SYNACKTIV
DIGITAL SECURITY

Solution Challenge SSTIC 2021

Fabien Perigaud

Enoncé

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!

Le challenge consiste à récupérer une vidéo particulière protégée par le système de DRM du SSTIC et d'en extraire un e-mail (de la forme xxx@challenge.sstic.org). Des flags intermédiaires (optionnels) de la forme SSTIC{...} sont disponibles au fur et à mesure de la résolution. Vous pouvez les saisir si vous le souhaitez, en bas cette page, tout en étant connecté.

Introduction

L'objectif du challenge de cette année est tout d'abord d'analyser une capture USB correspondant à un transfert de données sur une clé. L'analyse de cette capture nous amène à trouver des instructions pour accéder à un service en écoute sur un Windows 10, qu'il faudra exploiter pour récupérer la solution de DRM attendue. Ensuite, une whitebox cryptographique sera analysée afin de pouvoir récupérer des clés de chiffrement de niveau 1 sur un serveur de clés, puis y exécuter du code pour récupérer des clés de niveau 2, et enfin élever ses privilèges au niveau du noyau pour atteindre les clés de niveau 3.

La présente solution explique pas à pas comment résoudre les différentes étapes. Dans la mesure du possible, les temps de résolution seront également indiqués afin de montrer la difficulté de chaque étape.

Le présent PDF est également une archive ZIP contenant l'ensemble des scripts mentionnés au cours de la solution.

Etape 1 : Capture USB

Le fichier de départ du challenge est une capture USB au format PCAP.

La capture, une fois ouverte avec Wireshark, contient de nombreuses requêtes de lecture :

```
USBMS       95 SCSI: Read(10) LUN: 0x00 (LBA: 0x00000001, Len: 7)
USB         64 URB_BULK out
USB         64 URB_BULK in
USBMS     3648 SCSI: Data In LUN: 0x00 (Read(10) Response Data)
```

mais également d'écriture :

```
USBMS       95 SCSI: Write(10) LUN: 0x00 (LBA: 0x00000040, Len: 1)
USB         64 URB_BULK out
USBMS      576 SCSI: Data Out LUN: 0x00 (Write(10) Request Data)
USB         64 URB_BULK out
USB         64 URB_BULK in
USBMS       77 SCSI: Response LUN: 0x00 (Write(10)) (Good)
```

Scapy est un logiciel de premier choix lorsqu'il s'agit de traiter des fichiers de type PCAP.
Nous pouvons rapidement écrire un script qui détecte les requêtes de lecture et d'écriture, puis récupère la réception ou l'envoi de données, avant d'écrire le contenu dans un fichier correspondant au numéro de LBA demandé.

```
$ python extract.py usb_capture_C0.pcapng
-> READ <-
0 -> 1 block(s)
data 200 bytes
-> READ <-
1 -> 7 block(s)
data e00 bytes
[...]
-> READ <-
108 -> f8 block(s)
data ffbf bytes
ERROR ffbf / 1f000
```

La récupération de paquets de grande taille pose problème, la faute étant à imputer à une MTU positionnée à 0xffff dans Scapy. Après l'ajout d'un « f » supplémentaire, tout se passe pour le mieux.

L'étape suivante consiste alors à reconstruire le contenu de la clé. Pour cela, les paquets en lecture sont d'abord réassemblés (en « bouchant » les trous avec des octets nuls), puis les écritures sont appliqués. Il en résulte un fichier image reconstruit avec une table de partition valide :

```
$ file dmp_rw.bin
```

```
dmp_rw.bin: DOS/MBR boot sector; partition 1 : ID=0xb, active, start-CHS (0x0,1,1), end-CHS
(0x119,31,63), startsector 63, 60435585 sectors
```

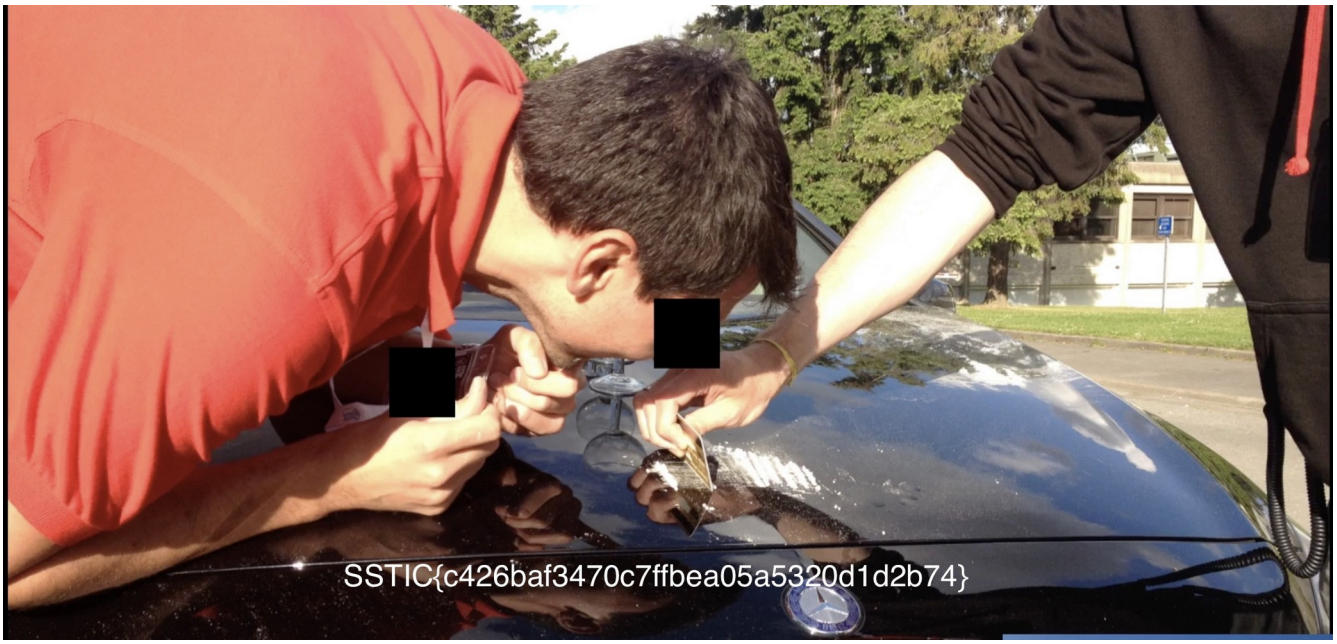
Il est alors possible de la monter ou de récupérer son contenu avec la suite sleuthkit :

```
$ /sbin/fdisk -l dmp_rw.bin
Disk dmp.bin: 20 MiB, 20971520 bytes, 40960 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x560405f2

Device     Boot Start      End  Sectors  Size Id Type
dmp.bin1   *       63 60435647 60435585 28.8G  b W95 FAT32

$ fls -o 63 dmp_rw.bin
r/r 3:SSTICKEY    (Volume Label Entry)
d/d 6:.Spotlight-V100
d/d 8:.fseventsd
r/r 11:      challenge.7z.001
r/r 14:      challenge.7z.007
r/r 17:      challenge.7z.005
r/r 20:      challenge.7z.003
r/r 23:      challenge.7z.006
r/r 26:      challenge.7z.004
r/r 29:      challenge.7z.002
r/r 32:      challenge.7z.008
v/v 181779:  $MBR
v/v 181780:  $FAT1
v/v 181781:  $FAT2
V/V 181782:  $OrphanFiles
```

Une fois réassemblé les fichiers challenge.7z.00X, nous obtenons une archive 7-zip contenant les fichiers suivants :

```
chall
chall/Readme.md
chall/env.txt
chall/flag.jpg
chall/A..Mazing.exe
```

L'image flag.jpg donne le premier flag intermédiaire :


SSTIC{c426baf3470c7ffbea05a5320d1d2b74}

Etape 2 : Exploitation userland Windows 10

Le fichier Readme.md nous donne les informations suivantes :

```
Hey Trou,

Do you remember the discussion we had last year at the secret SSTIC
party? We planned to create the next SSTIC challenge to prove that we
are still skilled enough to be trusted by the community.

I attached the alpha version of my amazing challenge based on maze
solving.

You can play with it in order to hunt some remaining bugs. It's
hosted on my workstation at home, you can reach it at
challenge2021.sstic.org:4577.

I've written in the env.txt file all the information about the remote
configuration if needed.

Have Fun,
```

Il semblerait que la suite consiste en un challenge d'exploitation de service distant. Le fichier env.txt donne des informations sur l'environnement dans lequel est exécuté le service :

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

L'exploitation va se faire sur un Windows 10 à jour, comprenant donc tous les mécanismes de défense en profondeur actuels, incluant CFG.

Reconnaissance

Le binaire implémentant le service est fourni et peut être analysé à l'aide d'IDA Pro.

Il s'agit d'un système de gestion de labyrinthes, et plusieurs actions sont proposées :
1. S'enregistrer (i.e. fournir un pseudonyme de maximum 127 caractères)
2. Créer un labyrinthe, manuellement ou de façon aléatoire

3. Charger une sauvegarde de labyrinthe
4. Jouer au labyrinthe courant
5. Supprimer un labyrinthe
6. Voir les scores du labyrinthe courant
7. « Upgrader » un labyrinthe

Le menu a la structure habituelle des challenges de CTF mettant en œuvre des vulnérabilités dans le tas (overflow, double free, use-after-free, …). Toutefois ces types de vulnérabilités n'ont pas été identifiés lors d'une revue attentive du code.

Il serait impossible de décrire complètement le binaire, mais quelques informations importantes pour la compréhension sont à noter :
- Il y a 3 types de labyrinthes possibles : classique (1), à chemins multiples (2) et avec des pièges (3)
- La sauvegarde d'un labyrinthe se fait dans un fichier `.maze`, et le tableau des scores associé est sauvegardé dans un fichier `.rank`
- Le mécanisme d'upgrade permet de changer le type d'un labyrinthe (1 vers 2 vers 3)
- Le calcul des scores est assez simple : un déplacement ajoute 1 point, et un piège ajoute un nombre arbitraire de points, choisi à la création du labyrinthe
- Le tableau des scores contient au maximum 128 entrées

Lors de l'analyse du binaire, deux bugs sont identifiés :

- A la création d'un labyrinthe avec pièges (ou lors d'une upgrade), un score pour les pièges est à renseigner : celui-ci est lu via un `scanf_s` en utilisant `%d`, et stocké dans un entier signé → il est donc possible de spécifier un score négatif pour les pièges

- Lors du chargement d'un labyrinthe précédemment sauvegardé, il est possible de l'ouvrir via son nom avec ou sans extension : le programme recherche alors le fichier `filename`, et s'il ne le trouve pas, le fichier `filename.maze` → il est donc possible de faire ouvrir un fichier arbitraire en tant que sauvegarde d'un labyrinthe

Pour exploiter cette dernière vulnérabilité, certaines contraintes sont à respecter concernant le nom de fichier : il est impossible de se déplacer dans les répertoires de part les caractères interdits.
On peut toutefois obtenir une confusion entre une sauvegarde de labyrinthe et une sauvegarde de scores en demandant à ouvrir `filename.rank` !

Afin de comprendre les implications de cette confusion, voyons les structures des deux fichiers :
- Fichier `.rank`

```
struct rank_file {
      uint8_t numbers_of_entries;
      struct highscore[numbers_of_entries];
}
```

```
struct highscore {
      uint8_t player_name_len;
      char player_name[player_name_len];
      uint64_t score;
}
```

- Fichier `.maze`

```
struct maze_file_type_1_2 {
      uint8_t author_name_len;
      char author_name[author_name_len];
      uint8_t type;
      uint8_t columns;
      uint8_t lines;
      char maze[columns*lines];
}

struct maze_file_type_3 {
      uint8_t author_name_len;
      char author_name[author_name_len];
      uint8_t type;
      uint8_t columns;
      uint8_t lines;
      char maze[columns*lines];
      uint8_t number_of_traps;
      struct trap[number_of_traps];
}

struct trap {
      uint64_t score;
      uint16_t position;
      uint8_t display_character;
}
```

Un constat intéressant est que le format du fichier est différent selon le type de labyrinthe : le type 3 contient en plus les informations concernant les pièges.

Lors du chargement en mémoire d'un labyrinthe, la structure créée est la suivante :

```
struct memory_maze_type_1_2 {
      uint8_t columns;
      uint8_t lines;
      uint8_t type;
      char author_name[128];
      char *maze;
      [...]
}

struct memory_maze_type_3 {
      uint8_t columns;
      uint8_t lines;
      uint8_t type;
      char author_name[128];
      uint8_t number_of_traps;
      struct trap_info[256];
      char *maze;
      [...]
}
```

```
struct trap_info {
      uint64_t score;
      uint16_t position;
      uint8_t enabled;
      uint32_t pad?;
}
```

Les étapes classiques d'exploitation sont généralement d'obtenir un leak d'une adresse mémoire, puis un read arbitraire, un write arbitraire, et enfin l'exécution de code arbitraire.

Leak

En ouvrant un fichier de scores contenant 128 entrées en tant que labyrinthe, les champs `number_of_entries` et `author_name_len` seront confondus : il en resultera la lecture de 128 octets du ficher dans le champ `author_name` de la structure en mémoire. Dès lors, si aucun de ces octets n'est nul, l'affichage du nom de l'auteur continuera jusqu'à lire le pointeur `maze` dans le cas d'un labyrinthe de type 1 ou 2.
Afin de réaliser cette confusion, il convient de créer un labyrinthe personnalisé, puis de réaliser 128 scores avec un pseudonyme de 127 caractères. Le score associé devra lui aussi être spécialement choisi, puisque la confusion le fera correspondre aux champs `type`, `columns` et `lines` : un score de 0x050301 par exemple correspondra à un labyrinthe de type 1 de 3 colonnes sur 5 lignes. Pour cela, le labyrinthe que nous créons sera le plus simple possible, et nous affecteront au score du piège la valeur souhaitée comme score final, moins 2 (le nombre de déplacements normaux nécessaires pour sortir) :

```
###
#x#
#^#
# o
###
```

La position initiale est le caractère « x », le piège « ^ » et la sortie « o ».

Si nous chargeons le fichier `.rank` en tant que labyrinthe, la première ligne du tableau des scores est la suivante :

```
Scoreboard                for                leak                (created                by
⌂AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBÉ┘ĐCè☻)
```

Les caractères non imprimables en fin de pseudonyme représentent un pointeur dans le heap, correspondant à une allocation de taille (lines * columns + 1).

Arbitrary read

Une façon d'obtenir un read arbitraire serait de provoquer une confusion entre les types des labyrinthes, afin que les informations sur les pièges soient confondues avec le pointeur `maze`.

Pour cela, nous pouvons exploiter un autre ensemble de bugs dans le programme : si le type du labyrinthe est supérieur à 3, les comportements varient selon les actions effectuées sur le labyrinthe : au chargement, la structure créée est considéré comme étant de type différent de 3, mais lors de la lecture du fichier, si le type est ≥ 3, alors les données des pièges sont lues par dessus le pointeur `maze`.
Il est alors possible de créer un fichier de scores correspondant à un labyrinthe de type 4, et de spécifier un pointeur arbitraire dans les données des pièges, pour obtenir une lecture arbitraire lorsque l'on choisit de jouer le labyrinthe : les données pointées sont alors directement affichées (séparées par des « \r\n » à chaque ligne) pour dessiner le labyrinthe.

Le read arbitraire nous permet alors de récupérer par mal d'informations :
- Depuis le leak, nous remontons à la structure `HEAP` correspondante ;
- Du heap, nous récupérons un pointeur dans `ntdll.dll` ;
- De ntdll.dll, nous récupérons un pointeur vers le `PEB` ;
- Du `PEB`, nous récupérons le `TEB` puis la valeur initiale du pointeur de pile ;
- De la pile, nous récupérons l'adresse de retour de la fonction `main` ainsi que sa position dans la pile ;
- Nous en déduisons l'adresse de base du binaire, et pouvons récupérer des pointeurs dans `kernel32.dll`.

Il ne reste plus qu'à trouver une écriture arbitraire pour pouvoir exécuter du code arbitraire en allant écrire une ROP-chain directement dans la pile.

Arbitrary write

En utilisant le mécanisme d'« Upgrade », il est possible de transformer notre labyrinthe de type 4 (inconnu) en type 2, puis en type 3. La transformation en type 3 provoque un changement de la structure en mémoire, et le pointeur `maze` maitrisé est recopié dans la nouvelle structure au nouvel offset.

Par la suite, en choisissant à nouveau une « Upgrade », il nous est proposé de modifier l'emplacement des pièges : pour cela l'utilisateur renvoie le labyrinthe complet au format précédemment affiché, sans les sauts de lignes. Ces données envoyées sont directement écrites à l'emplacement pointé par le pointeur `maze`, nous fournissant le write arbitraire souhaité.

Exécution

L'exécution consiste maintenant simplement à écrire une ROP-chain minimaliste : nous choisissons un simple `pop rcx ; ret` suivi de l'adresse de `WinExec` afin d'obtenir un shell interactif.

Une fois le shell obtenu, un minimum de reconnaissance nous montre la présence d'un fichier `DRM.zip` sur le bureau de l'utilisateur. Nous relançons l'exploit avec simplement la commande suivante :

```
type c:\users\challenge\desktop\drm.zip
```

Le contenu du fichier peut alors être récupéré en lisant directement la socket (environ 14Mo).

Une fois récupéré, nous pouvons l'extraire :

```
DRM/
DRM/DRM_server.tar.gz
DRM/Readme
DRM/libchall_plugin.so
```

Le fichier Readme a le contenu suivant, écrit par le président du CO SSTIC en personne :

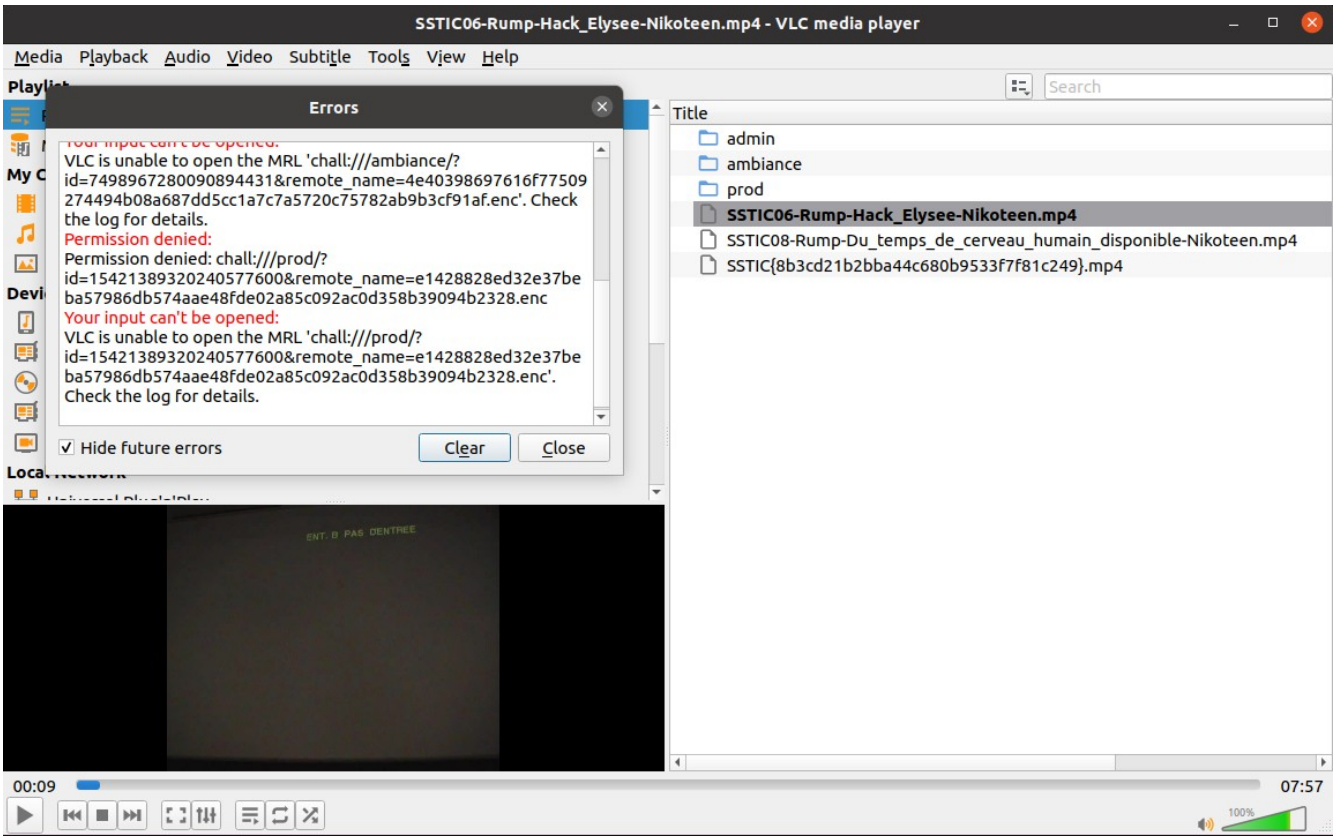
```
Here is a prototype of the DRM solution we plan to use for SSTIC 2021.
It's 100% secure, because keys are stored on a device specifically designed
for this. It uses a custom architecture which garantee even more security!
In any case, the device is configured in debug mode so production keys can't
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
```

Pas de flag ! Ca viendra surement plus tard :)

Etape 3 : Whitebox cryptographique en VM

La bibliothèque récupérée précédemment est un plugin pour le logiciel VLC. Celui-ci permet de gérer l'ouverture d'un flux media dont l'URL commence par `chall://`.

Si nous chargeons le plugin dans VLC et ouvrons `chall:///`, nous obtenons la playlist suivante :



Nous obtenons alors le fameux flag manquant à l'étape précédente.

Plusieurs messages d'erreur indiquent que nous n'avons les droits que pour ouvrir le répertoire « rumps », mais que l'accès nous est interdit pour « ambiance », « admin » et « prod ».

Il est temps de voir comment fonctionne tout ça.

Architecture

Le système de DRM est composé de 3 parties :

- Le client VLC et sa bibliothèque libchall_plugin.so

- Le serveur de contenu : challenge2021.sstic.org sur le port 8080 en HTTP ;

- Le serveur de clés : 62.210.125.243 sur le port 1337 (protocole custom).

Serveur de contenu

Lors de l'ouverture du flux, une requête est effectuée pour récupérer l'index au format JSON à l'adresse « http://challenge2021.sstic.org:8080/files/index.json » :


```
[
  {
    "name": "930e553d6a3920d05c99bc3111aaf288a94e7961b03e1914ca5bcda32ba9408c.enc",
    "real_name": "admin",
    "type": "dir_index",
    "perms": "0000000000000000",
    "ident": "75edff360609c9f7"
  },
  {
    "name": "4e40398697616f77509274494b08a687dd5cc1a7c7a5720c75782ab9b3cf91af.enc",
    "real_name": "ambiance",
    "type": "dir_index",
    "perms": "00000000cc90ebfe",
    "ident": "6811af029018505f"
  },
  {
    "name": "e1428828ed32e37beba57986db574aae48fde02a85c092ac0d358b39094b2328.enc",
    "real_name": "prod",
    "type": "dir_index",
    "perms": "0000000000001000",
    "ident": "d603c7e177f13c40"
  },
  {
    "name": "40f865fb77c3fd6a3eb9567b4ad52016095d152dc686e35c3321a06f105bcaba.enc",
    "real_name": "rumps",
    "type": "dir_index",
    "perms": "ffffffffffffffff",
    "ident": "68963b6c026c3642"
  }
]
```


Le champ « name » correspond au nom du fichier sur le serveur de contenu, « real_name » au nom affiché dans le client, « type » au type (répertoire ou fichier), « perms » aux permissions requises pour l'accès (plus la valeur est petite, plus les permissions sont élevées) et enfin « ident » est utilisé comme identifiant du fichier auprès du serveur de clés.

Puisque nous n'avons accès qu'au répertoire « rumps », nos permissions actuelles doivent être « ffffffffffffffff ».

Serveur de clés

La communication avec le serveur de clés suit un protocole relativement simple. Un premier octet est envoyé, correspondant à un numéro de commande. S'en suit l'envoi de 20 octets dont la signification nécessite de se plonger dans libchall_plugin.so.

Avant de communiquer avec le serveur de clés, une bibliothèque est téléchargée (« http://challenge2021.sstic.org:8080/api/guest.so ») vers un fichier temporaire, et celui-ci est supprimé juste après son chargement en mémoire.

Cette bibliothèque exporte 3 fonctions :

- useVM

- getPerms

- getIdent

Chacune de ces fonctions exportées appelle la même fonction qui est un interpréteur de machine virtuelle, tout en ayant positionné à 0, 1 ou 2 le premier octet d'un buffer d'entrée.

Sans surprise, la fonction `getPerms` retourne 8 octets à 0xff, correspondant à la permission « ffffffffffffffff ». La fonction `getIdent` renvoie elle un identifiant sur 4 octets qui correspond au timestamp unix du moment du téléchargement de la bibliothèque « guest.so ». On en déduit alors que cette bibliothèque doit être regénérée à chaque téléchargement.

Enfin, la fonction useVM semble effectuer le chiffrement d'un buffer de 16 octets. Sa première utilisation par le plugin effectue le chiffrement d'un buffer dont les 8 premiers octets sont à 0, et les 8 suivants correspondent à la permission courante (soit 8 octets à 0xff). Le retour de la fonction est ensuite envoyé en argument de la commande 0 au serveur de clés, suivi des 4 octets d'identifiant.

En retour, le serveur de clés renvoie un code d'erreur (1 pour « OK » et 2 si l'identifiant a expiré, ce qui arrive après 3600 secondes) et le contenu du buffer déchiffré.

Le serveur de clés implémente 4 fonctions, et seules les deux premières sont accessibles avec les permissions « ffffffffffffffff » :

- Commande 0 : déchiffre le buffer de 16 octets, et renvoie le contenu déchiffré ainsi qu'un indicateur si l'identifiant a expiré ;

- Commande 1 : déchiffre le buffer de 16 octets, et si l'identifiant n'est pas expiré, récupère la clé de chiffrement pour le fichier dont l'identifiant est spécifié dans les 8 premiers octets, si les permissions le permettent.

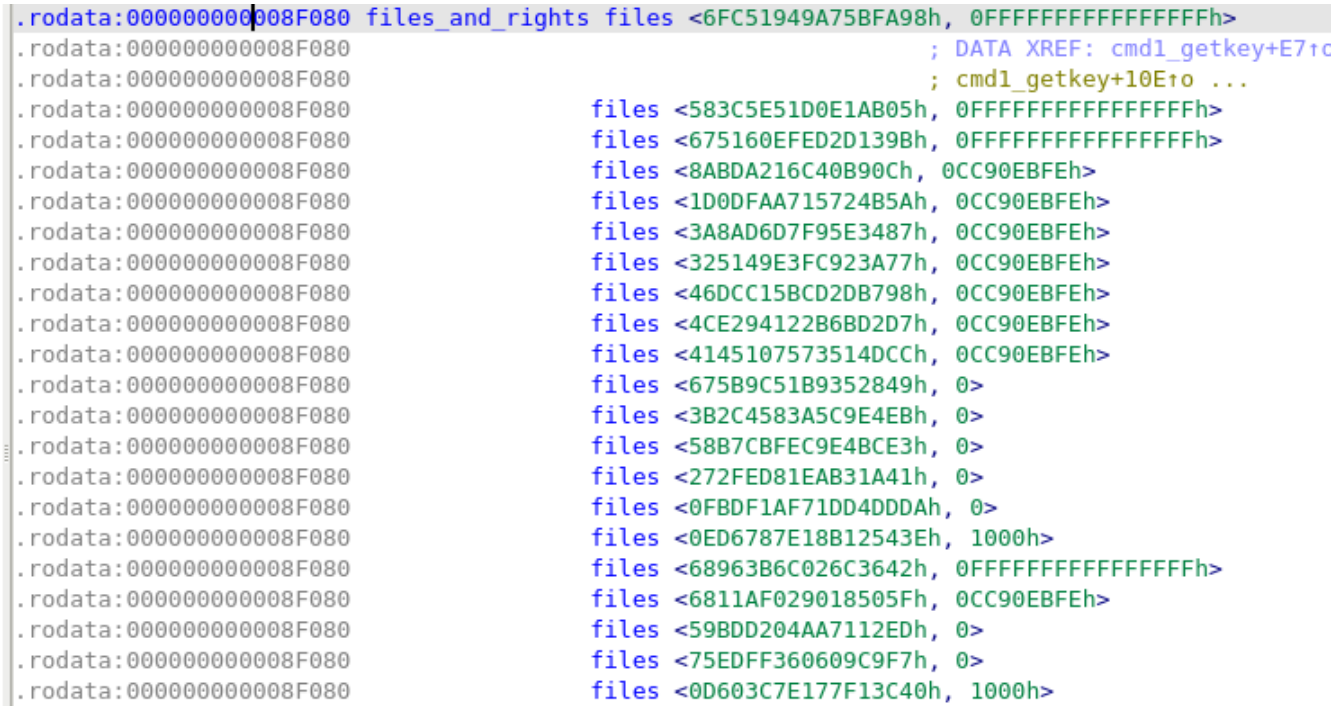


Figure 1: Identifiants de fichiers et permissions associées

Des vérifications supplémentaires sont effectuées :

- si le bit de poids fort de l'identifiant est positionné, il s'agit d'un fichier de prod, et le service refusera de récupérer la clé si le device est en debug ;

- si la permission requise pour lire le fichier est « 0 », l'accès est refusé.

Les mécanismes de déchiffrement et de demande de clés passent par l'envoi d'IOCTL à un driver `sstic.ko`. Celui-ci offre des primitives d'allocation de pages mémoire physiques pour faire le lien entre le userland et un périphérique PCI connecté au serveur, et permet l'exécution de différentes commandes sur le périphérique : ici, le déchiffrement de données et la récupération des clés.

L'objectif est alors plutôt clair : analyser la machine virtuelle de la bibliothèque « guest.so » afin de pouvoir chiffrer une demande de clé avec une permission arbitraire.

Bibliothèque de chiffrement

L'analyse de la VM ne pose pas de difficulté majeure, l'encodage des instructions est assez simple et leur nombre est réduit. Un émulateur est rapidement écrit afin de pouvoir simplement tester différentes entrées et observer la sortie.

Le code gérant `getPerms` et `getIdent` est très simple, la valeur de retour étant « en dur » dans le code.

Concernant `useVM`, le code vérifie tout d'abord que les 8 derniers octets de l'input sont à « 0xff », puis une whitebox cryptographique est exécutée. L'algorithme implémenté dans la whitebox n'a pas été identifié avec certitude, mais l'aspect du code montre 3 blocs de 6 tours séparés par une transformation. Quelques recherches pointent vers l'algorithme Camellia avec une clé de 128-bits, dont la structure correspondrait parfaitement.

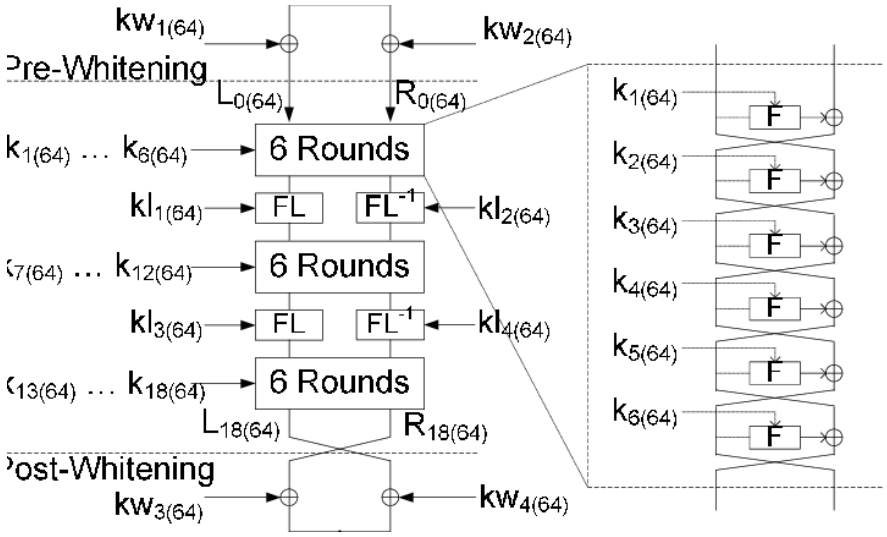


Figure 2: Camellia 128

L'une des caractéristiques de cette whitebox est que seuls les 8 premiers octets de l'input sont pris en compte. Puisque les 8 derniers sont testés en prologue, on peut penser que leur valeur a également été intégrée à l'algorithme de la whitebox.

Plusieurs solutions s'offrent alors à nous :

1. Extraire la clé de la whitebox, en espérant avoir correctement trouvé l'algorithme implémenté → Cette solution est complexe et nécessiterait pas mal de recherches

2. Injecter des fautes afin d'obtenir un chiffré correspondant à un niveau de permission souhaité

Si l'on observe le schéma décrivant l'algorithme, on constate qu'il est possible d'inférer sur les 8 derniers octets du clair en modifiant `R0`. Disposant d'un oracle de déchiffrement (le serveur de clés), nous allons modifier un octet de `R0` en nous plaçant en sortie du premier tour, et envoyer le chiffré obtenu au serveur de clés afin d'observer si le même octet dans le clair a pris la valeur que nous attendions. En répétant cette opération avec 256 valeurs différentes, nous finissons par obtenir la valeur souhaitée → il suffit alors d'effectuer cette opération pour les 8 octets de permission jusqu'à obtenir une suite de octets à zéro.

Dès lors, nous sommes en mesure de demander les clés pour tous les fichiers dont le niveau de permission est 0xCC90EBFE en générant pour chaque identifiant le chiffré incluant le bon niveau de permissions. Les autres fichiers (permission à 0 ou fichier de « prod ») resteront pour l'instant inaccessibles…

Afin de contacter le serveur, il est nécessaire de récupérer une nouvelle instance de la bibliothèque « guest.so », notre identité précédente ayant expiré (délai d'une heure). Surprise : la machine virtuelle change pour chaque nouvelle bibliothèque téléchargée : nouveaux opcodes et nouveau fil d'exécution. Toutefois, on constate rapidement que l'ordre d'apparition d'un nouvel opcode ne change pas d'une instance à une autre, il est alors possible, pour chaque instruction inconnue, d'aller piocher l'entrée suivante dans la liste des instructions supportées.

Un dernier point reste à élucider : il serait plus simple de gérer le déchiffrement des fichiers en direct plutôt que de passer par VLC. Celui-ci se réimplémente aisément en Python : il s'agit d'un simple AES-128-CTR.

L'extraction des clés nous permet alors de lire le répertoire distant « ambiance » :


```
[
  {
    "name": "5534d32f4fd6a1454d55924291fc1d179ff84521920272ae4e8ae718e0c39392.enc",
    "real_name": "Suite Sud Armoricaine.mp3",
    "type": "mp3",
    "perms": "00000000cc90ebfe",
    "ident": "1d0dfaa715724b5a"
  },
  {
    "name": "581ed636bd7a1bbab890aeb1b458bb4f3bff59827afdd8582486ff0a22944aec.enc",
    "real_name": "Swallowtail Jig - Irish Fiddle Tune.mp3",
    "type": "mp3",
    "perms": "00000000cc90ebfe",
    "ident": "3a8ad6d7f95e3487"
  },
  {
    "name": "1026f340ad5175f2a73d2e3513d69ffd96285ca9ec89f50629a3426e6be45b09.enc",
    "real_name": "The Banks of Spey -- Scottish Fiddle Tune.mp3",
    "type": "mp3",
    "perms": "00000000cc90ebfe",
    "ident": "325149e3fc923a77"
  },
  {
    "name": "f0808dfbf75a5afaddff38574fe2bf03f2ff43b78cfca74aace782e06bc69511.enc",
    "real_name": "The Era of Legends.mp3",
    "type": "mp3",
    "perms": "00000000cc90ebfe",
    "ident": "46dcc15bcd2db798"
  },
  {
    "name": "96fe4e62d09539ad93093c441766dfc0011dc824ab4b9b90f6b366cd9578ccbf.enc",
    "real_name": "The Lone Wolf.mp3",
    "type": "mp3",
    "perms": "00000000cc90ebfe",
    "ident": "4ce294122b6bd2d7"
  },
  {
    "name": "11b1aef316795c3a3a440596216dd288fbee939689fad49e82d78baf52b574da.enc",
    "real_name": "Tri Martelod.mp3",
    "type": "mp3",
    "perms": "00000000cc90ebfe",
    "ident": "4145107573514dcc"
  },
```

```
  {
    "name": "48e3847a2774bf900c2cda70503dab44e37b5cfe14e0367b555e246bf2e75943.enc",
    "real_name": "info.txt",
    "type": "txt",
    "perms": "00000000cc90ebfe",
    "ident": "08abda216c40b90c"
  }
]
```

Le fichier « info.txt » semble intéressant :

```
$ python decrypt.py 48e3847a2774bf900c2cda70503dab44e37b5cfe14e0367b555e246bf2e75943.enc
acf00abcec52b4332b316eeb13ffcce5
Musique pour les entractes

SSTIC{9a5914929b7947afbef39446aafacd35}
```

Etape 4 : Exécution de code sur un processeur inconnu

Nous sommes maintenant en mesure d'envoyer au serveur de clés un message avec des permissions arbitraires, ce qui nous donne accès aux gestionnaires de commande 2 et 3 :

- Commande 2 : permet de charger le périphérique PCI avec du code et des données fournies par l'utilisateur, et de déclencher l'exécution. L'état des registres et de la pile sont affichés en sortie.

- Commande 3 : charge un blob de code dans le périphérique PCI ainsi que des données fournies par l'utilisateur, et vérifie que la sortie correspond à 64 octets particuliers : si tel est le cas, un binaire peut être envoyé et directement exécuté sur le système Linux, et sa sortie est renvoyée.

Ces deux commandes s'appuient à nouveau sur le driver `sstic.ko` qui fournit les adresses physiques des buffers de code et de données au périphérique PCI.

L'objectif de cette étape va être de comprendre le jeu d'instruction de l'architecture CPU du périphérique PCI, avant d'envoyer les données attendues et être en mesure d'exécuter un binaire arbitraire sur le serveur.

Pour cela, nous récupérons le code prévu pour être chargé par la commande 3, et n'envoyons qu'un nombre restreint de octets pour observer le contenu des registres. Par exemple, en n'envoyant que les 4 premiers octets de code et des données au contenu prévisible on obtient :

```
---DEBUG LOG START---
Bad instruction
regs:
PC : 4
R0 : 42424242424242424242424242424242
R1 : 00000000000000000000000000000000
R2 : 00000000000000000000000000000000
R3 : 00000000000000000000000000000000
R4 : 00000000000000000000000000000000
R5 : 00000000000000000000000000000000
R6 : 00000000000000000000000000000000
R7 : 00000000000000000000000000000000
RC : 00000000000000000000000000000000
stack: []
---DEBUG LOG END---
```

La première constatation est que PC a toujours une valeur alignée sur 4 octets : les instructions sont de taille fixe.

Ensuite, la première instruction est composée des octets suivants : `4E 01 40 20`. En jouant avec cette instruction et le contenu du buffer de données, nous pouvons identifier plusieurs choses :

- l'instruction est un « load » de données vers un registre ;

- les deux derniers octets sont l'adresse à laquelle sont lues les données en little endian (0x2040) ;

- le numéro de registre est encodé dans les bits 2 à 4 du deuxième octet ;

- le code est chargé à l'adresse 0x1000 et les données à l'adresse 0x2000.

Les instructions suivantes ne montrent pas grand-chose, mais celle en 0x1C est particulièrement intéressante :

```
00 1B 01 00
```

D'après les données dont nous disposons, celle-ci devrait se désassembler en :

```
??? R6, 0x1
```

Toutefois, après exécution, le registre R6 contient la valeur suivante :

```
R6 : 01010101010101010101010101010101
```

Le processeur semble supporter des instructions vectorielles. Si l'on joue cette instruction deux fois de suite, le registre ne contient que des octets valant 0x2, il s'agit donc d'une instruction d'addition. En jouant sur le premier nibble du premier octet, on constate que selon la valeur, le registre peut prendre les valeurs suivantes :

```
R6 : 01000100010001000100010001000100
R6 : 01000000010000000100000001000000
R6 : 01000000000000000100000000000000
```

Ce nibble influe donc directement sur la taille des vecteurs !

L'association valeur / taille de vecteur semble être la suivante :

- 0 → 8 bits

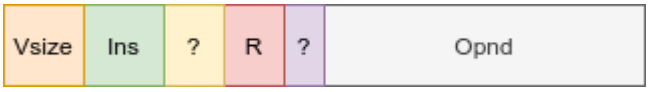
- 1 → 16 bits

- 2 → 32 bits

- 3 → 64 bits

- 4 → 128 bits

Le second nibble du premier octet semble quand à lui correspondre à l'instruction. Il ne reste maintenant que peu d'inconnues dans la composition d'une instruction :



- Vsize : taille du vecteur (4 bits)

- Ins : instruction (4 bits) : 0 pour `ADD`, 0xe pour `LDR`

- R : registre destination (3 bits)

- Opnd : opérande (16 bits)

Pour les 2 bits en violet, ceux-ci semblent influer sur l'interprétation de l'opérande : selon les deux instructions observées précédemment, si la valeur est « 1 », l'opérande est une adresse déréférencée, si la valeur est « 3 », l'opérande est une valeur immédiate.

A ce niveau d'analyse, il est possible de comprendre les autres instructions implémentées. Au total, il y a 14 instructions utilisées (sur les 16 possibles). Le registre « RC » indiqué dans la sortie de debug contient le résultat des opérations de comparaison, et influe directement sur les opérations de branchement. Les conditions de ces opérations de branchement et comparaison sont stockées dans les 3 bits en jaune qu'il restait à décoder.

Une fois le schéma d'encodage des instructions compris, il est possible de désassembler le bytecode de la VM et de l'analyser pour comprendre le format des données d'entrée attendues.

La valeur en sortie doit être la suivante :

```
00000000: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000010: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000020: ffff ffff ffff ffff ffff ffff ffff ffff  ................
00000030: 4558 4543 5554 4520 4649 4c45 204f 4b21  EXECUTE FILE OK!
```

Les données d'entrée ont une taille attendue de 0x50, et les 16 derniers octets sont vérifiés puis utilisés dans la première partie du bytecode pour déchiffrer une sous-fonction (à l'aide d'un simple XOR). Plusieurs conditions sont testées sur ces 16 octets, ce qui permet non seulement d'aider à comprendre le bytecode de la VM, mais également de connaître précisément la valeur attendue : il s'agit d'une distribution de valeurs de 0 à 0xf devant satisfaire de nombreuses contraintes selon différentes tailles de vecteurs.

Au final, la valeur attendue est :

```
0e03050a0804090b000c0d070f020601
```

Une fois la sous-fonction déchiffrée, on fait face à un algorithme de chiffrement (ou plutôt déchiffrement) en 20 rounds avec un schéma de Feistel. Les rounds pairs et impairs sont légérement différents et induisent des rotations sur 3 des 4 vecteurs en jeu (opération de shift rows). On pense alors à ChaCha au vu de la fonction de quart de tour.

L'inversion de l'algorithme, une fois réimplémenté en Python, ne pose pas de difficulté majeure, et on identifie alors l'entrée attendue :

```
00000000: 6578 7061 6e64 2033 322d 6279 7465 206b  expand 32-byte k
00000010: 62cc 273d e890 5581 c4fa c91c be45 1034  b.'=..U......E.4
00000020: 1a09 16ca fa05 14f6 80e4 604a a897 bad4  ..........`J....
00000030: ad62 a02d cd9b 3574 87f6 7ab4 7134 b697  .b.-..5t..z.q4..
00000040: 0e03 050a 0804 090b 000c 0d07 0f02 0601  ..............…
```

Une fois cette entrée envoyée à la commande 3, il est possible d'envoyer un binaire de maximum 900ko qui sera exécuté sur le serveur. Nous pouvons alors implémenter la récupération des clés pour les fichiers dont les permissions sont à 0. Il ne nous est toujours pas possible de récupérer les clés de prod, une vérification étant effectuée directement dans le driver `sstic.ko`.

Une fois les clés « admin » en notre possession, nous pouvons récupérer le contenu du répertoire :


```
[
  {
    "name": "bfed24eb16bacb67a1dd90468223f35d5d5f751ca1f1323b7943918ca2b3ae18.enc",
    "real_name": "CO_favorite_clip",
    "type": "dir_index",
    "perms": "0000000000000000",
    "ident": "59bdd204aa7112ed"
  },
  {
    "name": "6e875d839cac95d7ce50da2270064752ebf7e248e3e71498bb7ce77986d3b359.enc",
    "real_name": "SSTIC{377497547367490298c33a98d84b037d}.mp4",
    "type": "mp4",
    "perms": "0000000000000000",
    "ident": "675b9c51b9352849"
  }
]
```


Nous récupérons alors le flag de l'étape 4, et pouvons passer à l'étape finale. La vidéo est d'ailleurs l'occasion de renouer avec la mascotte du challenge SSTIC depuis 10 ans !

Etape 5 : Exploitation d'un driver Linux

Maintenant que nous sommes en mesure d'exécuter du code arbitraire sur le serveur, nous pouvons librement interagir avec le driver `sstic.ko`. Etant donné que l'IOCTL de récupération des clés empêche directement la récupération des clés de production si le device est en mode debug, cette étape doit consister en l'exploitation d'une vulnérabilité dans le driver afin d'obtenir les privilèges du noyau, et demander directement les clés au device PCI, ou désactiver le mode debug.

Comme indiqué précédemment, le driver expose des primitives d'allocation de mémoire physique par le biais d'une IOCTL, et dispose de son propre gestionnaire de `mmap()`, ainsi que d'un ensemble de fonctions de gestion d'opérations de mapping (« vm_operations ») : `open`, `close`, `fault` et `split`.

Tout pointe vers une vulnérabilité dans ce mécanisme, le driver pouvant être conçu pour se passer complètement de cette partie.

Pour allouer des pages, il faut passer par l'IOCTL `ioctl_alloc_region`. Celle-ci prend deux paramètres : un nombre de pages (puissance de 2) et des droits. Une allocation physique est alors réalisée via `alloc_pages_current`, puis découpée page par page via `split_page`. L'adresse de chaque page est insérée dans une structure `phy_region`, elle même insérée dans une structure `sstic_region`, ajoutée dans une liste doublement chaînée rattachée à la session (`sstic_session`) courante. Une session est créée à l'ouverture du device `/dev/sstic` et détruite à sa fermeture.

```
struct phy_region {
      uintptr_t vm_start;
      uintptr_t vm_end;
      int pages_count;
      int refcount;
      struct page *pages[pages_count];
};

struct sstic_region {
      struct phy_region *phy;
      int rights;
      int id;
      uintptr_t next;
      uintptr_t prev;
};

struct sstic_session {
      struct sstic_region *region0;
      struct sstic_region *region1;
      struct sstic_region *region2;
      struct sstic_region *region3;
      uintptr_t next_region;
      uintptr_t prev_region;
};
```

L'id de la structure `sstic_region` est incrémental.

Il est également possible via l'IOCTL `ioctl_assoc_region` d'associer une région à l'un des slots de région de la structure `sstic_session`, celles-ci servant directement lors des échanges avec le device PCI.

Une fois une région créée, il est possible de la mapper en userland via un appel à `mmap()`. Celui-ci déclenche l'appel du gestionnaire du driver, `sstic_mmap`. Cette fonction va rechercher une région dont l'id correspond au champ pg_off fournit, vérifier si la taille est la même que celle demandée, ainsi que d'autres vérifications sur les droits demandés. Si toutes les vérifications passent, une nouvelle structure `phy_region` est allouée, les pointeurs de pages y sont recopiés tout en incrémentant leur `refcount`. Cette structure est ensuite affectée au champ `vm_private_data` de la structure `vma`, et les opérations `sstic_vm_ops` sont affectées au champ `vm_ops`.

Ces opérations sont au nombre de 4 comme indiqué précédemment :

- `sstic_vm_fault` : ce gestionnaire se charge de mapper de manière effective la page physique lorsque l'adresse virtuelle correspondante est accédée en userland ;

- `sstic_vm_close` : decref la structure `phy_region` et la libère s'il n'y a plus de références dessus : la libération decref également chaque structure de page et libère la page physique s'il n'y a plus de référence ;

- `sstic_vm_split` : affecte les champs `vm_start` et `vm_end` de la structure `phy_region` avec respectivement l'adresse de start et l'adresse de split ;

- `sstic_vm_open` : si le champ `vm_end` de la structure `phy_region` est nul, incref la structure, sinon, effectue le découpage de la région en deux selon les valeurs positionnées dans `vm_start` et `vm_end` : une nouvelle `phy_region` est allouée, et les pages sont séparées entre l'existante et la nouvelle, puis la nouvelle `phy_region` est affectée à `vm_private_data`.

Pour comprendre ce dernier comportement, il convient de lire les sources de la fonction `__split_vma` dans le noyau Linux, chargée de découper une zone mappée : un appel au gestionnaire `split` est effectué avant un appel au gestionnaire `open`.

Un bug est présent dans cette implémentation lorsque `__split_vma` est appelé avec `new_below` à 1 : après un appel à `sstic_vm_open` après un `sstic_vm_split`, les champs `vm_start` et `vm_end` ne sont jamais repositionnés à 0. Si un nouvel appel à `sstic_vm_open` survient sans avoir un `sstic_vm_split`, une nouvelle opération de recopie des pointeurs de page va être effectuée. Il est possible d'exploiter ce comportement pour avoir deux fois un pointeur vers la même structure page dans une `phy_region`, ce qui va provoquer un double decref lors de sa libération. Il devient alors possible d'avoir un « UAF » de page physique, c'est à dire qu'un mapping userland continuera d'exister alors que la page physique a été libérée.

Pour appeler `sstic_vm_open`, il est possible de faire un fork() du processus.

La stratégie d'exploitation est alors la suivante :

- Allocation d'une région de 4 pages en R/W ;

- `mmap()` de cette région en userland ;

- `munmap()` de la première page mappée → cela déclenche un `__split_vma` avec `new_below` à 1 et la recopie des pointeurs de page des slots 1 à 3 de la `phy_region` vers les slots 0 à 2 ;

- `fork()` → nouvel appel à sstic_vm_open, et à nouveau recopie des pointeurs de page des slots 1 à 3 de la `phy_region` vers les slots 0 à 2.

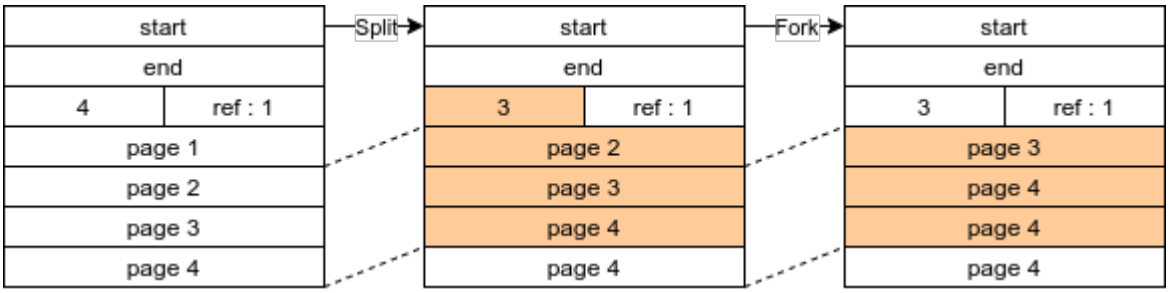


Figure 3: Double référence sans incref

Le processus fils peut alors appeler `munmap()` sur le reste de la région mappée, puis supprimer la région via l'IOCTL `ioctl_del_region` → le compteur de références de la page n°4 va alors atteindre 0, et la page sera libérée, alors que celle-ci est toujours mappée dans le processus parent.

Cette primitive est extrêmement puissante, il devient possible de mapper une page directory dans la page physique que nous venons de libérer, et ainsi manipuler directement des `PTE` depuis le mapping userland, fournissant un accès complet à la mémoire physique.

Nous utilisons alors cet accès pour retrouver le driver en mémoire, et patcher le code de l'IOCTL `ioctl_get_keys` afin de supprimer la vérification sur l'ID de fichier et provoquer une désactivation du mode debug. Une fois ce patch appliqué, nous pouvons récupérer les clés des fichiers de prod.

```
---EXEC OUTPUT START---
RAM 5f0b000
fd 5
Region 53248
addr 7FE561E09000
unmapped
forked 85
ready
open ok
Dumping ready
67 A0 A4 05 00 00 00 80  67 B0 A4 05 00 00 00 80  |  g.......g.......
67 C0 A4 05 00 00 00 80  67 D0 A4 05 00 00 00 80  |  g.......g.......
67 E0 A4 05 00 00 00 80  67 F0 A4 05 00 00 00 80  |  g.......g.......
67 00 A5 05 00 00 00 80  67 10 A5 05 00 00 00 80  |  g.......g.......
[+] Got a candidate! 0x10000000
Found 10 e2
48 8B 87 A8 00 00 00 48  89 70 08 48 8B 17 48 89  |  H......H.p.H..H.
10 31 C0 C3 66 66 2E 0F  1F 84 00 00 00 00 00 90  |  .1..ff..........
C3 66 66 2E 0F 1F 84 00  00 00 00 00 0F 1F 40 00  |  .ff...........@.
```

```
4C 8B 07 48 8B 77 18 49  8B 08 49 8B 80 A8 00 00  |  L..H.w.I..I.....
io_get_dbg -> 1
io_get_key 22
File fbdf1af71dd4ddda 0000000000000000 0000000000000000
io_get_key 22
File ed6787e18b12543e 0000000000000000 0000000000000000
io_get_key 22
File d603c7e177f13c40 0000000000000000 0000000000000000
[...]
File 6fc51949a75bfa98 7185392f20a4b231 59355a5be160fa69
File fbdf1af71dd4ddda 621479a299239497 643a6d84ed302640
File ed6787e18b12543e 4009a64d7fb824bb e918bd0f49702d0b
File d603c7e177f13c40 88eddef95e436fdb 97e26f70517eea1f
io_get_dbg -> 0
[...]
---EXEC OUTPUT END---
```

Ces clés nous permettent alors de lire le répertoire « prod » :

```
[
  {
    "name": "914f6f6e67591ac4d03baa5110c9c5322eec7ace16f311233bfe3f674d93a2bc.enc",
    "real_name": "Canal_Historique.mp4",
    "type": "mp4",
    "perms": "0000000000001000",
    "ident": "ed6787e18b12543e"
  },
  {
    "name": "a24fad5785bd82f71b184100def10e56e9b239930ad06cfe677f6a8d692e452c.enc",
    "real_name": "flags.txt",
    "type": "txt",
    "perms": "0000000000000000",
    "ident": "fbdf1af71dd4ddda"
  }
]
```

Le fichier flags.txt nous donne l'ultime flag :

```
$ python ../decrypt.py a24fad5785bd82f71b184100def10e56e9b239930ad06cfe677f6a8d692e452c.enc
643a6d84ed302640621479a299239497
SSTIC{bf3d071f5a8a45fabc549d54be841f8b}
```

Etape finale : l'adresse e-mail

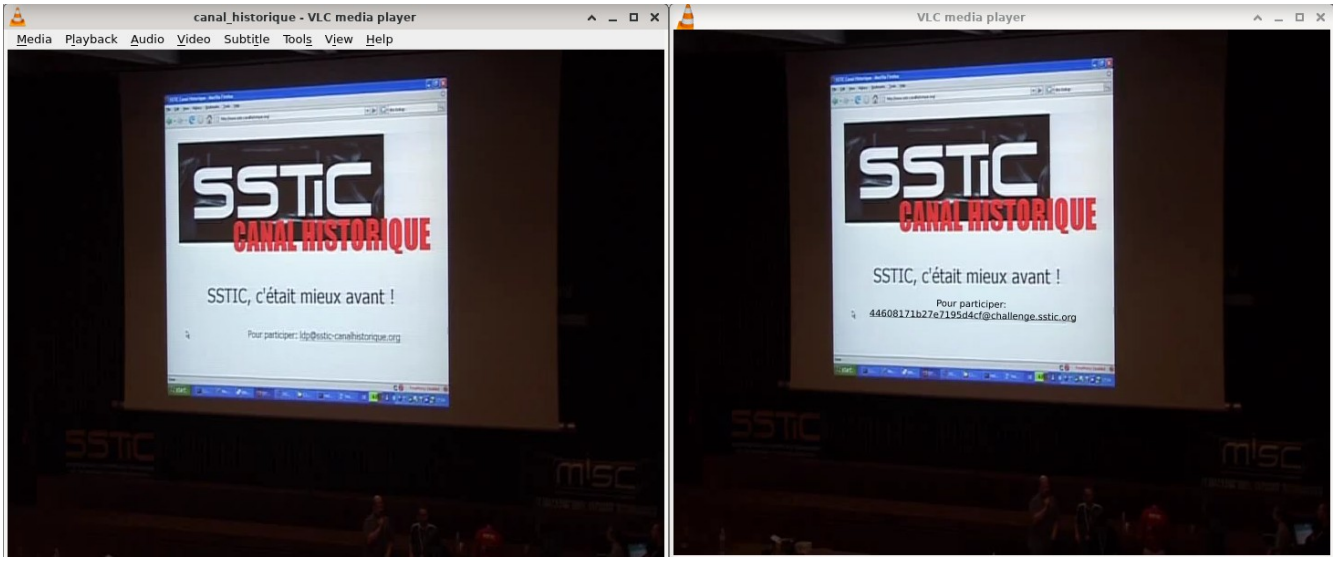
L'épreuve finale se trouve dans le fichier « Canal_Historique.mp4 ». Cette vidéo montre la rump réalisée à l'occasion des 10 ans du SSTIC par les pères de la conférence.

Une adresse e-mail apparaît dans l'un des slides, mais ce n'est pas celle recherchée.

MP4Info donne les informations suivantes sur le fichier :

```
$ mp4info canal_historique
mp4info version -r
canal_historique:
ReadChildAtoms: "canal_historique": In  atom, extra 1 bytes at end of atom
Track  Type   Info
1      video  H264 High@3, 97.400 secs, 166 kbps, 720x576 @ 25.000000 fps
2      audio  MPEG-4 AAC LC, 97.344 secs, 128 kbps, 48000 Hz
3      video  H264 Unknown Profile f4@3, 97.400 secs, 144 kbps, 720x576 @ 25.000000 fps
ReadChildAtoms: "canal_historique": In  atom, extra 1 bytes at end of atom
 Encoded with: Lavf58.45.100
```

On remarque la présence d'une seconde piste vidéo. En activant cette piste dans VLC, l'un des slides a quelque peu changé :



Conclusion

Merci aux auteurs du challenge de cette année, encore un excellent cru !
J'ai beaucoup appris, en particulier sur la gestion de la mémoire par le noyau Linux. La présente solution va un peu vite sur certains aspects, mais le lecteur curieux pourra se référer au code joint. Celui-ci ayant été nettoyé rapidement avant l'envoi, il est possible que des erreurs subsistent.