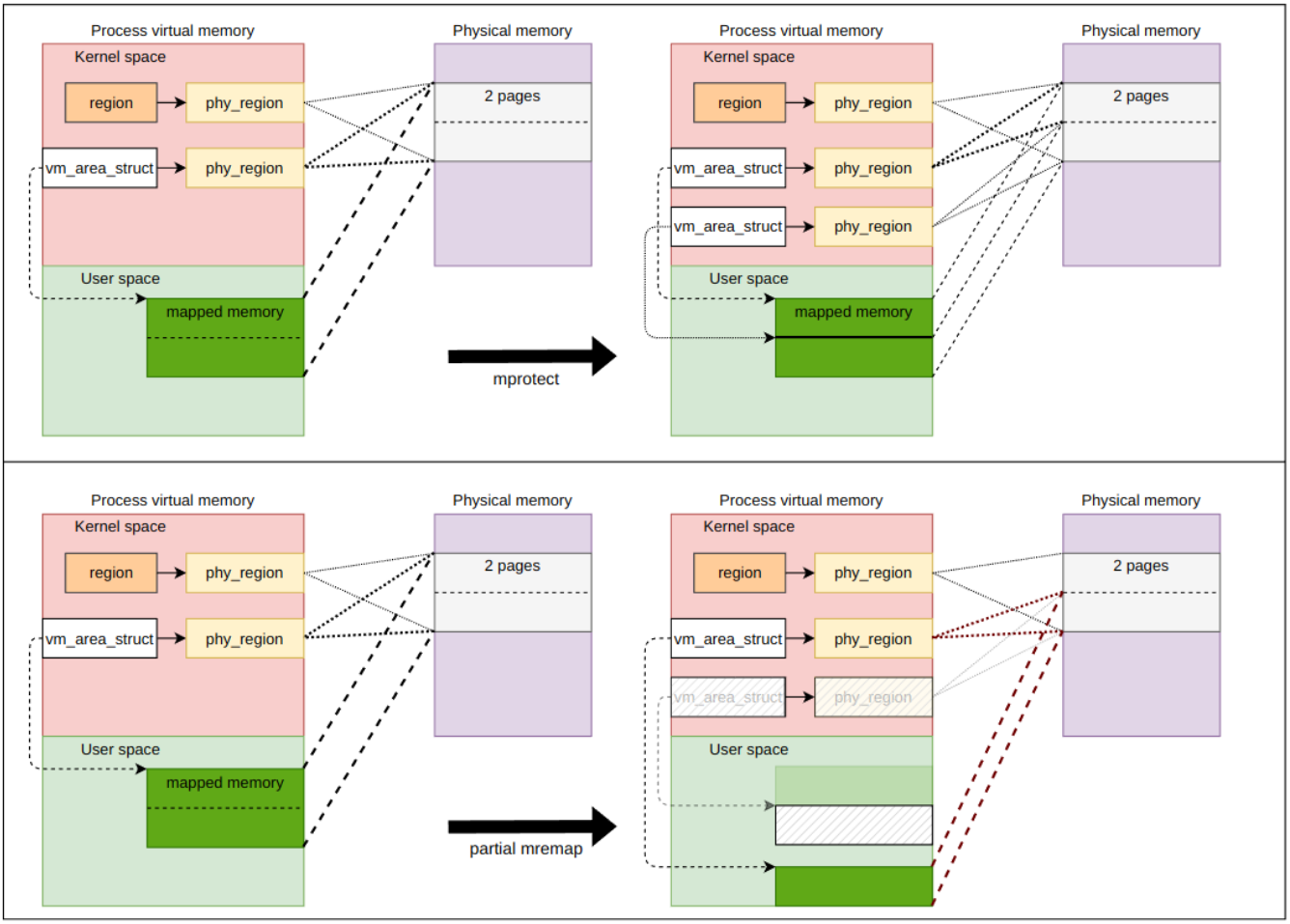

Figure 33: Le bug lors du split avec un mremap partiel

Nous pouvons exploiter ce comportement pour manipuler le `refcount` d'une page et la retourner à l'allocateur de page via l'ioctl `del_region` alors qu'elle est toujours mappée dans l'espace d'adressage virtuel de notre processus. Nous avons ainsi un accès à une page de la mémoire physique qui sera susceptible d'être utilisée par d'autres composants du système. La question est : **Comment rendre cette configuration mémoire exploitable ?**

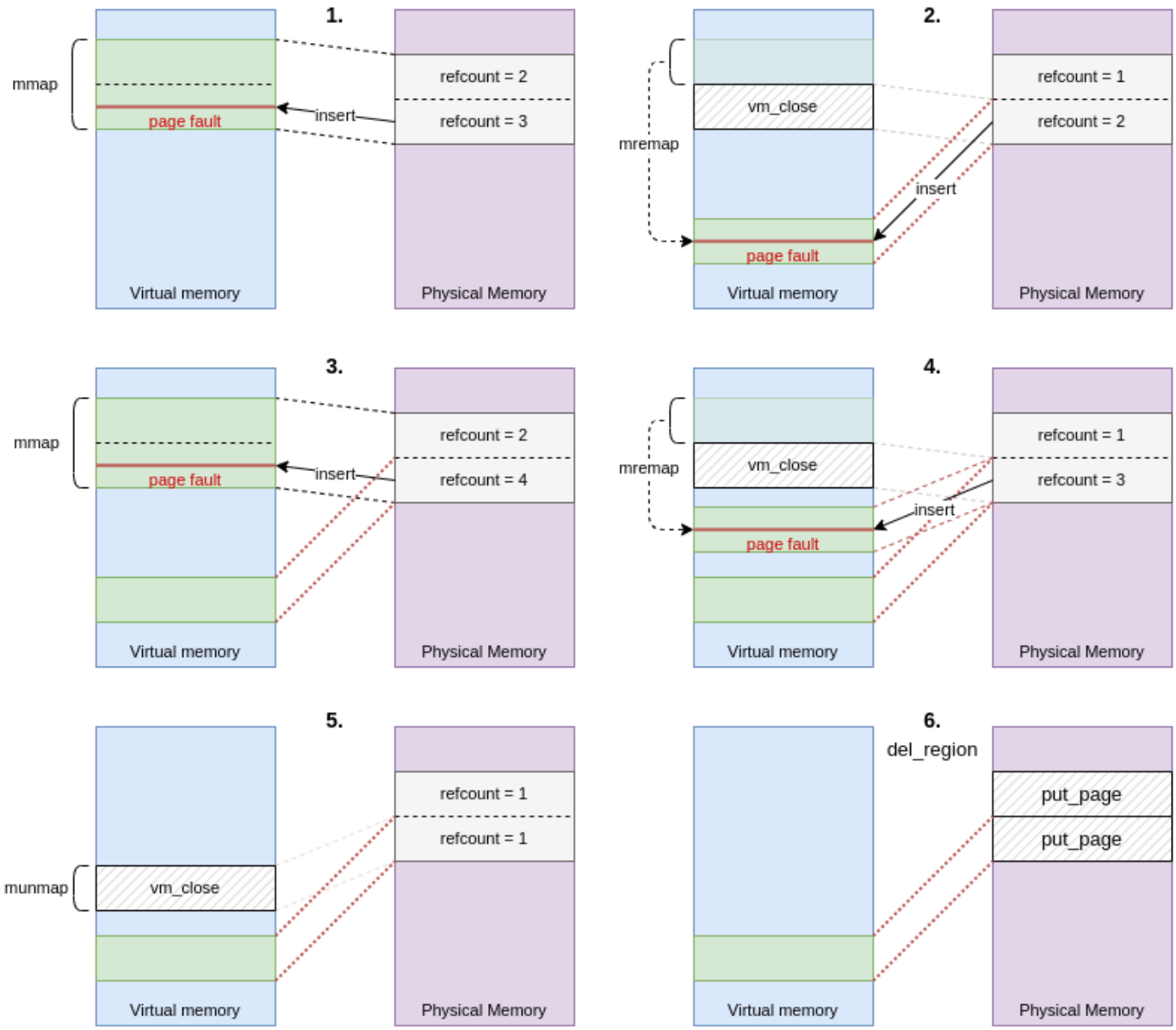

Figure 34: Manipulation du refcount d'une page

Pour l'exploitation, je me suis inspiré de la manière dont l'équipe **Project Zero** de Google a exploité le bug **RowHammer**[23].

Le but est de faire en sorte que le système utilise la page à laquelle nous avons accès pour y stocker une table de page. Nous pouvons ainsi manipuler les entrées dans cette table de page pour accéder à l'intégralité de la mémoire physique.

6.3.2 Accéder à toute la mémoire physique

Pour traduire une adresse virtuelle en une adresse physique, le noyau linux utilise des tables de pages. Ces tables de pages **PT** sont allouées par le noyau.

Chaque PT est stockée dans une page (une page a en général une taille de 4Kb, c'est le cas sur ce système).

[23] `https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`

Les PT peuvent stocker jusqu'à 512 entrées de 64 bits. Chaque entrée **PTE** dans une PT contient l'adresse **physique** d'une page ainsi qu'un set de flags indiquant plusieurs propriétés de la page, notamment ses droits d'accès.

Sachant qu'une page fait 4Kb et qu'une PT peut stocker 512 entrées, une PT permet de couvrir 2Mb de mémoire virtuelle (512*4Kb).

Lorsqu'un processus demande à créer un mapping dans son espace d'adressage virtuel, le noyau regarde si le nouveau mapping peut être adressé grâce à une table de page déjà existante associée au processus. Sinon, il alloue une nouvelle page pour y stocker la table de page qui permettra d'adresser le nouveau mapping.

En créant des mappings à des adresses fixes grâce à `MAP_FIXED` espacés de 2Mb, on force le noyau à allouer une nouvelle table de page à chaque création de mapping.

On va utiliser ce comportement pour faire du **spraying** de tables de pages et faire en sorte que le noyau réutilise la page à laquelle nous avons accès pour y stocker une PT.

On crée donc un fichier rempli d'un pattern reconnaissable qu'on **mmap** plusieurs fois en mémoire à des adresses espacées de 2Mb. Comme on mappe le même fichier à chaque fois, le noyau n'alloue pas de pages supplémentaires pour mapper les mêmes données. A chaque appel à **mmap**, le noyau alloue une page pour y stocker une PT pour adresser le nouveau mapping.

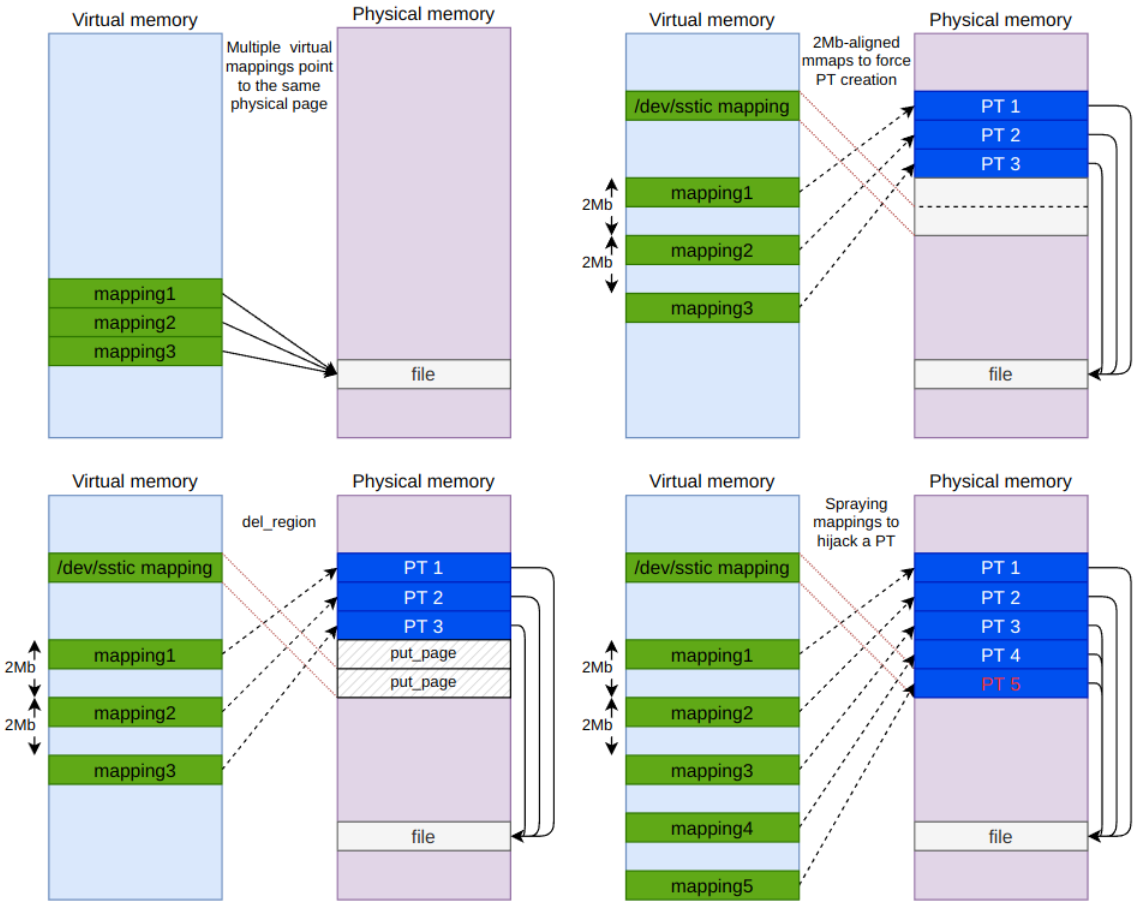

Figure 35: mmap spray pour détourner une table de page

Si on écrit maintenant dans notre mapping de `/dev/sstic` on peut modifier les PTE de la PT détournée. Pour retrouver le mapping dont on a détourné la table de page, on modifie une PTE pour pointer sur l'adresse physique 0 à laquelle se trouve le vecteur d'interruptions du BIOS.

On parcourt ensuite les mappings du spray pour voir lequel pointe sur le vecteur d'interruptions et plus sur le fichier.

Une fois le mapping victime trouvé, on peut unmapper tous les autres. On a notre agresseur (le mapping sur `/dev/sstic`) qui peut modifier les entrées de la table de page de la victime pour pointer sur une page physique arbitraire. On a donc accès à toute la mémoire physique du système !

Notes supplémentaires :

- Lorsqu'on modifie les PTE, il faut faire attention aux flags des pages. Il faut penser à set les flags `_PAGE_PRESENT` et `_PAGE_USER` pour pouvoir accéder à la page depuis notre mapping victime et aussi `_PAGE_RW` si on veut modifier son contenu.

- Pour optimiser les temps d'accès aux pages mémoires mappées, le processeur utilise une mémoire cache : le **TLB**[24]. Le TLB mémorise les dernières pages auquel le processeur a dû accéder pour éviter de parcourir les répertoires de PT, la PT pour enfin accéder à la page. Quand nous modifions l'entrée de la table de page de la victime et que nous essayons d'accéder aux données, il faut prendre en compte l'effet du TLB. Si le processeur utilise le TLB au lieu de la PT pour accéder à la page, notre modification d'entrée n'est pas prise en compte. Pour être certain de bien utiliser la PT qui prend en compte notre PTE modifiée, je remappe la victime à une nouvelle adresse puis je refait un mremap pour la replacer à l'adresse originale à chaque modification de la PTE.

6.4 Récupérer les clés

On rappelle que pour récupérer les clés, il faut désactiver le mode debug sur le périphérique PCI sur lesquelles elles sont stockées. Pour cela, on va scanner la mémoire physique pour trouver la page qui contient le code du module noyau qui intéragit avec le périphérique. Il suffit de lire les premiers octets de la page et de les comparer avec les premiers octets du module pour savoir si on a atteint la page cible.

Une fois qu'on a trouvé la page où se trouve le code du module, on va modifier le code de l'ioctl `submit_command` pour que sa seule action soit d'écrire 0 à `iomem + 0x28` ce qui désactive le mode debug.

Un ioctl `submit_command` désactive maintenant le mode debug.

On peut ensuite récupérer les clés du dossier **prod** et de son contenu.

Il y a deux fichiers à l'intérieur : `flags.txt` et `Canal_Historique.mp4`.

`flags.txt` contient le cinquième flag.

SSTIC{bf3d071f5a8a45fabc549d54be841f8b}

[24] Translation Lookaside Buffer

7 La fin

Enfin en possession de la fameuse vidéo qu'il fallait récupérer ! Plus qu'à retrouver l'adresse email de validation.

En regardant les flux du fichier mp4 avec ffmpeg, on découvre qu'il y a 1 flux audio et 2 flux vidéos. VLC permet d'ouvrir le deuxième flux avec `Vidéo > Piste vidéo`. L'email est incrusté dans l'image du deuxième flux.

44608171b27e7195d4cf@challenge.sstic.org

8 Conclusion

Ainsi se conclut ma première participation au challenge SSTIC. Un des CTF les plus qualitatifs et difficiles auquel j'ai participé. Un grand merci aux créateurs qui ont investi du temps dans la création des défis.

L'étape 2 était un peu à part par rapport au reste du challenge. Cependant, je n'ai pas trouvé ça dérangeant. Je n'avais jamais fait de pwn Windows et ça m'a donné l'occasion de m'y mettre. J'ai pû utiliser sérieusement WinDbg pour la première fois et comprendre un peu mieux certains mécanismes spécifiques à Windows. L'exploit a aussi été très fun à coder avec l'utilisation des différents formats de fichier.

L'environnement des étapes 3, 4, 5 m'a beaucoup plu aussi! Le fil rouge était bien trouvé : accéder aux différents dossier du DRM au fur et à mesure des étapes. On a l'objectif sous le nez dès le début de l'étape 3 et on voit notre progression tout le long du challenge. Le système de DRM dans son ensemble était vraiment amusant à étudier. Comprendre les interactions entre tous les composants, leurs spécificités... on a vraiment l'impression d'explorer un lieu complexe dont on doit se faire une carte mentale.

Tous les problèmes sont conçus d'une manière qui oblige à comprendre précisément ce qui se passe pour pouvoir arriver au bout. L'étape 5 m'a permis de vraiment pousser ma compréhension de certains mécanismes du noyau Linux, de lire du code et de la documentation.

Bref, j'en ressors avec beaucoup de connaissances supplémentaires, et la grande satisfaction d'être arrivé au bout après 3 semaines de travail !

Merci encore aux concepteurs du challenge et à ceux qui ont lu ce document jusqu'au bout !

Mathieu Dechambe