Solution concise du challenge SSTIC 2025

Pierre Bienaimé

24 mai 2025

Table des matières

1 Introduction

J'aime le challenge SSTIC ! Chaque année, j'apprends grâce à lui de nouvelles choses et je prends grand plaisir à le résoudre, même si c'est toujours un moment difficile. C'est un challenge exigeant, qui nécessite un sérieux investissement en temps et une bonne dose de motivation. On se sent souvent très nul. Mais le plaisir d'arriver au bout n'en est que plus grand !

Cette année, comme l'année dernière, ma solution sera concise, à savoir strictement une seule page par niveau. Ce format peut permettre aux curieux d'obtenir un rapide aperçu du contenu du challenge 2025. Il se focalisera sur les grandes étapes de ma résolution et contiendra forcément bon nombre d'ellipses. J'invite donc les lecteurs désireux de plus de détails techniques à consulter les solutions des autres participants.

Voici les thèmes abordés dans le challenge de cette année :

Prologue : Stéganographie dans des fichiers PDF.
Étape 1 : Cassage d'un chiffrement RSA dans GF2.
Étape 2 : Échappement d'une sandbox Lua.
Étape 3 : Exploitation d'une vielle version du moteur de rendu de Firefox.
Étape 4 : Reverse d'un algorithme ne contenant que des instructions `mov`.
Épilogue : Connexion à un service réseau complexe.

2 Prologue : Mestre du pdf

Le challenge comment avec le fichier strange_sonnet.pdf dont la première page est une image portant le texte XOR ME. Les autres pages (visibles) du PDF peuvent être ignorées. Elles renferment de la poésie en hommage aux clients lourds et au fait de réinventer la roue. Il faut donc appliquer un XOR sur la première image. Mais un XOR avec quoi ?

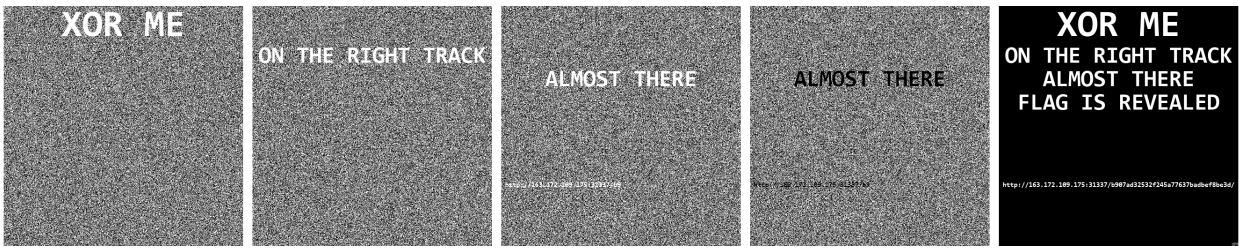
Ce prologue se révèle être un niveau de stéganographie dans le format PDF. On commence par extraire proprement l'image XOR ME. Pour ce faire, le mieux est d'éviter d'utiliser un outil qui risquerait de prendre un peu trop de libertés avec l'image, comme par exemple la convertir silencieusement en PNG, ce qui peut être gênant quand on s'attaque à de la stéganographie. Le format PDF peut se parser à la main avec un éditeur hexa. L'image XOR ME est contenue dans l'objet 8. On extrait le flux qui se trouve entre les mots clés `stream` et `endstream`, puis on le décode/décompresse (`base64.a85decode`, `zlib.decompress`). On obtient ainsi une liste de 262144 pixels (c'est à dire 512 x 512), avec un octet par pixel. Voici une commande issue d'ImageMagick permettant de visualiser ces pixels.

```
# display -size 512x512 -depth 8 gray:xorme.raw
```

On analyse ensuite les autres objets contenus dans le PDF. L'objet 36 est une chaîne de 262144 octets, c'est à dire la même taille que l'image XOR ME. Quand on la xor avec cette dernière, on voit apparaitre le texte *ON THE RIGHT TRACK*.

L'objet 39 est un deuxième fichier PDF (secret.pdf). Il est protégé par mot de passe. On extrait le hash du mot de passe et on le casse avec `john` et la base RockYou (on trouve « lobsterpumpkin »). Le PDF déchiffré contient une nouvelle image à xorer avec les autres, ce qui fait apparaître le texte *ALMOST THERE*.

L'objet 42 est un troisième fichier PDF, qui n'est autre que la RFC du format PDF. Son objet 100 est une image qui contient une incohérence entre la taille annoncée dans les métadonnées et la taille effective de son stream. Dans les données supplémentaires du stream, on trouve une nouvelle image découpée en deux morceaux. En xorant successivement toutes nos images, on fait apparaître l'URL contenant la suite du challenge : http://163.172.109.175:31337/b907ad32532f245a77637badbef8be3d/



Une fois l'URL en poche, certains participants ont foncé vers la suite du challenge sans voir le flag contenu dans les derniers octets de l'image finale.

```
SSTIC{4d80a6b32f8ff039c39f67b150b2b8d33a991b2e38a9ce96}
```

3 Étape 1 : Crypto luron

En visitant l'URL trouvée à la fin du prologue, on obtient le code nécessaire aux 4 étapes suivantes, ainsi qu'un client lourd qui servira de support à ces épreuves. Le client lourd comprend notamment un tchat permettant de discuter avec les bots de chaque niveau.

Pour le premier niveau, on discute avec Crypto Luron qui nous met au défi de déchiffrer un message arbitraire chiffré avec sa clé privée. On dispose du script de chiffrement. C'est un petit script Python d'une cinquantaine de lignes qui implémente un chiffrement RSA (jusque là, tout va bien) mais dans le corps de Galois GF(2) (aïe).

Le script implémente l'addition dans GF(2) (qui correspond à un XOR) ainsi que la multiplication (équivalent à un AND). De prime abord, difficile de savoir ce que ça implique concrètement pour la sécurité du chiffrement RSA. Mais on imagine que comme pour le cassage d'un RSA classique, la première étape est de factoriser la clé publique n.

Comme désormais il faut vivre avec le fait que des IA sont capables de réfléchir à notre place, j'ai demandé à ChatGPT de m'écrire un script capable de factoriser n dans GF(2) et je dois dire que j'ai été assez impressionné. Il m'a créé un script `sympy` qui a fonctionné du premier coup. En pratique, ce script est assez simple. Il instancie d'abord n sous la forme d'un polynôme dans GF(2), puis appelle la fonction factor_list() de `sympy`. La factorisation de n prend quelques secondes. On obtient un produit de 9 facteurs irréductibles dans GF(2).

Maintenant, il nous reste à recalculer Φ puis la clé privée d. Dans le cas classique du RSA, la clé publique n est le produit de deux nombres premiers p et q et on a $\Phi(n) = (p-1)(q-1)$. Mais dans notre cas, nous avons 9 facteurs au lieu de 2, et nous sommes dans GF(2). Qu'est-ce que ça implique pour le calcul de Φ ?

Puisqu'on ne change pas une équipe qui gagne, j'ai demandé à ChatGPT de m'écrire un script avec `sympy` capable de calculer Φ puis la clé privé d dans GF(2). Et dans un mélange de fascination et de dégoût, ça a encore une fois fonctionné du premier coup. Avec la clé privée en notre possession, on peut déchiffrer les messages de Crypto Luron et le flag du premier niveau.

Au final, cette épreuve aura été résolue en 30 minutes montre en main avec l'aide d'une IA. Heureusement, pour les niveaux suivants, ça fonctionnera beaucoup moins bien et il sera nécessaire de réellement réfléchir. Parfois, l'utilisation de l'IA sera même contre-productive quand elle se mettra à inventer des offsets erronés, des noms de fonction IdaPython fantaisistes, ou des techniques d'exploitation qui ne fonctionnent pas.

Bref, l'IA est un outil très puissant, qui a évolué de manière impressionnante en quelques années et qui change la manière d'aborder un CTF. Pour l'heure, sa principale faiblesse semble être de soutenir mordicus des solutions fausses plutôt que d'être capable de reconnaître elle-même ses propres limites.

`SSTIC{f5ab077834d560a2711413da4646bfa1f02e9b24df9c0863}`

4 Étape 4 : Movfuscated

Pourquoi l'étape 4 juste après l'étape 1 ? Cette année, la numérotation des niveaux est quelque peu déroutante. Une fois le prologue terminé, nous avons accès aux fichiers de tous les autres niveaux. Mais les bots du client lourd n'acceptent de nous parler qu'après avoir validé suffisamment de flags. L'ordre attendu de résolution des étapes est 1 → 4 → 2 → 3.

L'étape 4 est un crackme qui attend un mot de passe de 16 caractères en entrée afin de déchiffrer un fichier flag.enc fourni. Dans IDA, on constate que la vérification du mot de passe et le déchiffrement du fichier se font dans une *petite* fonction qui ne contient que des instructions `mov` (on en compte 102151). Nous sommes en présence de code obfusqué par le movfuscator (https://github.com/xoreaxeaxeax/movfuscator/). Car oui, l'instruction `mov` en x86 est Turing-complete. Le movfuscator n'est pas un outil nouveau. Il date de 2015 et a déjà été utilisé plusieurs fois lors de CTF. En lisant quelques writeups, on observe trois approches approches possibles. **1)** Un canal auxiliaire afin de ne pas avoir besoin de désobfusquer (typiquement, compter le nombre d'instructions exécutées). Dans notre cas, ça ne fonctionne pas car ce nombre est constant. **2)** L'outil demovfuscator (https://github.com/leetonidas/demovfuscator) afin d'obtenir du code plus lisible. Mais il ne fait pas de miracles et semble trop difficile à adapter à notre cas. **3)** La désobfuscation manuelle. C'est plus long, mais c'est plus fun ! Et c'est la solution pour laquelle on opte.

Une grande partie de la magie du movfuscator repose sur des lookup tables qui implémentent toutes les opérations arithmétiques. en outre, il n'est pas possible de faire des `jmp` avec des `mov`. Tout le programme doit s'exécuter en intégralité à chaque tour de n'importe quelle boucle. L'astuce est de dupliquer chaque variable et d'exécuter le code sur des données jetables lorsqu'on est en dehors de la boucle courante. L'obfuscateur a besoin d'utiliser ses propres registres. Au final, on se retrouve à analyser une VM exotique.

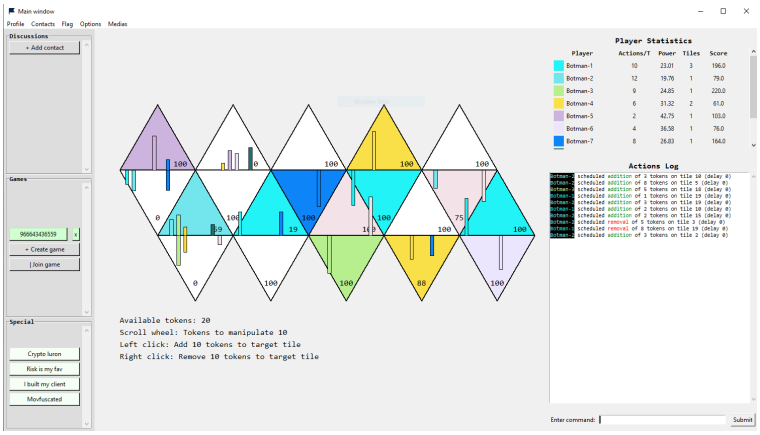
On documente toutes les lookup tables du programme. C'est suffisant pour trouver les 13 premiers caractères du mot de passe. Il s'agit d'une simple soustraction (dans un sens, puis dans l'autre) entre deux tableaux hardcodés de 13 octets. Il ne manque donc que 3 octets à bruteforcer. Mais le programme prend trop de temps à s'exécuter pour que ce soit viable, et la suite du code se complexifie car on rencontre nos premières boucles. Il devient nécessaire de désobfusquer. J'ai opté pour l'écriture d'un script IDAPython qui reconnaît des motifs parmi les suites de `mov` et qui les traduit en pseudocode. J'ai ainsi identifié 31 motifs différents, plus ou moins complexes. Par exemple, il faut 11 `mov` pour faire un xor entre deux octets, et il n'en faut pas moins de 151 pour réaliser une addition entre deux registres de 64 bits.

On obtient 2000 lignes de pseudocode et on étudie l'algorithme de déchiffrement. On patch ensuite le binaire pour insérer de vrais `jmp` à la place de certains `mov`, ce qui augmente drastiquement la vitesse d'exécution et facilite l'utilisation du débugueur. Mais c'est toujours trop lent, d'autant que l'aglorithme contient du code inutile pas si facile à supprimer proprement. Au final, il a été nécessaire de recoder tout l'algorithme en Python (une centaine de lignes), ce qui fut assez fastidieux, mais a rendu le bruteforce possible. On trouve ainsi le bon mot de passe `Reegh3meiXuvu7re` et le flag dans la foulée.

`SSTIC{21c66b2c691438c8a99b33e28c1cd5f42009468d3c68d701}`

5 Étape 2 : Risk lover

Le but du niveau est de s'échapper d'une sandbox Lua implémentée en Python. Via le client lourd, on démarre une partie d'un jeu nommé `bridge`, dont les règles semblent cryptiques. Il y a des triangles. Des barres. Des chiffres. Et des bots qui envoient des actions.



L'interface nous permet d'uploader un fichier Lua afin d'automatiser nos actions. On audite donc les fonctions Lua autorisées pour trouver un moyen d'exécuter du code. Notre recherche est biaisée par l'erratum publié sur la page du challenge, laissant entendre qu'il y a une erreur dans `bridge.py` rendant l'échappement de sandbox beaucoup plus simple que prévu. On part donc en quête de cette erreur. Et on la trouve assez rapidement. La sandbox autorise l'appel à une fonction load_state() qui permet de charger du code Lua arbitraire en contournant tout le filtrage mis en place dans la sandbox. En trois lignes de Lua, on peut donc exécuter une commande bash sur le serveur de jeu.

```
local func = load_state("return _G")
local g = func()
g.os.execute("ls /")
```

Pour lire la sortie de notre commande, on passe par le protocole du jeu. On écrit dans un fichier temporaire, on le lit avec du code Lua et on le convertit en entiers de 32 bits qu'on exfiltre dans le champ Delay du format Json attendu par le jeu. Ainsi, les entiers sont affichés dans la fenêtre de logs du jeu. Après quelques uploads, on trouve le flag dans le fichier `/thiswillforceyoutorce/dontguessthis/onemore/hmmmm/flag.txt`.

```
SSTIC{b871c80ae6baa5fb806f7241109e9d399f8641f2a63c7f69}
```

On récupère ensuite le fichier `bridge_expected.py`, qui est identique à `bridge.py` à l'exception de la fonction load_state qui a été supprimée. Voici quelques pistes pour s'échapper de cette vraie sandbox. La version de Lua utilisée est ancienne (5.2.4, sortie en 2015) et vulnérable à des chargements de bytecode Lua. En outre, la fonction setlocale() est marquée UNSAFE pour une sandbox dans la documentation (http://lua-users.org/wiki/SandBoxes). Mais pour ma part, j'ai tracé tout droit vers la suite du challenge et je n'ai pas pris le temps d'y revenir. L'exploitation de `bridge_expected.py` est donc laissée en exercice au lecteur :)

6 Étape 3 : Gecko party

C'est non. Voici un niveau pour lequel j'ai eu du mal à trouver de la motivation. Via le tchat du client lourd, on peut demander à un bot de venir visiter une URL arbitraire. On sait que son navigateur utilise geckofx45.64 en version 45.0.34 sur Windows Server 2019. Geckofx est un wrapper permettant d'utiliser le moteur de rendu de Firefox dans l'écosystème .NET. Le but du challenge est de se choisir une 1-day ciblant cette vieille version du moteur de rendu de Firefox. L'année dernière, je n'avais déjà pas aimé l'exploitation Chrome. Les concepteurs avaient créé un plugin vulnérable. La vulnérabilité était facile à trouver. Toute la difficulté, c'était savoir quoi faire de la primitive RW obtenue. La difficulté, c'était Chrome. Ce que j'aime dans un challenge, c'est me confronter à des énigmes atypiques produites par l'esprit tortueux de ses concepteurs. Exploiter un navigateur, c'est difficile, mais ce n'est pas le type de difficulté que je recherche. Et cette année, voilà qu'on nous demande d'exploiter Firefox. Et on n'a même pas droit à la partie amusante de devoir trouver une vulnérabilité dans du code écrit pour l'occasion, puisqu'il n'y a pas de code.

On commence par se fabriquer un environnement identique à celui du client (VM Windows Server 2019, Visual Studio, geckofx45). Puis on crée un projet .NET minimaliste qui utilise gecko pour visiter une URL. On vérifie que notre User-Agent est identique à celui du client. Ensuite, il faut se choisir une CVE. Et le choix n'est pas si facile. On veut la vulnérabilité la plus simple et la plus stable possible. Devoir porter l'exploit d'un use-after-free obscur provoqué par une race condition, ce n'est pas l'idée du siècle. D'autant qu'on est en remote et qu'on ne dispose pas du code exact du client web. Rien ne nous garantit qu'il se comportera exactement de la même façon que notre client.

Plusieurs CVE sont prometteuses. Mais les POC de 2016 et 2017 sont tous sur des versions de Firefox 32 bits. Les porter en 64 bits risque d'être un parcours du combattant. On opte finalement pour la CVE-2019-9791. Son POC offre une primitive RW et AddrOf relativement stable (sauf quand on fait trop d'actions et que le gc passe au mauvais moment). Le problème est que le POC est conçu pour Firefox 60 et que nous sommes en version 45.

Le debug a été fastidieux et désagréable car je n'ai jamais réussi à trouver les symboles de debug de la lib xul.dll embarquée dans geckofx45. La transformation de la primitive RW en exécution de code arbitraire fut âpre, car les différentes techniques lues çà et là n'ont pas fonctionné. Après une infinité de tâtonnements, j'ai fini par réussir à contrôler `rip` en créant un objet XMLHttpRequest et en écrasant un pointeur :

```
let xhr = new XMLHttpRequest();
let xhr_open = memory.addrOf(xhr.open);
let target = Int64.add(xhr_open, 0x28);
memory.write8(target, shellcode_addr);
xhr.open();
```

Le shellcode est un reverse-shell généré avec metasploit et encodé en flottants, qu'on spray partout avec la methode ASM JS afin de maximiser nos chances de trouver son adresse. Chaque étape fut douloureuse, mais on finit par obtenir un exploit fonctionnel.

```
SSTIC{58e9ab359732a4a5408661470bb3bf34e9b8362c639f5b83}
```

7 Épilogue

Le code source du niveau bonus se récupère avec le reverse shell du niveau précédent. C'est un service réseau nommé `FlagProvider` qui est assez difficile à prendre en main. Il y a beaucoup de code (67 fichiers Python), plusieurs frameworks utilisés, de l'authentification par certificat, et autres joyeusetés. Le but est de modifier quelques lignes pour obtenir un client capable de se connecter à ce service avec le privilège challenge_finisher. On appelle ensuite la fonction `GetFlagsOrder()`. L'adresse email finale se trouve en calculant un super hash à partir du sha512 de tous les flags (dans l'ordre `mestre du pdf` → `crypto luron` → `risk lover` → `gecko party` → `movfuscated`) et en rajoutant le suffixe _you_deserve_rest@sstic.org.

`76a304cf910e6c6e4051ca7c7c05f8d51fc3e60c4f180077630994484fc9c654_you_deserve_rest@sstic.org`

8 Conclusion

Ainsi se conclut le 13ème challenge SSTIC que je parviens à terminer. Rédiger cette solution fut pour moi l'occasion de me replonger avec une certaine nostalgie dans quelques unes de mes anciennes solutions. Au fil des années, elles ont évolué et j'ai pris conscience qu'on pouvait les ranger dans 4 cycles :

Cycle initiatique : 2011, 2013, 2014, 2016 et 2017, avec des solutions classiques faisant entre 30 et 50 pages, marquant mes premiers pas en cybersécurité.

Cycle poétique : 2018 et 2019, avec des solutions courtes et l'envie inexpliquée d'écrire de la poésie.

Cycle pédagogique : 2020, 2021, 2022 et 2023 avec des solutions détaillées, qui s'étirent en longueur, et ont permis de remporter deux fois le classement qualité.

Cycle synthétique : 2024 et 2025 avec des solutions concices d'une page par niveau.

Merci aux concepteurs de cette année, qui ont su relever la difficile tâche de concevoir un challenge SSTIC, qui plus est en étant peu nombreux et avec un calendrier serré. J'ai pris beaucoup de plaisir à mettre les mains dans le movfuscator. J'ai également apprécié le prologue, car j'aime bien la stéganographie. Et si j'avais été un peu moins paresseux, j'aurai bien voulu profiter de cette épreuve pour coder mon propre parseur de PDF orienté stegano. Cependant, la stéganographie étant assez impopulaire, le choix de mettre cette épreuve dès le prologue était une stratégie risquée et semble avoir dégoûté prématurément plusieurs participants, ce qui est regrettable.

Je n'ai pas vraiment d'avis sur le niveau de cryptographie (résolu trop vite grâce à l'IA) et sur la sandbox Lua (un peu gâchée par l'erreur laissée dans le code. Mais ce sont des choses qui arrivent). Concernant l'exploitation Firefox, je l'ai clairement fait en traînant les pieds.

L'édition 2025 du challenge SSTIC avait en tout cas le bon goût d'être un peu moins longue que certaines éditions précédentes. Mais il reste une question qui me taraude. Pourquoi un client lourd ? Pourquoi avoir dépensé autant d'énergie à le coder, mais ne pas nous avoir forcé à le décortiquer ? Je pensais que le dernier niveau serait le client lourd. Et je crois que j'aurai préféré ça à Firefox :)

Merci pour tout et à l'année prochaine !