Solution du challenge SSTIC 2023

Rémy O.

8 juin 2023

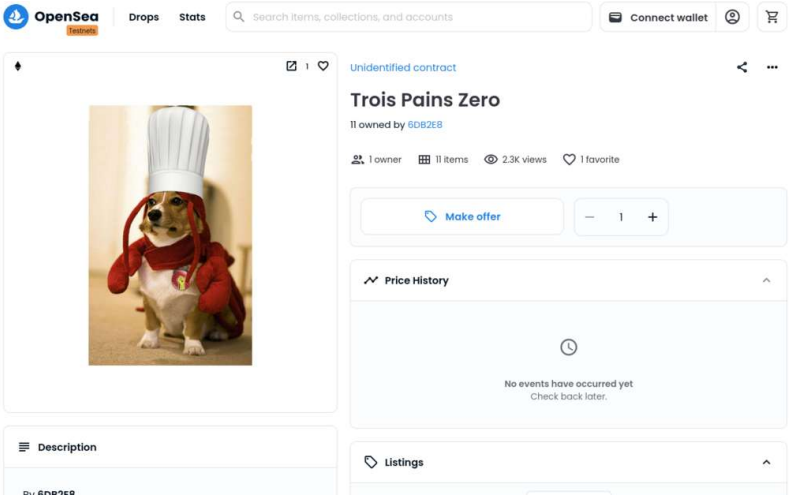
Introduction

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.

`https://testnets.opensea.io/assets/goerli/0x43F99c5517928be62935A1d7714408fae90d1896/1`

Étape 0

0. La page Opensea



0. Le contrat

On peut lire le code source du contrat:

- Onglet `About` > `Etherscan`
- `https://goerli.etherscan.io/address/0x43f99c...0d1896`
- Onglet `Contract`
- Contract Source Code > `TroisPainsZeroJNF.sol`

```
string constant BASE_URI =
  'data:application/json;base64,eyJuYW1lIjogIlRyb2lzIFBhaW5zIFplcm8iLAogI
  CAgICAgICAgImRlc2NyaXB0aW9uIjogIkxvYnN0ZXJkb2cgcGFzdHJ5IGNoZWYuIiwKICAg
  ICAgICAgICJpbWFnZSI6ICJodHRwczovL25mdC5xdWF0cmUtcXUuYXJ0L25mdC1saWJyYXJ
  5LnBocD9pZD0xMiIsCiAgICAgICAgICAiZXh0ZXJuYWxfdXJsIjogImh0dHBzOi8vbmZ0Ln
  F1YXRyZS1xdS5hcnQvbmZ0LWxpYnJhcnkucGhwP2lkPTEyIn0K';
```

0. Solution


```
{"name": "Trois Pains Zero",
 "description": "Lobsterdog pastry chef.",
 "image": "https://nft.quatre-qu.art/nft-library.php?id=12",
 "external_url": "https://nft.quatre-qu.art/nft-library.php?id=12"}
```

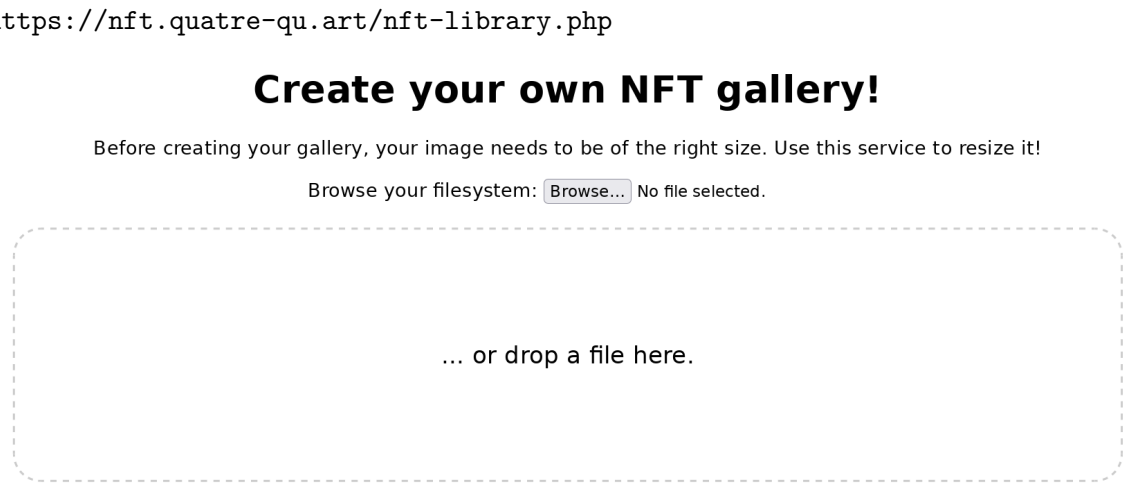

On trouve facilement en explorant: `https://nft.quatre-qu.art/nft-library.php?id=1`

`SSTIC{6a4ec745c1403b1ebf09fbd5a3021d1226330197641d4f65008ba0cd0fe48c62}`

Étape 1

1. Formulaire de la galerie

`https://nft.quatre-qu.art/nft-library.php`



1. En-têtes HTTP

Les en-têtes sont “un peu” suspects:

```
HTTP/1.1 200 OK
Server: nginx/1.18.0
Date: Fri, 31 Mar 2023 17:35:45 GMT
Content-Type: image/png
Content-Length: 0
Connection: keep-alive
X-Powered-By: ImageMagick/7.1.0-51
```

Et la version d'ImageMagick est **très** suspecte

1. CVE-2022-44268

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

Le correctif est dans la version 7.1.0-52

```
04ee6cec5 (tag: 7.1.0-52) release
...
05673e63c possible DoS @ stdin (OCE-2022-70);
          possible arbitrary file leak (OCE-2022-72)
09e738e84 OCE-2022-70: DoS at Stdin
73dd9deac eliminate unnecessary file open when globbing ...
839984c93 cosmetic
d3539aed9 Pass image's type instead of colorspace to ...
7b771b436 check extension attribute type to set the alpha channel
42bae95a1 support optional extension area
bf925a7f1 beta release
aea87b538 (tag: 7.1.0-51) release
```

1. Exploit

PoC pour CVE-2022-44268:

\x89 P N G \r \n \x1a \n			
len (u32)	IHDR	.....	CRC (u32)
len (u32)	IDAT	.....	CRC (u32)
len (u32)	tEXt	"profile nft-library.php"	CRC (u32)
0 (u32)	IEND		CRC (u32)

Les données sont dans une section `zTXt` (compression zlib, lire avec Python PIL par exemple).

On peut extraire:

- `nft-library.php` (flag)
- `/backup.tgz` et `/devices.tgz` (étape 2)

```
<?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
```

Étape 2

2. Les Quatre Quarts

Les tarballs contiennent:

- 4 challenges permettant de trouver 4 clés privées
- les sources du site `https://trois-pains-zero.quatre-qu.art/` pour les étapes 2 et 3

L'authentification sur le site se fait par une multisignature.

> *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.*

Étape 2A

2.a. Introduction

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.*

2.a. Le protocole MuSig2

- `https://eprint.iacr.org/2020/1261`
- Protocole multi-signature
- Signatures de type Schnorr

Instancié ici avec:

- 4 participants
- La courbe elliptique SECp256k1 (générateur G)

Les 4 clés publiques sont fournies.

2.a. Le protocole MuSig2

Signature de Schnorr:

- Message M, une fonction de hachage H
- Clé publique $L = \ell G$ (clé privée ℓ)
- Un *nonce* r et $R = rG$
- Une signature (s, R)

$$s = r + H(M)\ell \qquad sG = R + H(M)L$$

Si H est fixé on peut combiner des signatures avec des coefficients a_i

$$\Big(\sum a_i s_i\Big)G = \sum a_i R_i + H(M)\sum a_i L_i$$

Pour MuSig2, on ne publie pas vraiment une signature valide mais le principe reste similaire (les R_i sont saucissonnés).

2.a. Nonce déterministe

On est fortement incités à regarder de ce côté :

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

Le problème est mentionné dans les articles (ePrint 2020/1261 et 2018/068).

2.a. Équations

Pour chaque message, le participant A publie une presque-signature:

$$s_i = \left((\ell_A m)^{h(1)} + (\ell_A m)^{h(2)} b + (\ell_A m)^{h(3)} b^2 + (\ell_A m)^{h(4)} b^3 \right) + c(a_A \ell_A)$$

c est le condensat du message

L'inconnue est la clé privée ℓ_A

C'est une équation linéaire pour les inconnues ℓ_A et $\ell_A^{h(i)}$.

2.a. Données

Dans le fichier `logs.txt` on a les traces de 5 sessions:

Message m:

```
LOG: MESSAGE TO SIGN: b'250 grammes de beurre'
LOG: RECEIVED: b'250 grammes de beurre'
```

Points partiels $R_{A,j}$:

```
LOG: SENT:
0xfa50e69c485cde4664a97f8f7cbbf0b11dfc06b2d36e1f59dbe722736c99f223
0x96d762b43f6a293141d7d7dd4a9024085bbc2de6e667d857de88ce1427ecfcd5
...
```

2.a. Données

Points R_j (servent à calculer b, $R = \sum b^j R_j$, c):

```
LOG: RECEIVED:
0xca0216f379a499e2e9773245267e3d7b1245750de4358ac2499b66ae0f45c211
0xbf9e67581992eb02a12b795cce5d6bdc794c1ee8129006a665dc958754773cce
...
```

Signature partielle s_i:

```
LOG: SENT:
0x57c314c11adfe86309032c70f227339866e8e47fed91e133e89556f218235d8
```

2.a. Équations

On a 5 sessions dans le log, on peut écrire un système linéaire:

$$\begin{pmatrix} ca_A & b^0 m_1^{h_A} & b^1 m_1^{h_B} & b^2 m_1^{h_C} & b^3 m_1^{h_D} \\ ca_A & b^0 m_2^{h_A} & b^1 m_2^{h_B} & b^2 m_2^{h_C} & b^3 m_2^{h_D} \\ ca_A & b^0 m_3^{h_A} & b^1 m_3^{h_B} & b^2 m_3^{h_C} & b^3 m_3^{h_D} \\ ca_A & b^0 m_4^{h_A} & b^1 m_4^{h_B} & b^2 m_4^{h_C} & b^3 m_4^{h_D} \\ ca_A & b^0 m_5^{h_A} & b^1 m_5^{h_B} & b^2 m_5^{h_C} & b^3 m_5^{h_D} \end{pmatrix} \begin{pmatrix} \ell_A \\ \ell_A^{h_A} \\ \ell_A^{h_B} \\ \ell_A^{h_C} \\ \ell_A^{h_D} \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{pmatrix}$$

Équations modulo n (l'ordre de la courbe, un nombre premier).

2.a. Solution

- On résout le système linéaire
- On vérifie que le vecteur solution est bien de la forme $(\ell, \ell^{h_A}, \ell^{h_B}, \ell^{h_C}, \ell^{h_D})$
- On vérifie que la solution ℓ_A est bien le logarithme discret de la clé publique A

2.a. Avec moins d'équations ?

Attention: on peut être tenté de dire que les équations sont des polynômes avec des grands exposants $h(1)$ etc. et calculer le PGCD (qui doit être le polynôme $X - \ell_A$).

Mais le PGCD de polynômes creux avec des grands exposants h_A, h_B, h_C, h_D n'est pas rapide à calculer !

2.a. Solution

Les flags sont chiffrés avec ECIES (Diffie-Hellman + AES).

Il suffit de la clé privée pour déchiffrer:

```
from ecies import decrypt
enc = open("encrypted_flags/deviceA.enc", "rb").read()
priv = 0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
print(dec(priv.to_bytes(32, "big"), enc))
b'SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}\n'
```

Étape 2B

2.b. Description

On nous dit:

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

Il y a aussi un fichier de données `seed.bin`

2.b. Description

```
def get(self, i):
    g = self.gs[i]
    if g.tstamp < self.cycles:
        ...
        elif g.kind == 6:
            res = self.get(g.a) ^ \
              self.get(g.b) ^ g.n
        elif g.kind == 7:
            res = int(not self.get(g.a))
        elif g.kind == 8:
            if self.get(g.c):
                res = self.get(g.b)
            else:
                res = self.get(g.a)
        elif g.kind == 9:
            res = g.dff ^ g.n
        ...
```

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

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

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

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

2.b. Description

- Un script Python avec un fichier de données
- Environ 6000 portes logiques
- Des portes de type DFF pour stocker un état
- Une horloge qui évalue le circuit
- 2 bits du mot de passe sont entrés dans le circuit tous les 2 cycles

2.b. Solution avec Z3

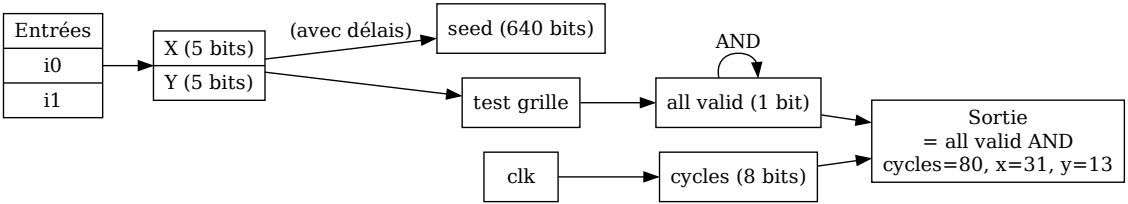
On convertit le circuit en formule logique pour Z3.

C'est fini:

```
995b90996f4564409191
```

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

2.b. Détail du circuit



- 3 registres principaux (X, Y, Cycles)
- Une fonction de validation avec un accumulateur
- Un circuit de dérivation de clé (640 bits en sortie)

Il suffit de comprendre la fonction de validation

2.b. Analyse des registres

On ignore le circuit de dérivation de clé: il ne reste que 20 registres!

On affiche leur formule. On trouve le compteur de cycles en 8 bits:

```
g5565 = Xor(g5565, True)
g3684 = Xor(g3684, g5565)
g1868 = Xor(g1868, And(g3684, g5565))
g288 = Xor(g288, And(g1868, And(g3684, g5565)))
g5358 = Xor(g5358, And(g288, And(g1868, And(g3684, g5565))))
g2078 = Xor(g2078, And(g5358, And(g288, And(g1868, And(g3684, g5565)))))
g3041 = Xor(g3041, And(g2078, And(g5358, And(g288, And(g1868,And(g3684,g5565))))))
g3415 = Xor(g3415, And(g3041, And(g2078, And(g5358, And(g288,And(g1868,And(...)))))))
```

2.b. Analyse des registres

On isole aussi 10 portes DFF qui dépendent des 2 bits d'entrée.

Les 4 entrées possibles donnent:

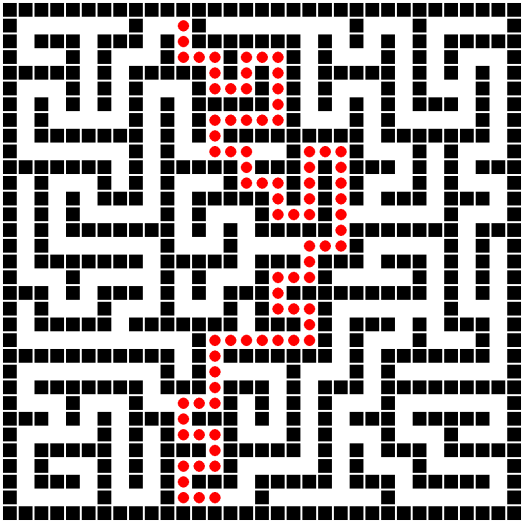
- deux registres 5 bits X et Y
- 00: décrémente X
- 01: incrémente Y
- 10: incrémente X
- 11: décremente Y

Interprétation :

- On contrôle un curseur (X, Y)
- Les 2 bits d'entrée définissent la direction
- Chaque direction est utilisée 2 fois

2.b. Labyrinthe

La fonction de validation est un prédicat sur la valeur des registres X et Y.
On calcule sa table de vérité complète (32×32).



Étape 2.c

2.c. Introduction

— 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.

Fichiers fournis:

- `frontend_service.bin`
- `remote_lib.so.6` (glibc)
- `ld-linux-aarch64.so.1`

2.c. Premier service

Service réseau interactif:

```
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:

Option: E
A. Add data
V. View data
E. Encrypt data
B. Back to main menu
```

2.c. Premier service

Les opérations sont Add, View, Encrypt, Decrypt, Sign

- Add ajoute un message dans le service (capacité: 10)
- View affiche les messages actuels
- Encrypt chiffre les messages et affiche le résultat
- Decrypt déchiffre les messages et affiche le résultat
- Sign renvoie un message d'erreur ("non implémenté")

Un système d'exceptions gère les erreurs avec `setjmp` et `longjmp`

2.c. Problème avec une exception

```
cmd_add(int fd, int mode) {
  if (! DATA_HAS_FREE_SLOTS) raise("Cannot add more data\n");
  u64 idx = DATA_COUNT;
  DATA_COUNT = DATA_COUNT + 1;
  read_data_entry(fd, &DATA_ENTRIES[idx], mode);
  if (DATA_COUNT == 10) {
    DATA_HAS_FREE_SLOTS = false;
  }
  wrap_write(fd,"Data successfully added\n", 0x18);
}
```

On peut lever une exception dans `read_data_entry` !

Problème de logique avec `DATA_COUNT`.

2.c. Exploitation

Disposition mémoire:

(12B)	——— Données 0 (256B) ———	(8B)
(12B)	——— Données 1 (256B) ———	(8B)
(12B)	——— Données .. (256B) ———	(8B)
(12B)	——— Données 9 (256B) ———	(8B)

DATA_COUNT (u32)	——————— (272B) ———————

(52B)	x19	x20	...	x30 xor K	0	SP xor K

2.c. Exploitation

En lecture:

- utiliser l'exception pour avancer le compteur à 12
- utiliser la fonction `View` pour afficher les registres (et casser l'ASLR)

En écriture:

- utiliser un gadget de `ld-linux` (fourni) pour charger `/bin/sh` dans `x0` et `system()` dans `x30`
- avancer le compteur à 11 et écraser `x19`, `sp`, `pc`

```
13788 <init_tls+0x11c>:
    13788:    aa1303e0    mov x0, x19
    1378c:    a94153f3    ldp x19, x20, [sp, #16]
    13790:    a9425bf5    ldp x21, x22, [sp, #32]
    13794:    a8c37bfd    ldp x29, x30, [sp], #48
    13798:    d65f03c0    ret
```

2.c. Exploitation

- On écrase l'emplacement des registres sauvés par `setjmp`
- On déclenche une exception en envoyant un CRC invalide (par exemple)
- On obtient un shell interactif

2.c' Second service

Avec le shell on découvre:

- un système presque vide (VM + kernel + initramfs)
- un 2e service sous un autre utilisateur
- un binaire corrompu `/home/backendUser/update-unit`
- pas de vulnérabilité système évidente

On hérite des FD du 1er service: le FD 5 est une connexion au 2e service.

2.c' Second service

Dans `update-unit` toutes les sections ELF sont supprimées sauf `.text`.

On ne voit pas les noms des fonctions de la libc, il faut les deviner.

2.c' Protocole et structure de données

Dans le protocole réseau, on envoie un identifiant de commande et une entrée:

- 1337 Add
- 1338 Encrypt
- 1339 Decrypt
- 133a Sign
- 133b Hello / Bye
- 133c Login (password)
- 133d Dump FW
- 133e Get Key (secret!)

Le protocole est visible dans le 1er service sauf la commande 133e

Dec/Enc? u8	Pad 3B	Taille (dec) u32	Index u32	Données 256B	CRC32 u32	Taille (enc) u32

2.c' Protocole et structure de données

Pour récupérer la clé de chiffrement AES:

- connaître le mot de passe de la commande `Login` (non)
- l'utiliser dans la commande 133E

```
getkey() {
   local_ptr = &AES_KEY
   char local_buf[32];
   if (memcmp(COMMAND.data, PASSWORD, 32) == 0) {
      send key
   } else {
      ???(local_buf, 33, COMMAND.data) // c'est un snprintf !
      send zero
      memcpy(reply, local_buf, 32)
      send local_buf
   }
}
```

2.c' printf

Exemple: si on envoie %p %p %p %p dans le corps de la commande

```
0x25 0x32 0x4 0xffffcd97f410
```

L'adresse de la clé est stockée dans sp+40 (14e argument)

```
1170: b00000a0       adrp    x0, 0x16000 <.text+0x224>
1174: 91004000       add     x0, x0, #16
1178: f90017e0       str     x0, [sp, #40]
```

Si on envoie %10$p %11$p %12$p:

```
0xffffd8effde0 0xaaaab5d66010 0xaa [tronqué à 32B]
```

Il suffit d'envoyer %11$s pour avoir la clé

```
04 c6 cb 31  e7 f3 ba 69  4c c0 1f 50  d6 57 3f 8d
22 be 2e 1b  d7 86 1e 17  6d 5b 4e d4  3c 13 f9 f9
```

2.c' Ou pas printf

On peut aussi se tromper et penser que tout est un `memcpy` (wtf):

```
memcpy(local_buf, 33, COMMAND.data)
send zero
memcpy(reply, local_buf, 32)
```

et chercher un autre bug (imprévu!)

2.c' Ou pas printf

2.c' API du service

- Au démarrage le service choisit 10 IV aléatoires
- Ils ne changent pas pendant une session (sauf si on les perturbe)
- On peut les trouver en déchiffrant des blocs de zéros
- La clé est statique

2.c' Erreur de logique

Vérification de longueur (version simplifiée) dans la commande `Add data`

```
ok = encrypted ? (enc_size <= 256) : (dec_size <= 256);
if (!ok) return false;
```

Une seule des 2 longueurs est vérifiée.

Les fonctions `encrypt` et `decrypt` ne vérifient pas le type de l'entrée.

On met une longueur valide et une longueur invalide. On obtient des appels `encrypt` et `decrypt` de longueur arbitraire.

2.c' Primitive XOR avec CBC

On applique encrypt(IV) puis decrypt(IV $\oplus$ X)

$$A \mapsto \mathrm{Enc}(A \oplus IV) \mapsto A \oplus X$$

Généralisation à 2 blocs

Opération	Bloc 1	Bloc 2
-	A	B
encrypt(IV,2)	A'	Enc(B $\oplus$ A')
encrypt(IV,1)	Enc(A' $\oplus$ IV)	Enc(B $\oplus$ A')
decrypt(IV$\oplus$X,1)	A' $\oplus$ X	Enc(B $\oplus$ A')
decrypt(IV,2)	??	B $\oplus$ X

2.c' Primitive XOR avec CBC

On itère, on peut faire `XOR X` sur un bloc arbitraire.

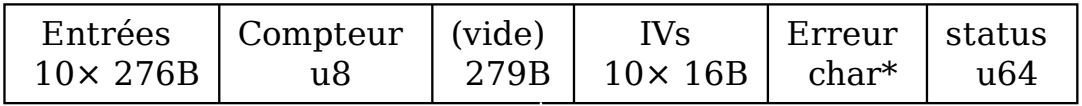
Mais les blocs précédents sont «corrompus».

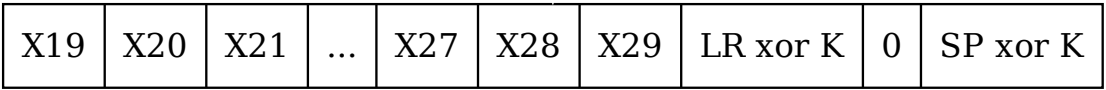
Il suffit de faire les opérations à l'envers, sans toucher au dernier bloc!

Attention: l'implémentation peut être périlleuse.

2.c' Exploitation

Disposition mémoire:





On utilise le XOR pour modifier le *link register* vers un gadget et le registre SP.

2.c' Exploitation

Gadget “dump firmware”: renvoie 256 octets depuis la pile

On appelle `decrypt` pour avoir l'expansion de clé AES sur la pile.

On obtient 24 octets de la clé et il suffit de faire 2 XOR pour les 8 octets restants.

```
01 00 00 00  36 13 00 00  00 00 00 00  00 01 00 00
00 00 00 00  4c c0 1f 50  d6 57 3f 8d  22 be 2e 1b <= la fin de la clé
d7 86 1e 17  6d 5b 4e d4  3c 13 f9 f9  78 5f 52 da
9f ac e8 b3  d3 6c f7 e3  05 3b c8 6e  49 5c c6 84
9e da d8 93  f3 81 96 47  cf 92 6f be  35 f7 fc 50 <= clés de tours
aa 5b 14 e3  79 37 e3 00  7c 0c 2b 6e  59 a2 37 1b
c7 78 ef 88  34 f9 79 cf  fb 6b 16 71  4e b0 5f 5f
e4 eb 4b bc  9d dc a8 bc  e1 d0 83 d2  a1 d2 db ae
66 aa 34 26  52 53 4d e9  a9 38 5b 98  41 89 19 8c
a5 62 52 30  38 be fa 8c  d9 6e 79 5e  94 4d 6d f6
f2 e7 59 d0  00 ee 71 d6  ff ff 00 00  6c 22 7a bf <= pointeurs divers
aa aa 00 00  48 f0 71 d6  ff ff 00 00  01 00 00 00
00 00 00 00  00 80 c4 89  ff ff 00 00  ad 21 6c dd
3d 4e 15 60  30 ee 71 d6  ff ff 00 00  70 23 7a bf
```

Étape 2.d

2.d. Introduction

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`.

2.d. Données

Forte suggestion d'attaque par canaux auxiliaires.

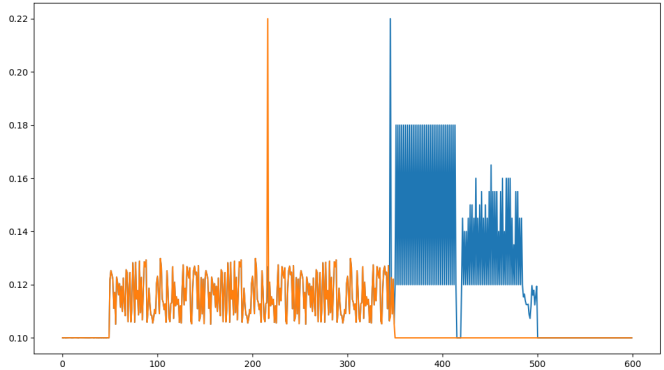
Point de départ: un fichier HDF5 `data.h5`

Suggestions:

- Utiliser le module Python `tables` pour lire les fichiers HDF5
- Utiliser les fonctions `numpy` pour vectoriser les calculs pendant la phase d'étude
- Utiliser `matplotlib` ou Jupyter pour afficher les courbes

2.d. Données

Traces de consommation (25000 échantillons)



2.d. Modèle de corrélation

Modèle standard (*Correlation Power Analysis*): corrélation entre la consommation et le poids de Hamming W de l'entrée `KEY[i] ^ MASK[i]`

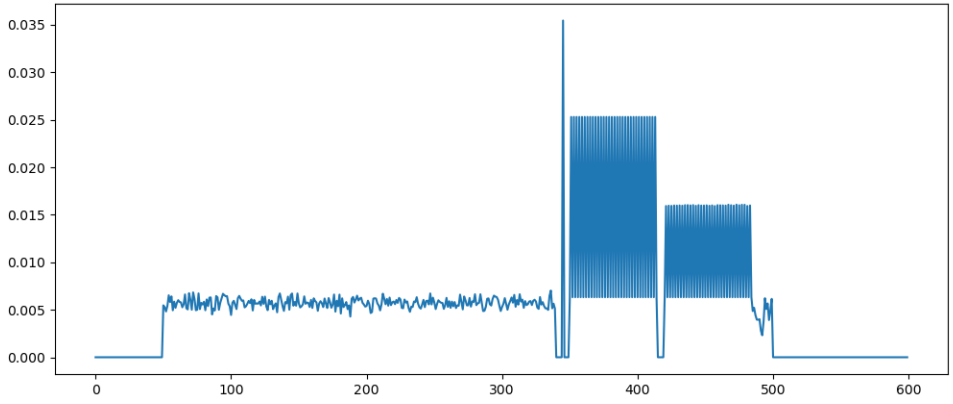
On peut aussi corréler les bits indépendants: selon les bits de la clé

$$W = \sum \pm \texttt{MASKBIT[i]}$$

$$\mathrm{Cov}(w, \text{Signal}) = \sum \pm \mathrm{Cov}(\texttt{MASKBIT[i]}, \text{Signal})$$

2.d. Écart-type

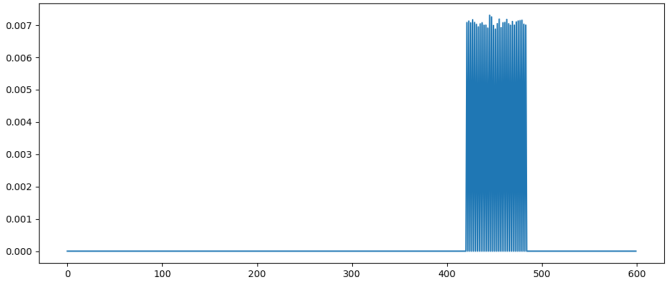
Écart-type de chaque valeur du signal (sur $t < 350$ on a un signal constant sauf un glitch à 0.22 sur un point aléatoire)



2.d. Écart-type avec un filtre

Écart-type de chaque valeur du signal (filtré sur les signaux intéressants à $t > 350$)

- Le glitch est toujours à t=345 exactement
- Tous les signaux sont identiques ($\sigma = 0$) sauf sur une zone

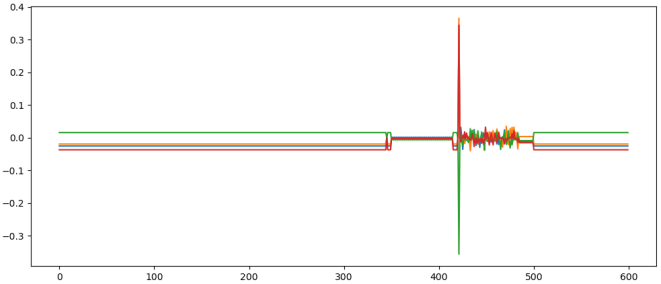


On voit que la partie intéressante est à t=421, 423, etc.

2.d. Corrélation

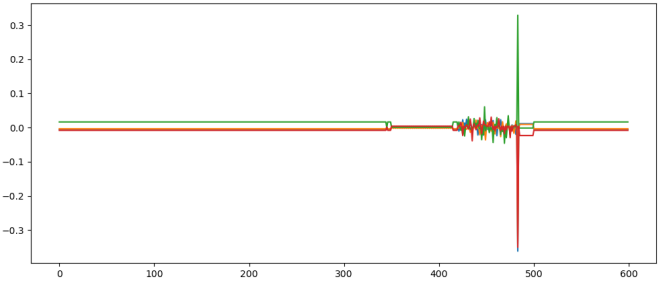
Corrélation entre le signal (filtré) et 4 bits de l'octet 1 du masque:

$$\text{Correl}(t) = \frac{\mathbb{E}[W \times \text{Signal}(t)]}{\sigma(W)\sigma(\text{Signal}(t))}$$



2.d. Corrélation (encore)

Corrélation entre le signal (filtré) et 4 bits de l'octet 32 du masque:



Tout se passe à $t = 483$ avec une corrélation (trop) «parfaite» $\pm 1/\sqrt{8}$

2.d. Solution

- Corrélation proche de $+1/\sqrt{8} \implies$ bit de la clé à 0
- Corrélation proche de $-1/\sqrt{8} \implies$ bit de la clé à 1

```
import tables, numpy as np
h5 = tables.open_file("data.h5")
masks = np.array(h5.get_node("/mask"))
leaks = np.array(h5.get_node("/leakages"))
masks = masks[leaks[:,450] > 0.11]
leaks = leaks[leaks[:,450] > 0.11,:]
stdev = np.std(leaks, axis=0)
bits = ""
for bit in range(256):
    weights = (masks[:,bit // 8] >> (7 - bit % 8)) & 1
    w = weights - np.average(weights)
    correl = np.tensordot(w, leaks, [0,0]) / len(w) / stdev / np.std(w)
    bits += "0" if np.max(correl) > 0.25 else "1"
key = int(bits, 2).to_bytes(32, "big")
```

Étape 3

3. Connexion au site

Pour se connecter on doit signer un message aléatoire:

```
We hereby authorize an admin session of 5 minutes
starting from 2023-04-13 11:13:26.392313+00:00
(nonce: 2a43111f887bc3fad5c4943471637604).
```

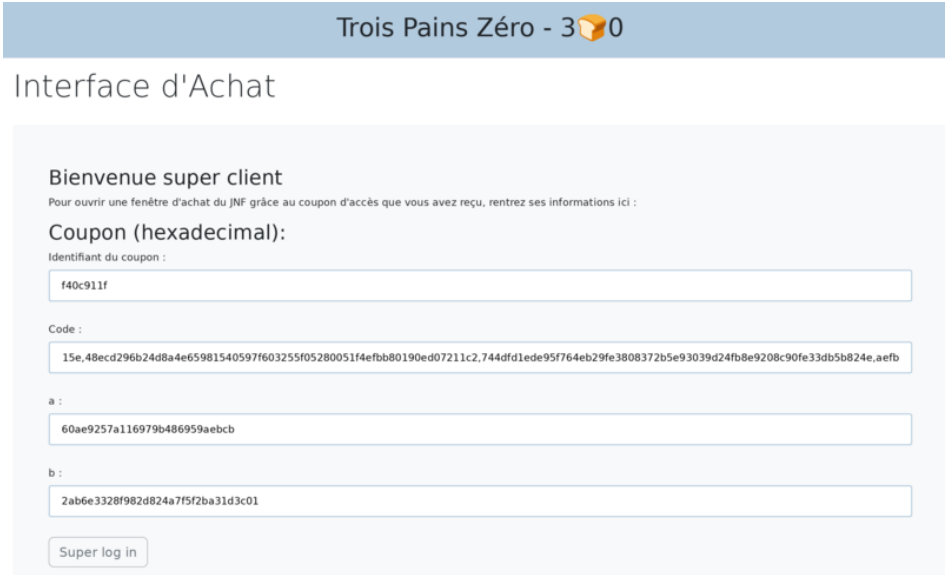
avec les 4 clés et le protocole Musig2.

Pas besoin du protocole entier, la clé privée agrégée suffit:

$$\ell = \sum a_i \ell_i$$

$$(R, s) = (G, 1 + H(m) \times \ell) \implies sG = R + H(m)L$$

3. Page d'achat



3. Contrat

L'achat est validé par un contrat Starknet:

```
def is_valid(ans: int, code: list[int], a: int, b: int):
    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
```

3. Exploration

Comme dans l'étape 0, on cherche le code du contrat avec l'API StarkNet:

```
c = FullNodeClient(node_url=URL, net="testnet")
for b in range(12):
    print(c.get_transaction_by_block_id_sync(
      block_number=b, index=0))
```

On trouve 10 blocs avec 1 transaction chacun.

3. Exploration

Dans les transactions:

- Transaction `declare` (bibliothèque ERC20)
- Transaction `invoke` (fonction `deploy(contract)`)

On a l'adresse du contrat dans les «événements» émis par l'invocation:

0x6b0a96cac8fada00f85569b27c0feee4b2fb1923159c6673b0d3c8b5f5a2ceb

Et on a aussi des informations utiles pour la suite:

- arguments du constructeur: owner `0x4ece...8b4d` et valeur `0x5b65565f4e4fc51283f9b627d5a075d8`
- valeur `salt=0x1337`

3. Langage Cairo

- Modèle de programmation inhabituel
- Programmation par contraintes imitant l'impératif
- Mémoire immutable
- Nombres modulo $p = 2^{251} + 17 \times 2^{192} + 1$ (environ 251 bits)

3. Décompilation

Le contrat a deux fonctions `validate` et `get_owner`.

On appelle l'API StarkNet `starknet_getClassAt` pour obtenir le (byte)code complet

Outil de désassemblage / décompilation: `thoth`

3. Décompilation

```
func validate(id, code, a, b) {
  nc = self.nonce;
  let res = first(nc, code);
  second(res, a, b);
  let result = pedersen(nc, id);
  assert len(code) >= 3
  j(result, code)
}
```

3. Décompilation `first` et `second`

La fonction `first` calcule un hash (Pedersen) glissant:

```
reduce(pedersen, code, nc)
```

La fonction `second` exprime des contraintes:

- `a` et `b` sont le découpage de `res` en 2 entiers 128 bits
- la taille de `a` est moins de 108 bits

Si on a choisi `code` on peut ajuster pour que le hash soit petit.

On calcule `a` et `b`.

3. Décompilation j

La dernière fonction de validation est bizarre. On exécute des instructions écrites en mémoire:

```
j(hash, code) {
  blob = {
    BYTECODE code[2] // ignoré
    let x = hash
    let y = code[0]
    assert y = x * x
    let z = y * 0x1337
    let t = 0x1336
    assert t = z * code[1]
    BYTECODE hash * code[2]
  }
  call &blob[1]
}
```

3. Décompilation `j`

Les équations sont:

- `code[0] = hash * hash`
- `code[0] * 0x1337 * code[1] == 0x1336`
- `hash * code[2]` est l'encodage de l'instruction `RET` (`0x208b7fff7fff7ffe`)

Solution:

- On choisit `id`, on calcule `hash = pedersen(nonce, id)`
- On calcule `code[0]`, `code[1]`, `code[2]`
- On ajoute `code[3]` pour que `H = first(nonce, code)` soit petit
- On calcule `a = H >> 128` et `b = H & (2**128 - 1)`

On envoie `{id, code, a, b}`

Étape finale

Puzzle

```
montage {1..24}.png -tile 12x2 -background black puzzle.png
```

Solution

```
montage {10,14,24,9,16,23,3,19,11,22,1,5}.png \
  {8,18,4,20,13,15,17,7,6,21,2,12}.png \
  -geometry +0+0 -tile 12x2 -background black puzzle.png
```

```
To: 48ad57aa9c8c600d4c649a5232ef5abf@sstic.org
Subject: Information importante
```

Fin

Merci au Donjon de Ledger pour ce challenge !