Solution concise du challenge SSTIC 2024

Pierre Bienaimé

21 mai 2024

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 !

S'il m'est parfois arrivé de rédiger des solutions détaillées (105 pages en 2022...), ma solution de cette année sera concise pour deux raisons :

— J'ai terminé ce challenge le jour de sa clôture, ce qui ne m'a pas vraiment laissé le temps de nettoyer mon code ou de me perdre en explications.
— Étant membre du comité de programme de la conférence SSTIC cette année, je suis hors concours et je ne peux de pas prétendre au classement qualité.
— J'ai menti, il y a une troisième raison. Je préfère une solution concise plutôt que pas de solution du tout. Puisse-t-elle permettre à quelques lecteurs curieux mais pressés de se faire une vague idée du contenu du challenge de cette année.

Ainsi, cette solution ne fait que 9 pages au total, dont strictement une seule page par niveau, et ne contient pas de code. Les niveaux 2, 3, 4 et 5 sont particulièrement difficiles et ont demandés des jours, voire des semaines de travail qu'il est utopique de pouvoir condenser en une page. Ne soyez donc pas étonnés d'y trouver *quelques* ellipses.

Voici le scénario de cette édition 2024 :

> The year is 2022. Gunshots shattered the night. A witness caught a glimpse of a vehicle at the scene – a Tesla, they claim ? The investigators on the case firmly believe this incident is affiliated with a clandestine criminal group that has eluded them for years... You are given the police reports and a backup of the car's logs. Your technical expertise is needed to move the investigation forward !

Le but du challenge est d'enquêter sur une organisation mafieuse afin de trouver une adresse email. Mais la route est semée d'embûches. Chaque niveau est découvert séquentiellement à travers des pages web particulièrement soignées et roleplay. Elles présentent l'avancée de notre enquête et le contexte de nos missions successives. Au programme : crypto, exploit, crypto, expoit et encore exploit.

Prologue : Explorer la base de données d'une voiture Tesla.

Chapitre 1 : Casser un protocole de communication chiffré.

Chapitre 2 : Exploiter la heap d'un jeu vidéo sous Linux.

Chapitre 3 : Casser un HMAC SHA256 à l'aide d'une injection de faute.

Chapitre 4 : Exploiter une version modifiée du navigateur Chrome sous Windows.

Chapitre 5 : Exploiter un driver Windows pour élever ses privilèges.

Bonus : Trouver l'adresse email cachée dans un enregistrement sonore.

2 Prologue

Cette année, le premier niveau du challenge a été baptisé *Prologue* par les concepteurs. Dans le cadre d'une enquête de police fictive, on nous demande d'analyser les logs d'une voiture Tesla dont le propriétaire est impliqué dans un homicide. Pour cela, on dispose d'un fichier *teslamate.bck* de 171Mo qui est un dump au format texte d'une base PostgreSQL.

La base SQL contient une grande quantité de positions GPS, une liste d'adresses postales, des niveaux de charge de batterie, des historiques de trajets, etc. Ma première idée a été d'extraire les positions GPS et de les convertir au format gpx pour les afficher sur une carte. Le conducteur a-t-il voyagé à travers le monde pour dessiner un flag géant avec ses positions GPS ? Non.

En examinant le contenu de la base à la main, on trouve des chaînes de caractères suspectes, notamment dans la table *addresses*. En effet, au milieu d'adresses légitimes (bien qu'aux noms parfois farfelus) situées au Pays Bas, en Allemagne et en Belgique, on trouve une adresse nommée **:8080**. Elle nous sera bientôt utile. Ce qui est moins chouette, c'est qu'on trouve également des données suspectes mais qui s'avèrent être de fausses pistes, comme par exemple une adresse qui contient un numéro de rue beaucoup trop élevé : 222508376036.

Dans la table *geofences*, on trouve les noms **Domicile_http_163** et **ServerRoom_99**. On commence à comprendre qu'il va falloir recoller les morceaux d'une URL HTTP. Un rapport de police nous explique justement qu'il serait utile de retrouver les endroits les plus visités par la Tesla. Pour cela, j'ai directement travaillé sur le dump texte que j'ai parsé grossièrement avec des grep et du python. J'ai ensuite recoupé les identifiants de la table *drives* avec ceux des tables *addresses* et *geofences*.

S'il apparaît évident que l'URL va commencer par **http ://163**, se terminer par **:8080** et contenir un 99 quelque part, trouver les deux morceaux manquants pose problème. Dans les adresses les plus visitées, on en trouve une dont le numéro de rue est 172 et une autre 233, mais ces numéros font partie d'adresses postales légitimes et il n'est pas si logique de vouloir les utiliser comme des morceaux d'adresses IP.

On se retrouve donc à scanner le port 8080 de machines sur Internet en testant toutes les combinaisons des nombres 163, 99, 172 et 233, ce qui n'est pas très satisfaisant. Finalement, on trouve le flag du prologue à l'adresse `http://163.172.99.233:8080`, un flag qui a quand même un petit arrière goût de guessing.

Plot twist : il existe en réalité un logiciel qui s'appelle *TeslaMate* et qui propose une GUI pour gérer les logs d'une voiture Tesla. En chargeant le dump *teslamate.bck* dans ce logiciel, il était possible en quelques clics d'afficher les endroits les plus visités par la voiture et d'éliminer le guessing. Mais encore fallait-il penser à chercher l'existence de ce logiciel. Certes, le nom du logiciel était *littéralement* dans le nom du fichier à analyser. Mais en pratique, quand on a entre les mains des données dans un format standard (ici une base PostgreSQL au format texte), il n'est pas forcément naturel de partir en quête d'outils plus évolués qu'un simple éditeur de texte ou qu'un client postgreSQL.

`SSTIC{0a3ef4d4bb265ca2f27dd557be06e47e84aaacabdc501daade6ee97b8c0e8f3c}`

3 Chapitre 1 : Intrigue in the Interstice

Grâce à notre travail sur les données du Prologue, les enquêteurs ont remarqué que le propriétaire de la Tesla se rendait régulièrement dans une salle serveur soupçonnée d'héberger des données d'une mystérieuse organisation mafieuse. Elle héberge notamment le serveur de licence d'un jeu vidéo auquel jouent des membres de l'organisation. Les enquêteurs se sont rendus sur place et ont installé un boîtier permettant de faire du *Man in the Middle* entre le serveur de licence et ses clients. Notre mission est d'utiliser ce boîtier pour analyser le trafic réseau, comprendre le protocole et récupérer une clé de licence valide.

En envoyant des requêtes invalides au serveur de licence, on obtient une grande variété de messages d'erreurs qui nous permettent de comprendre le format du protocole. Le message d'erreur le plus intéressant est `Invalid ISO 7816-4`. Cette erreur nous permet de mener une attaque par oracle de padding et de *décrypter*[1] octet par octet, bloc par bloc, tous les messages envoyés par les clients.

Ensuite, on cherche à *décrypter* les messages envoyés par le serveur. Dans ce sens, l'attaque par oracle de padding est toujours possible mais extrêmement lente car on ne peut pas spammer les clients, on doit attendre qu'ils contactent le serveur. Heureusement, on peut utiliser une autre astuce : si les requêtes contiennent du JSON malformé, le contenu fautif est affiché en clair. On peut donc utiliser les clients pour qu'ils déchiffrent pour nous les messages du serveur.

Une fois une trace complète déchiffrée dans les deux sens, c'est tout de suite plus facile de voir comment récupérer une clé de licence. Certains clients utilisent la commande 07 pour demander au serveur de leur envoyer leur mot de passe et leur clé de licence. Mais ces clients ne sont pas *activés* et n'ont donc pas de licence. A l'inverse, on voit d'autres clients qui utilisent des commandes privilégiées. Le plan est donc de voler l'ID de session d'un client activé et d'envoyer une commande 07 en son nom.

L'attaque par oracle de padding est tellement puissante qu'elle permet même de *crypter* des messages arbitraires sans jamais connaître ni la clé ni l'algorithme de chiffrement. Mais en pratique, pas besoin de forger des messages complets puisqu'il suffit de voler l'ID de session d'un client activé. On se met en MITM, on attend de voir passer un message 07 (reconnaissable par sa taille) et un message 09 (lui aussi facile a reconnaître, et envoyé uniquement par des clients activés). On *décrypte* le message 09 pour trouver l'ID de session et on modifie l'IV du message 07 pour altérer son ID de session par effet de bord.

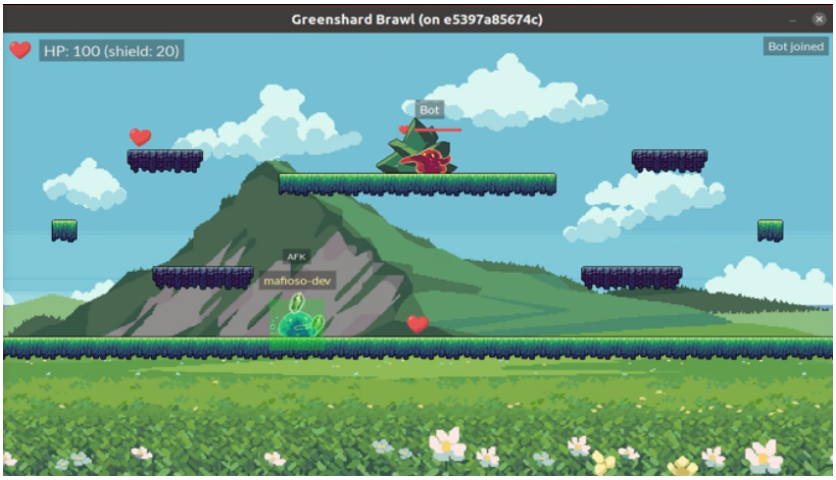
J'ai beaucoup aimé ce niveau de cryptographie qui illustre parfaitement les conséquences désastreuses d'un simple message d'erreur de padding.

`SSTIC{f4746e9051d51bcf26c77f02ccb5790375f873a2a2560478dd41d309bac9ab2d}`

1. Oui, je sais, ça fait bizarre, mais c'est un des seuls cas où l'on a le droit d'utiliser ce mot donc j'en profite. En effet, l'attaque permet de déchiffrer tous les messages mais sans jamais connaître ni la clé de chiffrement, ni même l'algorithme de chiffrement.

4 Chapitre 2 : The Green Shard Brawl

Bon, on arrive au moment où ça devient insensé de vouloir faire tenir un niveau en une seule page. Surtout si je mets une image... ou que je gaspille des mots pour écrire cette phrase. Grâce à notre clé de licence, on va pouvoir jouer au jeu vidéo préféré de l'organisation, *Green Shard Brawl*, un jeu de baston où l'on incarne des genres de blobs qui se tapent dessus. Un joueur de la mafia est connecté au jeu en permanence. Le but est de rejoindre sa partie et d'exploiter une vulnérabilité dans le code de son client, afin d'obtenir un reverse shell sur sa machine. On dispose d'images Docker pour faire tourner le client et le serveur de jeu.



Le client est en C (sous linux) et le serveur en python. On va coder un Bot (en python lui aussi) capable de tricher. Il peut se téléporter, donner des coups aux dégâts quasi illimités, etc. Ce bot va effectuer une suite d'actions bien précises afin de d'exploiter des vulnérabilités dans le client de notre victime.

Une première vulnérabilité nous permet d'obtenir un leak de l'adresse de la stack de la victime et de sa libc. Pour cela, il faut se connecter au serveur avec un joueur nommé **%p** puis **%s%p** et donner un coup à notre adversaire. Ensuite, on frappe l'adversaire pour le forcer à prendre un bouclier, puis on le pousse pour qu'il changer de zone. Le bouclier est alors libéré mais on conserve un pointeur dessus, provoquant un Use After Free.

L'affichage de la valeur du bouclier nous offre un leak d'adresse de Heap. En frappant très fort l'adversaire, on va pouvoir faire baisser la vie du bouclier... et donc décrémenter la mémoire afin d'écrire une valeur arbitraire dans la Heap à l'endroit du chunk libéré. Les derniers morceaux de l'exploitation sont les messages de chat. Une feature utile qui nous permet d'allouer des chunks de taille arbitraire et de contrôler leur contenu.

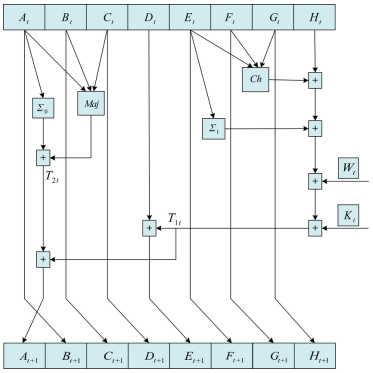
La difficulté est que nous sommes en présence d'une libc récente et que les techniques que je connais ne fonctionnent plus (il n'y a même plus de `malloc_hook`!). C'est une bonne occasion pour se mettre à jour en lisant tous les exemples de https://github.com/shellphish/how2heap. J'ai beaucoup apprécie ce niveau et les efforts apportés pour créer ce jeu vidéo. Et je suis content d'avoir pu démystifier le fonctionnement du tcache!

`SSTIC{ae50935e902c926aa72cd5c526d128c947561f1ec3a0f6df4f753ad4e5e35a90}`

5 Chapitre 3 : The Authenticator in SHAdow

Grâce à notre reverse shell sur le code d'un développeur de l'organisation, on obtient le code du serveur de Chat sécurisé de la mafia. Il est en trois parties : un client *shardy*, un serveur *mafiachat* et un HSM, un composant hardware RISCV qui sert à stocker des secrets cryptographiques et à authentifier les messages du chat. Le HSM contient une backdoor insérée par la mafia. Le but est d'utiliser cette backdoor pour envoyer un message authentifié au BOSS avec l'identité du DEV.

Très bien. Mais quand on voit la *backdoor* en question, on déchante un peu ! Elle nous permet juste d'ajouter 32bits de random à un tour pair de notre choix de la fonction de compression de SHA256. Et on ne peut même pas connaître la valeur du random qui a été ajouté ! Le SHA256 est utilisé pour calculer les HMAC qui servent à authentifier les messages du chat. En quoi cette backdoor va-t-elle nous aider à usurper l'identité de DEV ? Voici le schéma qui représente un tour de la fonction de compression de SHA256 et qui a hanté plusieurs de mes nuits :



Après un temps infini à poser des équations sans parvenir à les résoudre, tout est devenu plus clair en trouvant la bonne terminologie. Ajouter du random, ça revient à injecter des fautes. La publication de 2014 *Algebraic Fault Attack on the SHA-256 Compression* explique comment tirer profit de ces injections de faute à l'aide d'un solveur (z3 pour ma part).

Dans la souffrance, j'ai reproduit chaque étape de la publication afin de *déhasher* un bloc de SHA256. On peut *déhasher* un bloc mais pas deux, on ne peut donc pas retrouver la clé du HMAC. A la place, on applique l'attaque *Hash Length Extension* au cas particulier d'un HMAC. A l'aide des fautes, on fait fuiter l'état interne du HMAC et on l'utilise pour le prolonger et forger un message ayant un HMAC valide alors qu'on ne connaît pas la clé.

Le prolongement de HMAC fonctionne car le serveur ne parse pas les messages de manière très sécurisée : quand un token est dupliqué, il ne conserve que la dernière occurrence, ce qui nous permet d'usurper l'identité de DEV. J'ai beaucoup apprécié la leçon de ce niveau : un minuscule changement anodin peut suffire pour que la sécurité d'un algorithme cryptographique s'effondre ! Par contre, qu'est ce que j'ai souffert avec z3 ! Dès que je soufflais un peu trop fort sur mes équations, j'arrivais souvent dans cet état si frustrant : z3 va-t-il me trouver une solution dans une minute... ou cinq... ou jamais ?

```
SSTIC{b3e2c71f3e67d5a8d5855adb30842549ad2bc6e3e7bdbec141d61ee83e2f8f46}
```

6 Chapitre 4 : Hic Jacet Chromium

Grâce au cassage du HMAC du niveau 3, des agents ont pu faire de l'ingénierie sociale auprès du BOSS de l'organisation. Ce dernier nous a confié la mission de tester un navigateur *sécurisé* développé par son cousin. Il s'agit d'une version de Chrome pour Windows, modifiée et compilée sans sandbox. Le but est de trouver une vulnérabilité dans ce navigateur pour obtenir un reverse shell lorsque le BOSS naviguera sur notre page HTML.

Trouver une vulnérabilité, c'est la partie facile, car il y a peu de code ajouté à Chrome : deux petits fichiers C++ qui implémentent les objets *Authenticator* et *AuthenticationData* utilisables en Javascript. Il n'y a pas de vérification sur la valeur de l'offset *endOfAnthenticode*, ce qui permet d'obtenir directement une lecture arbitraire relative et une écriture temporaire. Mettre un offset négatif permet de lire des adresses de Heap et donc d'obtenir une lecture mémoire absolue.

L'écriture temporaire peut être transformée en écriture définitive d'un octet en se creusant un peu la tête. Pour cela il faut choisir 0xFFFFFFFFFFFFFFDF comme *endOfAnthenticode* et écraser deux fois d'affilé 7 octets de la *saved_value* avec comme *delimiter* notre adresse destination. Seule limitation, on ne peut écrire qu'à des adresses qui sont avant nous dans la heap car on ne contrôle que 7 octets de l'adresse destination (l'octet de poids fort valant toujours 0xFF).

La partie qui m'a découragé (outre le fait de devoir tout coder en Javascript) c'est plutôt : « *Bon, vous avez une primitive RW. Maintenant voici tout Chrome. Bon courage !* ». Car à ce stade du challenge, je manque à la fois de temps et de motivation pour découvrir un sujet aussi vaste que Chrome. D'autant que je n'ai jamais réussi à me préparer un environnement de travail confortable. Je ne suis pas parvenu à charger le PDB de chrome.dll dans IDA (il faut dire que ce PDB monstrueux fait 4.5 Go). Dans windbg ça « fonctionne » mais c'est d'une lenteur atroce.

La technique d'exploitation qui a fonctionné, c'est de déclarer des fonctions Wasm afin que des pages RWX soient allouées pour stocker leur code JIT. On peut retrouver nos pages RXW en lisant `chrome!v8::internal::ThreadIsolation::trusted_data_.jit_pages` grâce à notre lecture arbitraire et à la connaissance de l'adresse du code et des globales de Chrome, retrouvées via des vtables présentes dans la heap. On cherche une page qui est avant nous en mémoire, puis on écrit dedans le shellcode d'un reverse shell.

Ce niveau a été interrompu par deux semaines de vacances. À mon retour, j'ai vraiment failli ne pas reprendre. Au final il se sera écoulé un mois entre le début et la fin de ce chapitre. C'est long. Par contre, mention spéciale à ChatGPT qui m'a bien aidé à écrire le code Javascript qui utilise les objets C++ ajoutés à Chrome.

```
SSTIC{eb92ddb9cbc5fd3114f117a30796fd17340fa6cda03e08f158c6e7572252d31a}
```

7 Chapitre 5 : Whispers in the Shadows

Un reverse shell sur la machine Windows 10 du BOSS de l'organisation, c'est n'est pas suffisant. Pour accéder à ses fichiers les plus confidentiels, il faut une élévation de privilège. Le dernier niveau de ce challenge SSTIC 2024 est donc de l'exploitation Kernel Windows.

Pour cela, il faut trouver une vulnérabilité dans le driver netshdw.sys, utilisé pour faire du chiffrement *multi-couche*. Quand on regarde ce driver dans IDA, on voit effectivement plusieurs algorithmes de crypto imbriqués, mais ce n'est pas là que se trouve la vulnérabilité. Pour communiquer avec le driver, on va envoyer des IOCTLs à un device. Ces IOCTLs permettent de créer des *ports*, sur lesquels on va pouvoir envoyer et recevoir des données. Au moment de la création, on peut choisir si le port sera en reception ou en émission, mais pas les deux.

Pas les deux ? En fait si. Il y a une vulnérabilité TOCTOU qui permet de modifier les flags du port entre le moment où ils sont vérifiés et celui où ils sont utilisés. Il est possible de faire autant d'essais que l'on veut jusqu'à temps de parvenir à créer un port hybride. L'existence de ce port provoque une confusion de type. En envoyant un message avec un délai puis un message immédiat (et vice-versa), on obtient une lecture et une écriture relative dans la Non-Paged Pool.

Pour savoir quoi faire de notre primitive, une bonne source d'inspiration se trouve dans les nombreux writeup d'exploitations du driver HEVD (HackSys Extreme Vulnerable Driver), un driver truffé de vulnérabilités, spécialement développé pour que les chercheurs en cybersécurité s'entraînent à l'exploitation Kernel Windows.

Une technique qui revient souvent est l'utilisation de NamedPipes, qui peuvent être crées facilement depuis le userland. On spray des NamedPipes et on écrit des données dedans, afin que des structures DATA_QUEUE_ENTRY se retrouvent en grande quantité dans la Non-Paged Pool. Ensuite, on détruit quelques pipes de manière à créer des trous dans lesquels nos objets *Ports* vont pouvoir se loger. De cette façon, la lecture et écriture relative permet de lire et d'écrire dans la DATA_QUEUE_ENTRY d'un NamedPipe.

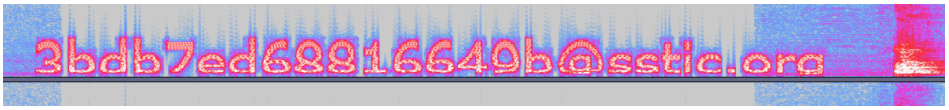
Pour obtenir un lecture mémoire absolue, on altère une DATA_QUEUE_ENTRY pour la faire pointer en userland (car il n'y a pas SMAP) vers une fausse DATA_QUEUE_ENTRY possédant une fausse structure IRP dont le SystemBuffer pointe vers l'adresse qu'on veut lire. Rien que ça. Pour obtenir une écriture mémoire absolue, c'est un peu plus compliqué car il faut d'abord leaker le contenu d'une structure IRP légitime. On le fait grâce à notre nouvelle lecture arbitraire et à un appel à `NtFsControlFile` sur un de nos pipes, ce qui lui ajoute un IRP. En le lisant, on trouve aussi un pointeur vers une structure EPROCESS, qui permet de trouver le token de notre process courant et celui du process system (pid 4).

Pour finir, on copie le token system à la place de celui de notre process, on lance un cmd.exe avec les privilèges maximums et c'est gagné !

```
SSTIC{4a0eb806f92bdd6d5be4dc2b5dc5759096a22b186f67610120e254e03c3e1429}
```

8 Bonus

Traditionnellement, il reste toujours une dernière étape à la fin du challenge pour trouver l'adresse email de validation. Une fois le shell privilégié obtenu, on trouve un dossier *confidential* sur la machine du BOSS. Il contient un dessin de chien et un PDF. Pas de stégano ? Rien dans le PDF ? Ah ben non. En cherchant mieux, on trouve dans un dossier *personal* un mp3 sur lequel on entend des aboiements et des sons *bizarres*. Après une analyse avec Audacity et quelques tests (changement de vitesse, lecture à l'envers, etc) il s'avère que l'adresse email finale est cachée dans le spectrogramme de ces sons.



9 Conclusion

Je dis souvent que le challenge SSTIC n'est pas un sprint, c'est un marathon [2]. Mais en l'occurrence, cette année c'était plutôt un ultra-trail. Et si j'ai d'emblée choisi de le faire en petites foulées, j'ai quand même dû sprinter pour terminer dans les temps et sur les rotules, le jour de la clôture du challenge, presque deux mois après son début.

C'est le 12ème challenge SSTIC que je parviens à résoudre. Chaque année, je me demande si ce sera le dernier ou pas. En 2022 j'avais déjà mis plus d'un mois et clairement, c'était déjà *trop*. Donc quel bilan tirer de cette édition 2024 ?

J'ai beaucoup aimé le scénario, l'enchaînement logique entre les chapitres et la façon très soignée dont chaque épreuve était introduite. J'ai apprécié la fourniture des nombreuses images Docker pour la mise au point de nos exploits en local. J'ai beaucoup apprécié les chapitres 1, 2 et 3. Et pour finir, grâce au challenge de cette année j'ai découvert le si pratique serveo.net ! Comment je faisais avant de connaître ce service ?

Ce que j'ai moins apprécié, c'est que c'était trop long ! Peut-être que je deviens vieux. Mais j'ai de plus en plus de mal à concilier ce loisir qu'est le challenge SSTIC avec ma vie personnelle et professionnelle. A vrai dire, si le challenge s'était terminé à la fin du chapitre 3, j'aurai trouvé ça parfait. Le prologue part d'une bonne idée, mais à mon sens l'occasion est un peu manquée à cause du guessing qui a frustré du monde. Pour les chapitres 4 (et 5 dans une moindre mesure), je pense que j'aurai pu les apprécier davantage dans d'autres conditions. Car dans l'absolu, je ne suis pas fermé à l'idée de mettre les mains dans le code de Chrome à l'occasion d'un challenge. Mais après avoir déjà dépensé trop de temps et d'énergie dans les trois premiers chapitres, je l'ai plutôt vécu comme un punition.

Je remercie mes collègues et ex-collègues, et en particulier mes partenaires de galère de challenge SSTIC. Car bien que ce challenge soit individuel, si je n'avais personne pour m'épancher sur les difficultés rencontrées, je sais que je n'irai pas souvent au bout. Et pour finir, un grand merci à l'équipe de Thalium pour l'immense travail de conception de ce challenge. Merci pour tout et (peut-être) à l'année prochaine !

2. Sauf pour Robert Xiao, notre Champion Olympique de challenges