Solution challenge SSTIC 2023

Morieux Florent

24 avril 2023

Présentation

Le challenge se déroule du 32 mars au 22 mai et le but est de valider 6 étapes grâce à des flags sous la forme SSTIC{...} avant de trouver une adresse mail finale permettant de valider la réussite du challenge.

Dans ce rapport je présente la solution qui m'a permis d'arriver au boût du challenge. Voici l'énoncé de cette année :

> En titubant dans la rue Saint-Michel, vous avez rencontré une personne coiffée d'une toque de pâtissier qui vous a tendu un tract.
> À tête reposée, vous l'avez lu et celui-ci contient le message suivant :
>
> Salud deoc'h !
> Votre nouvelle boulangerie Trois Pains Zéro a décidé d'innover afin d'éviter les files d'attente et vous permettre de déguster notre recette phare : le fameux quatre-quarts.
> À partir du 1er juillet 2023, il vous suffira d'acquérir un Jeton Non-Fongible (JNF) de notre collection sur OpenSea, et de le présenter en magasin pour recevoir votre précieux gâteau.
> La page d'achat sera bientôt disponible pour tous nos clients et nous espérons vous voir bientôt en magasin.
> Délicieusement vôtre,
> Votre boulangerie Trois Pains Zéro

Etape 0 : Une histoire de JNF.

On commence le challenge de cette année avec un lien OpenSea qui est une plateforme de marché en ligne pour les NFTs. On se retrouve alors sur la page nous permettant d'acheter ce magnifique NFT :

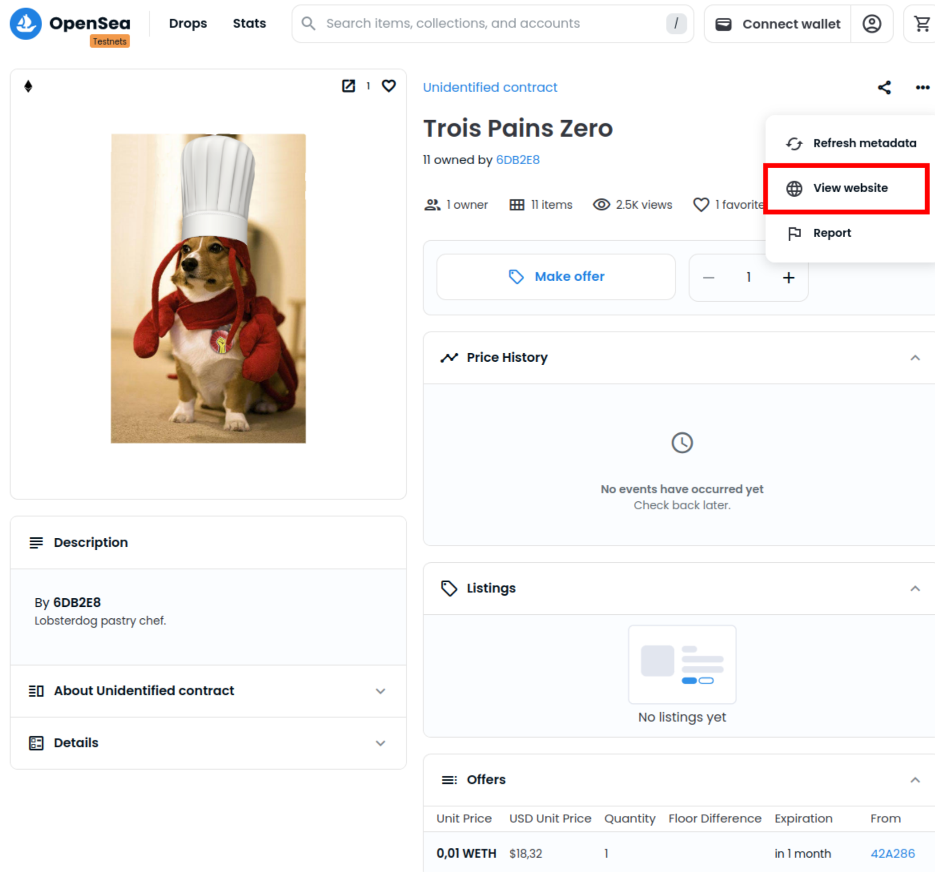

Figure 1: loBsterDoG.NfT

On commence donc par naviguer dans la page openSea jusqu'à cliquer sur le lien du site hébergeant l'image: `https://nft.quatre-qu.art/nft-library.php?id=12`

On modifie le paramètre "id" dans l'url avec la valeur 1 et on trouve le premier flag : `https://nft.quatre-qu.art/nft-library.php?id=1`

SSTIC{6a4ec745c1403b1ebf09fbd5a3021d1226330197641d4f65008ba0cd0fe48c62}

Etape 1 : ImageMagick

On continue donc en explorant le site et on y découvre un service nous permettant de redimensionner des images. En analysant un peu les messages échangés, on peut supposer que c'est ImageMagick 7.1.0-51 qui est utilisé par le serveur pour redimensionner les images :

Figure 2: HTTP request

Une recherche google nous permet de vite trouver une CVE concernant ImageMagick permettant notamment des lectures de fichiers arbitraires:
`https://github.com/duc-nt/CVE-2022-44268-ImageMagick-Arbitrary-File-Read-PoC`
On commence par essayer de remettre en oeuvre la méthode décrite sur la page github afin de récupérer un fichier sur le serveur :

Usage:

Installing dependencies:

```
1. $ apt-get install pngcrush imagemagick exiftool exiv2 -y
```

Change the filename you want to read below:

```
2. $ pngcrush -text a "profile" "/etc/hosts" vjp.png
```

Confirm everything worked perfectly

```
3. $ exiv2 -pS pngout.png
```

Trigger the PoC via convert or upload image to the vulnerable service:

```
4. $ convert pngout.png gopro.png
```

View content of file was read:

```
5. $ identify -verbose gopro.png
```

Figure 3: CVE-2022-44268

Ceci nous permet de récupérer rapidement le fichier `/etc/host` du serveur. On essaie ensuite de récupérer le fichier `/var/www/html/nft-library.php` mais sans succès. Il faut en effet faire faire attention à la fonction identify qui nous induit en erreur pour les fichiers qui ont une extension dans leur nom.

Pour résoudre ce problème, j'ai utilisé le module python **wand.image** qui permet de parser le png récupéré. Pour récupérer le fichier `nft-library.php` il faut regarder dans les profils du png et pas dans

le **"Raw profile type"**.

Finalement, on récupère le fichier `nft-library.php`:

```php
<?php
header("X-Powered-By: ImageMagick/7.1.0-51");

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

Figure 4: nft-library.php

Et le flag :

SSTIC{8c44f9aa39f4f69d26b91ae2b49ed4d2d029c0999e691f3122a883b01ee19fae}

Etape 2: Récupération des clefs privées

Enoncé de l'étape 2:

> 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 :

L'étape 2 consiste donc en 4 sous-étapes durant lesquelles on devra récupérer les clefs privées de 4 personnes différentes.

Etape 2.a: Clef privée de Jean-Michel A

Enoncé de l'étape 2.a:

> - 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.

On a à notre disposition le script `musig2_player.py` et un fichier de journalisation de signatures.

Après avoir rapidement survolé le code de `musig2_player.py` on se rend compte qu'un gros indice a été laissé dans les commentaires de la fonction qui génère les nonces :

```
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)
```

En effet, le nonce est totalement déterministe. On se renseigne ensuite sur le protocole musig2 afin de mieux comprendre comment utiliser cette information :
`https://eprint.iacr.org/2020/1261.pdf`

Setup(1^λ)

$(\mathbb{G}, p, g) \leftarrow \mathsf{GrGen}(1^\lambda)$
Select three hash functions
$\mathsf{H}_{\text{agg}}, \mathsf{H}_{\text{non}}, \mathsf{H}_{\text{sig}} : \{0,1\}^* \to \mathbb{Z}_p$
$par := ((\mathbb{G}, p, g), \mathsf{H}_{\text{agg}}, \mathsf{H}_{\text{non}}, \mathsf{H}_{\text{sig}})$
return par

KeyGen()

$x \leftarrow\$ \mathbb{Z}_p\,;\ X := g^x$
$sk := x\,;\ pk := X$
return (sk, pk)

KeyAggCoef(L, X_i)

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

KeyAgg(L)

$\{X_1, \ldots, X_n\} := L$
for $i := 1 \ldots n$ **do**
$a_i := \mathsf{KeyAggCoef}(L, X_i)$
return $\widetilde{X} := \prod_{i=1}^{n} X_i^{a_i}$

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

$\widetilde{X} := \widetilde{pk}\,;\ (R, s) := \sigma$
$c := \mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$
return $(g^s = R\widetilde{X}^c)$

Sign()

// Local signer has index 1.
for $j := 1 \ldots \nu$ **do**
$r_{1,j} \leftarrow\$ \mathbb{Z}_p\,;\ R_{1,j} := g^{r_{1,j}}$
$out_1 := (R_{1,1}, \ldots, R_{1,\nu})$
$state_1 := (r_{1,1}, \ldots, r_{1,\nu})$
return $(out_1, state_1)$

SignAgg$(out_1, \ldots, out_n)$

for $i := 1 \ldots n$ **do**
$(R_{i,1}, \ldots, R_{i,\nu}) := out_i$
for $j := 1 \ldots \nu$ **do**
$R_j := \prod_{i=1}^{n} R_{i,j}$
return $out := (R_1, \ldots, R_\nu)$

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

// Sign' must be called at most once per $state_1$.
$(r_{1,1}, \ldots, r_{1,\nu}) := state_1$
$x_1 := sk_1\,;\ X_1 := g^{x_1}$
$(R_{1,1}, \ldots, R_{1,\nu}) := (g^{r_{1,1}}, \ldots, g^{r_{1,\nu}})$
$(X_2, \ldots, X_n) := (pk_2, \ldots, pk_n)$
$L := \{X_1, \ldots, X_n\}$
$a_1 := \mathsf{KeyAggCoef}(L, X_1)$
$\widetilde{X} := \mathsf{KeyAgg}(L)$
$(R_1, \ldots, R_\nu) := out$
$b := \mathsf{H}_{\text{non}}(\widetilde{X}, (R_1, \ldots, R_\nu), m)$
$R := \prod_{j=1}^{\nu} R_j^{b^{j-1}}$
$c := \mathsf{H}_{\text{sig}}(\widetilde{X}, R, m)$
$s_1 := ca_1x_1 + \sum_{j=1}^{\nu} r_{1,j} b^{j-1} \bmod p$
$state_1' := R\,;\ out_1' := s_1$
return $(state_1', out_1')$

SignAgg'$(out_1', \ldots, out_n')$

$(s_1, \ldots, s_n) := (out_1', \ldots, out_n')$
$s := \sum_{i=1}^{n} s_i \bmod p$
return $out' := s$

Sign''$(state_1', out')$

$R := state_1'\,;\ s := out'$
return $\sigma := (R, s)$

Figure 5: Protocole musig2

Si on regarde comment les signatures de chaque signataire sont calculées on obtient, dans un cas à 4 signataires, l'équation suivante :

$$s_k = ca_1x_k + r_{k,0} * b^0 + r_{k,1} * b^1 + r_{k,2} * b^2 + r_{k,3} * b^3$$

où:
- s_k est la signature avant aggrégation du signataire k.

Si on remplace tous les termes que l'on connaît dans l'équation (en récupérant les valeurs contenues dans le fichier `logs.txt`), on se rend compte que, du fait du nonce déterministe, il ne nous manque plus que 5 inconnues qui sont:

$$x,\ x^{d0},\ x^{d1},\ x^{d2},\ x^{d3}$$

On obtient donc une équation de la forme :

$$s_k = a_0x_k + a_1x_k^{d_0} + a_1x_k^{d_1} + a_2x_k^{d_2} + a_3x_k^{d_3}$$

Sachant que l'on a à notre disposition les logs de signature de 5 messages différents on peut finalement sortir la clef privée x_k de ce système à 5 équations et 5 inconnues

Pour ce faire, on utilise le module python sympy :

```
from sympy import *

x, xd0, xd1, xd2, xd3 = symbols("x xd0 xd1 xd2 xd3")

eq1 = Eq(s1, c1a*x + xd0*m1d0b0 + xd1*m1d1b1 + xd2*m1d2b2 + xd3*m1d3b3)
eq2 = Eq(s2, c2a*x + xd0*m2d0b0 + xd1*m2d1b1 + xd2*m2d2b2 + xd3*m2d3b3)
eq3 = Eq(s3, c3a*x + xd0*m3d0b0 + xd1*m3d1b1 + xd2*m3d2b2 + xd3*m3d3b3)
eq4 = Eq(s4, c4a*x + xd0*m4d0b0 + xd1*m4d1b1 + xd2*m4d2b2 + xd3*m4d3b3)
eq5 = Eq(s5, c5a*x + xd0*m5d0b0 + xd1*m5d1b1 + xd2*m5d2b2 + xd3*m5d3b3)

solve([eq1,eq2,eq3,eq4,eq5], [x,xd0,xd1,xd2,xd3])
```

Le résultat est présenté sous la forme d'une fraction de 2 grands nombres. On calcule l'inversion modulaire du dénominateur, que l'on multiplie au numérateur avant d'appliquer le modulo. On obtient donc la clef privée qui va nous permettre de déchiffrer le premier flag de cette étape 2.

SSTIC{dc3cb2c61cb0f2bdec237be4382fe3891365f81a0fb1c20546d888247dd9df0a}

Etape 2.b : Clef privée de Bertrand

Enoncé de l'étape 2.b :

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

Tout ce que l'on a pour cette étape sont un script python `seedlocker.py` et un fichier binaire `seed.bin`.

On analyse le fichier binaire et on se rend compte qu'il n'est en réalité que le script python implémentant un circuit logique qui est décrit dans le fichier `seed.bin` qui ne serait donc qu'une sorte de bitstream. Le script python, après avoir chargé le circuit logique, va venir prendre un mot de passe en entrée qu'il va donner à son circuit logique qui va lui même dériver une seed Bip39 qui va finalement nous donner la clef privée.

Le mot de passe est entré dans le circuit 2 bits par 2 bits et chaque entrée de 2 bits est maintenue pendant 2 coups de clock.

Heuresement le circuit logique sort une variable intermédiaire *e.good* qui va déterminer si le mot de passe sera valide ou non.

On va donc commencer par tenter de visualiser graphiquement toutes les portes logiques dont dépend *e.good*. Je n'inclerai pas le graphique entier dans le rapport pour des raisons de taille et de lisibilité.

Première chose rassurante, on se rend compte que *e.good* dépend bien de notre entrée.

En analysant le circuit d'un peu plus prêt on se rend compte que *e.good* dépend de 3 conditions:

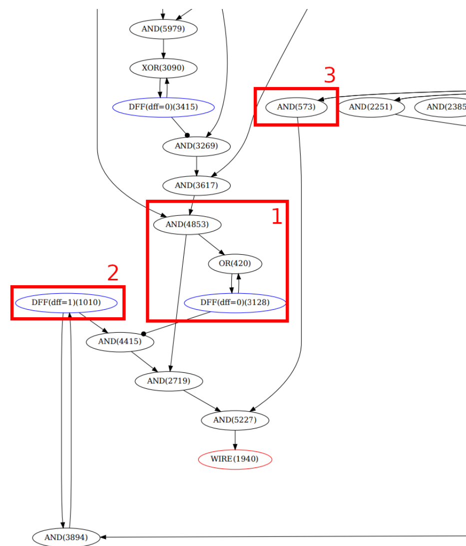

Figure 6: Circuit logique

- La première ne dépend pas directement de la valeur des entrées mais est un compteur qui compte le nombre de coups d'horloges total du circuit. Le nombre de coups d'horloge du circuit devra être de 80. Ce qui nous donne l'indication que le mot de passe devra faire exactement $80/8 = 10$ octets.

- La seconde condition découle directement de la valeur de 10 dff à l'état final.

- La troisième condition se termine par une dff qui est initialisée à 1 mais qui, si elle passe à 0 à un moment de l'exécution le restera. Cette dff dépend également uniquement de la valeur des mêmes 10 dff que pour la deuxième condition. La valeur des 10 dff est passée dans une sorte de filtre composé de portes **AND** et **OR**.

Les 10 dff dont dépendent la seconde et la troisième condition se découpent en 2 groupes de 5 dff. En jouant un peu avec les entrées du circuit et en affichant les valeurs de ces 2 groupes on se rend compte qu'il s'agit de 2 compteurs qui sont incrémentés ou décrémentés selon les 4 valeurs possibles en entrée ('00','01','10','11').

Après quelques heures de réflexion on se rend compte que ce qu'on a sous les yeux est en réalité un labyrinthe dont les 2 groupes de 5 dff représentent les coordonnées dans le labyrinthe. Les 3 conditions précédemment vues illustrent les règles du jeu :

- La première établit la longueur du chemin.

- La seconde définit le point de sortie du labyrinthe.

- La dernière définit les murs et chemins du labyrinthe.

En instrumentant une fois de plus le script python on ajoute une fonction nous permettant de forcer les valeurs des dff. Il ne nous reste plus qu'à énumérer les 2^{10} valeurs possibles pour les coordonnées et à demander la valeur en sortie pour chaque combinaison afin de savoir où sont les murs de notre labyrinthe 32x32 :

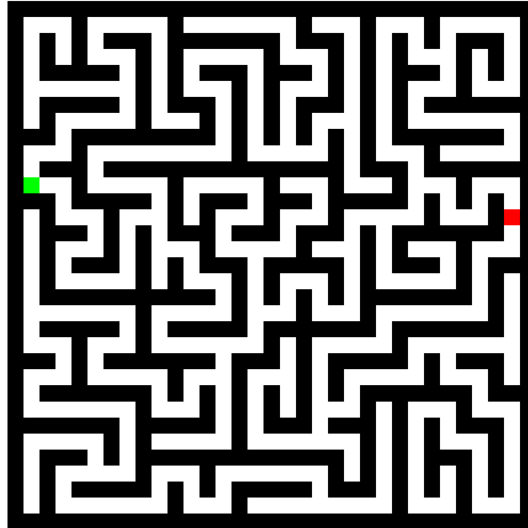

Figure 7: Labyrinthe

Enfin, on détermine exactement à quelle direction correspond quelle entrée de 2 bits. Il ne nous reste plus qu'à entrer le bon mot de passe et à déchiffrer le fichier contenant le flag avec la clef privée affichée :

SSTIC{f5967cae6478fa6bb9ea1bc758aee0961a68a8b4767f74888ce0bb8563a6218e}

Etape 2.c : Clef privée de Charles

Enoncé de l'étape 2.c :

> - 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.

Exploit du frontend

On commence par se connecter à l'équipement avec le mot de passe donné et après avoir résolu la preuve de travail. On se rend compte que ce dernier nous propose des options de chiffrement, de déchiffrement et de signature. Il possède également un mode admin qui nous demande un mot de passe.

La première étape consiste à reverse le binaire avec lequel on communique et à chercher un moyen ou un autre de récupérer la clef privée qui est utilisée par l'équipement.
Le premier point à noter est que l'équipement avec lequel on communique n'est en réalité qu'un frontend qui lui-même commande à un backend d'effectuer les chiffrements/déchiffrements.

Après avoir vu qu'il était possible de récupérer le firmware du backend en mode admin, on comprend donc qu'il va falloir réussir à exploiter le serveur afin de pouvoir analyser le backend.

Vulnérabilité

Le frontend, lorsqu'on lui demande de stocker des données à chiffrer va utiliser un booléen afin de savoir s'il reste de la place dans son tableau et un index qui va lui permettre de savoir combien de chunks sont déjà présents dans son tableau.

La vulnérabilité réside dans le fait que le frontend commence par incrémenter l'index, avant de gérer l'ajout de données et avant de mettre à 0 le booléen lui permettant de savoir s'il reste de la place dans le tableau. De plus, si l'ajout de données échoue, le binaire utilise la fonction longjump pour retourner dans la boucle principale du menu. Il est donc possible d'incrémenter l'index du tableau au delà des 10 chunks disponibles sans que le booléen indiquant qu'il n'y a plus d'espace dans le tableau soit set.

Maintenant, si l'on va voir en mémoire ce qui se trouve après ce tableau on retrouve notamment la structure utilisée par le longjump pour restaurer le contexte des registres. Il va donc être possible dans un premier temps de leak cette structure afin de contourner les probèmes d'ASLR mais aussi de cookie utilisé pour sauver les valeurs des registres. Puis dans un second temps de venir écraser ce contexte avant de provoquer une erreur afin d'appeler directement la fonction qui nous envoie le firmware du backend sans demander le mot de passe.

On récupère ainsi le firmware du backend...

Analyse du backend

Le firmware du backend est très ressemblant au firmware du frontend mais sans aucun symbole importé. Les fonctions de la libc ne sont pas nommées lorsque l'on reverse le binaire. Il est dans un premier temps intéressant de renommer toutes les fonctions afin de mieux comprendre le fonctionnement du backend (chose que l'on n'a pas fait immédiatement bien sûr et qui nous aura fait perdre prêt de 6 jours).

Voici les informations que l'on retire du reverse de ce binaire :

- La clef privée n'est pas présente dans le fichier récupéré,
- Le chiffrement ressemble très fortement à un AES-CBC,
- Il y a une commande additionnelle au niveau de la communication avec le frontend. Très intéressant !

Modification de l'exploit du frontend

Pour la suite il sera intéressant de pouvoir communiquer directement avec le backend depuis chez nous. Et pour ce faire, on va modifier l'exploit du frontend en une ropchain qui va venir:

- mprotect la stack en RWX,
- recv un shellcode depuis la socket utilisée par l'utilisateur,

- sauter dans ce shellcode.

On aura au préalable cross-compilé un shellcode effectuant une sorte de **socat** entre les 2 sockets (USER et BACKEND) du frontend:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/errno.h>
#include <sys/select.h>

#define BUFFER_SIZE 1024
#define SOCK1 4
#define SOCK2 5

typedef struct {
    int (*p_select)(int, fd_set *, fd_set *, fd_set *, struct timeval *);
    ssize_t (*p_send)(int, const void *, size_t, int);
    ssize_t (*p_recv)(int, void *, size_t, int);
    char* (*p_malloc)(ssize_t);
} Ctx_t;

void socat(Ctx_t *ctx) {
    fd_set read_fds;

    char* buffer = ctx->p_malloc(BUFFER_SIZE);

    while (1) {
        FD_ZERO(&read_fds);
        FD_SET(SOCK1, &read_fds);
        FD_SET(SOCK2, &read_fds);

        ctx->p_select(SOCK2 + 1, &read_fds, NULL, NULL, NULL);

        if (FD_ISSET(SOCK1, &read_fds)) {
            int bytes_received = ctx->p_recv(SOCK1, buffer, BUFFER_SIZE, 0);
            ctx->p_send(SOCK2, buffer, bytes_received, 0);
        }
        if (FD_ISSET(SOCK2, &read_fds)) {
            int bytes_received = ctx->p_recv(SOCK2, buffer, BUFFER_SIZE, 0);
            ctx->p_send(SOCK1, buffer, bytes_received, 0);
        }
    }
}
```

Figure 8: SOCAT

Il nous est maintenant possible de communiquer directement avec le backend.

Exploit du backend

Après avoir tenté pendant plusieurs jours d'exploiter une vulnérabilité au niveau des confusions de taille entre données à chiffrer et données à déchiffrer, et avoir réussi à leak le contenu du onzième chunk de données qui ne contenait que des zéros, on se repenche sur cette fameuse commande additionnelle.

Cette commande additionnelle vérifie le mot de passe de 32 caractères qu'elle reçoit avec un mot de passe qui n'est pas non plus contenu dans le binaire. Si ce mot de passe est correct elle nous renvoie directement la clef, mais il n'existe à première vue pas de moyen de récupérer le mot de passe. Si le mot de passe n'est pas bon, notre entrée est utilisée comme chaîne de format dans un snprintf et le contenu du buffer de destination nous est renvoyé.

Il y a donc une vulnérabilité de type format string. Après avoir augmenté dans ghidra le nombre d'arguments que prenait la fonction snprintf, on se rend compte que le onzième argument correspond à l'adresse de la clef privée. Il ne nous reste donc plus qu'à envoyer la chaîne de caractère "%11$s" afin de récupérer la clef privée.

On finit par déchiffrer le fichier contenant le flag de l'étape C :

SSTIC{ba75fa41a81c43c1095588250d45af850cfcec187ae269f2389829224ae6060b}

Etape 2.d : Clef privée de Daniel

Enoncé de l'étape 2.d :

> - 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 ?

On commence par télécharger le fichier et le parser à l'aide du module python h5py. On se rend compte qu'il contient des traces d'injections de fautes pour différents masques que l'on va dans un premier temps afficher. On retrouve alors 2 cas différents :

- Les traces qui correspondent à une injection de fautes qui n'a rien donné :

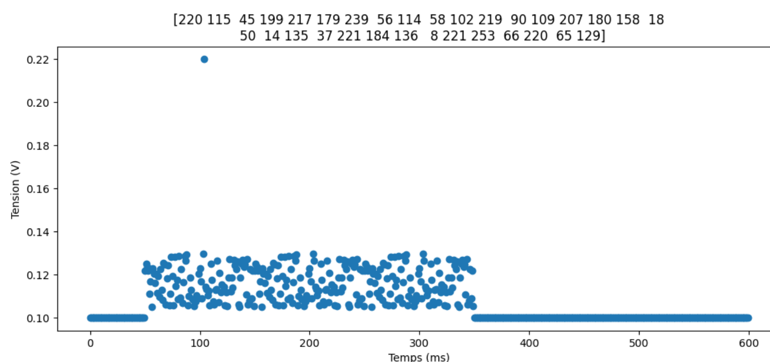

Figure 9: Injection ratée

- Les traces qui correspondent à une injection réussie :

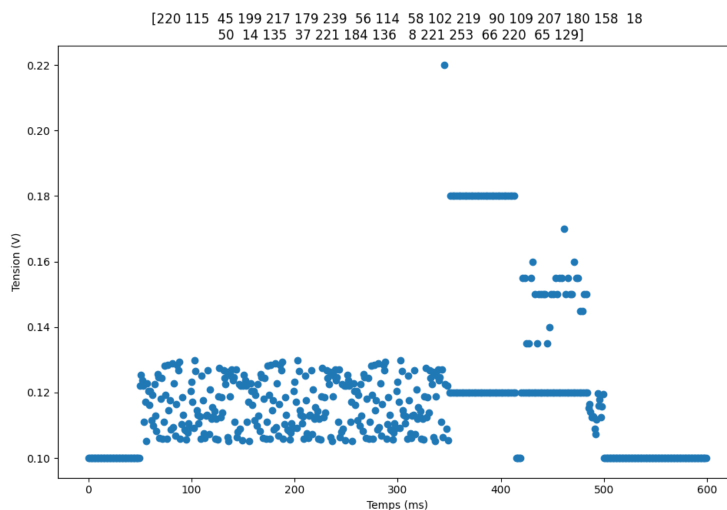

Figure 10: Injection ratée

On peut donc par la suite ne s'intéresser qu'aux traces qui ont donné un résultat intéressant... Si l'on y regarde de plus prêt on peut voir que vers la fin de la trace d'injection on retrouve 32 points un peu dispersés (comme les 32 octets de la clef privée):

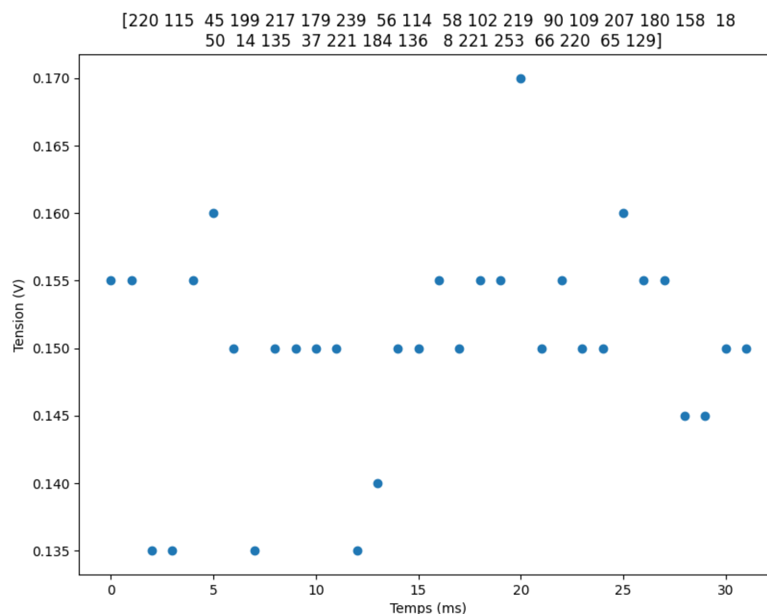

Figure 11: Injection ratée

Finalement en récupérant toutes les traces on peut déduire que chaque fois qu'un point a une valeur de courant maximale, l'octet du masque correspondant est correct. On parcourt alors toutes les courbes et on reconstitue alors une chaine de 32 octets. On tente de déchiffrer le fichier de clef correspondant à l'étape D avec ce que l'on pense être la clef, mais sans succès... En effet, il ne faut pas oublier de XOR la "clef" obtenue avec 0xffff... afin de récupérer la vraie clef privée et le flag :

SSTIC{15fb587e4dc04bbb7abb68fc6651f593d6eb0e4fd84bbfa800c6a66043bda86a}

Etape 3 : Du Cairo

Maintenant que l'on a réussi à récupérer les 4 clefs privées de nos boulangers préférés, il nous est possible de réimplémenter la partie aggrégation de clefs du protocole musig2 et ainsi de pouvoir signer le message qui nous est affiché :

Figure 12: Authentification

Une fois connecté, on se retrouve une fois de plus face à un formulaire qui prend différentes entrées :

- Un **ID de jeton**,
- un champ **code** sous forme de liste,
- une valeur **a**,
- une valeur **b**.

Comme nous possédons le code du serveur il nous est possible d'aller voir ce qui est fait avec ces différentes entrées. En réalité le serveur va appeler la méthode validate d'un smart contract qui se trouve sur une blockchain.

On commence donc par parcourir un peu la blockchain en utilisant le module python starknet_py afin de retrouver l'adresse du smart contract qui nous intéresse. En affichant les blocs déjà présents on se rend compte que plusieurs semblent faire un appel à une méthode d'un smart contract et en essayant différentes valeurs présentes dans ces blocs on finit par retrouver le smart contract qui contient la méthode **validate**. Il est possible d'utiliser l'utilitaire **thoth** afin de récupérer le code désassemblé ou décompilé du smart contract.

Après un analyse du code de la fonction validate on se rend compte que cette dernière appelle plusieurs fonctions:

- La première est la fonction **first** qui calcule le hash de pedersen du tableau **code** que nous lui avons donné.

- La fonction second calcule une preuve de travail sur a et b tels que $(a*2^{128}+b) \ mod \ p = hashed_code$. Des conditions sont mises sur les valeurs de **a** et **b** qui vont surement nous obliger à effectuer un peu de bruteforce par la suite.

- la fonction **j** qui est en réalité une sorte de texte à trou mettant des conditions de validité sur l'**id** du jeton et les différents champs de **code**.

On extraie donc les conditions suivantes :

$$code[0] == pedersen_hash(id)^2$$

$$code[0] * 4918 == code[1] * 4919$$

$$code[2] * pedersen_hash(id) == ret$$

En effet la dernière valeur va être exécutée et devra contenir l'instruction "ret" afin de bien compléter ce "texte à trous".

Il ne nous reste plus qu'à trouver une valeur d'**id** qui nous permettra de passer les 2 conditions sur **a** et **b**.

Le script python suivant nous donne les différentes valeurs valides pour le formulaire :

```python
import sys

NONCE = 121485921437276981477059375547635758552
PRIME = 0x800000000000011000000000000000000000000000000000000000000000001
RET = 0x208b7fff7fff7ffe

if len(sys.argv) < 2:
    exit(0)

def hash_chain(code):
    h = NONCE
    for i in code:
        h = pedersen_hash(h, i)
    return h

id = encode_shortstring("BightLubl was here :p          ") + int(sys.argv[1], 10)

while True:
    id += 16
    CODE = []
    hashed_id = pedersen_hash(NONCE, id)

    # code[0] == hashed_id**2
    CODE.append((hashed_id*hashed_id) % PRIME)
    # code[0] * code[1] * 4919 == 4918
    CODE.append((4918 * pow(CODE[0], -1, PRIME) * pow(4919, -1, PRIME))%PRIME)
    # code[2] * hashed_id == RET
    CODE.append((RET*pow(hashed_id, -1, PRIME))%PRIME)

    hashed_code = hash_chain(CODE) % PRIME

    # proof of work :
    a = hashed_code >> 128
    b = hashed_code - (a << 128)
    if a < 0x10000000000000000000000000000 and b < 2**128:
        break

print("id=", hex(id))
print("CODE=", [hex(i) for i in CODE].join(","))
print("a=", hex(a))
print("b=", hex(b))
```

Figure 13: Authentification

On récupère ainsi le dernier flag :

SSTIC{15fb587e4dc04bbb7abb68fc6651f593d6eb0e4fd84bbfa800c6a66043bda86a}

Fin du challenge

Avec le flag précédent on a également récupéré un puzzle que l'on reconstitue avec GIMP:

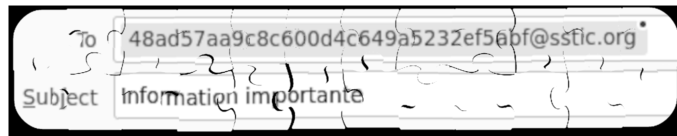

Figure 14: Adresse mail

Encore un plaisir de résoudre le challenge SSTIC cette année ! Merci aux concepteurs.