Solution du challenge SSTIC 2020

Pierre Bienaimé

29 avril 2020

Table des matières

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 !

Voici la consigne de cette édition 2020.

```
Il y a quelque mois, un pays voisin, la Bièrique, a mis en production un
logiciel de messagerie instantanée baptisé « Kazh-Boneg ». Ce logiciel met en
œuvre des algorithmes de chiffrement à l'état de l'art afin que les serveurs
d'infrastructure ne puissent pas voir le contenu des messages.

Cela a attiré divers groupes d'opposition, dont le Sivi-Ha-Kerez, qui promeut
le remplacement de la boisson nationale par le sirop de fruits rouges.

La police de la Bièrique soupçonne ce groupe de préparer un coup d'État, et
cherche ainsi à écouter certaines discussions de Kazh-Boneg.

L'interrogation de quelques individus a permis de mettre la main sur
l'ordinateur d'un membre du groupe. Avec celui-ci, il devrait être possible de
déchiffrer les messages et de remonter à l'identité du cerveau du Sivi-Ha-Kerez.

Toutefois la police est en sous-effectif et l'urgence de cette mission la force
à demander à l'aide à ses alliés.

Êtes-vous en mesure de l'aider ?
```

Le but du jeu est d'analyser un logiciel de messagerie instantanée baptisé **Kazh-Boneg** et utilisé par des activistes soupçonnés de préparer un coup d'état. Le but ultime de ce groupe d'opposition, nommé **Sivi-Ha-Kerez**, étant de remplacer la boisson nationale (la bière) par du sirop de fruits rouges.

Ces noms à la consonance bretonne nous incitent à ouvrir un dictionnaire Breton-Français afin de vérifier cette impression. On y apprend que Kazh-Boneg signifie littéralement Chat-Code, tandis que Sivi-Ha-Kerez se traduit par Fraises-et-Cerises.

1 Niveau 1

1.1 Retrouver la mémoire

Le fichier de départ du challenge est une capture mémoire de la machine d'un membre du Sivi-Ha-Kerez. La capture a été réalisée avec LiMe [1], un module kernel pour Linux dédié à l'acquisition de mémoire volatile.

Le fichier `memory` est un flux compressé avec zlib. Une fois décompressé vers un fichier qu'on appellera `mem`, nous sommes en présence de 512Mo de mémoire brute. Avant d'utiliser des outils de forensics plus évolués, on se fait une première idée du contenu de la capture avec `strings`, `hexdump` et `grep`.

1. https://github.com/504ensicsLabs/LiME

```
$ strings mem | grep -i sstic
$ strings mem | grep -i virtualbox
[...]
Apr  2 12:56:32 kazbng-backup kernel: [    1.444741] usb 2-1: Manufacturer: VirtualBox
[...]
```

Tout d'abord — ça valait le coup de vérifier — on ne trouve pas de flag sous la forme SSTIC{...}. On apprend par contre que la machine tourne sous VirtualBox. On tombe alors sur certains messages de log qui indiquent que la machine se nomme kazbng-backup, nom qui fait référence à Kazh-Boneg, le logiciel de messagerie instantanée que nous devons analyser.

Une recherche sur ce nom de machine permet de révéler des données intéressantes. On reconnaît ci-dessous un extrait du fichier `/var/log/auth.log` qui indique que l'utilisateur bakeup s'est connecté sur la machine via `ssh` et qu'il a monté un disque externe. On trouve même des traces de commandes bash qu'il a tapé. Enfin, on trouve un équivalent de la sortie de `uname -a`.

```
$ strings mem | grep -i kazbng-backup
[...]
Apr  2 12:56:43 kazbng-backup sshd[493]: pam_unix(sshd:session): session opened for user bakeup by (uid=0)
Apr  2 12:56:43 kazbng-backup systemd-logind[387]: New session 1 of user bakeup.
Apr  2 12:57:11 kazbng-backup sudo:   bakeup : TTY=pts/0 ; PWD=/home/bakeup ; USER=root ; COMMAND=
/usr/bin/mount /dev/disk/by-label/external /mnt/external
Apr  2 12:57:12 kazbng-backup sudo:   bakeup : TTY=pts/0 ; PWD=/home/bakeup ; USER=root ; COMMAND=
/usr/bin/umount /mnt/external
[...]
Linux kazbng-backup 4.19.0-6-amd64 #1 SMP Debian 4.19.67-2+deb10u2 (2019-11-11) x86_64
[...]
bakeup@kazbng-backup:~$ export SSH_AUTH_SOCK="${XDG_RUNTIME_DIR}/gnupg/S.gpg-agent.ssh"
bakeup@kazbng-backup:~$ ssh synapse-node
bakeup@kazbng-backup:~$ ./backup.sh &
[...]
```

Pour finir, on trouve un message de log du `gpg-agent` qui nous apprend l'identité du concepteur de ce premier niveau.

```
Please unlock the card
Number: 0006 12042555
Holder: Vincent Dehors
```

Et en effet, la machine virtuelle a une YubiKey branchée sur un de ses ports USB 2.0.

```
MESSAGE=usb 2-2: Product: YubiKey OTP+FIDO+CCID
MESSAGE=usb 2-2: Manufacturer: Yubico
MESSAGE=input: Yubico YubiKey OTP+FIDO+CCID as /devices/pci0000:00/0000:00:06.0/
        usb2/2-2/2-2:1.0/0003:1050:0407.0002/input/input9
```

1.2 Volatility

J'ai ensuite utilisé `binwalk` et `photorec` avec des résultats sympathiques mais difficilement exploitables. Il est temps de passer à quelque chose de plus efficace. Pour analyser une trace mémoire en profondeur, l'outil de référence est Volatility. Je suis content car je ne l'avais

encore jamais utilisé, c'est donc une belle opportunité de le découvrir.

Après la lecture de quelques tutoriels, j'apprends que la première chose à faire pour utiliser Volatility sur Linux est de générer un profil sur une machine similaire à celle d'où provient la capture (même version du noyau, même distribution Linux). On a déjà trouvé ce qui ressemble à un `uname -a`, mais pour confirmer cette information, on cherche le fragment de log kernel (`dmesg`) qui indique le fichier de démarrage.

```
$ strings mem | grep BOOT_IMAGE
Apr  2 12:56:32 kazbng-backup kernel: [    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-4.19.0-6-amd64
root=UUID=fa99deef-3ad0-4001-b221-48546abe6065 ro quiet
```

L'étape suivante est l'installation d'une machine virtuelle Debian Buster x64 avec un noyau 4.19.0-6. Une fois la VM prête, on y installe Volatility et on génère le profil.

```
$ git clone https://github.com/volatilityfoundation/volatility.git
$ cd volatility/tools/linux
$ make
$ cd ../../../
$ zip $(lsb_release -i -s)_$(uname -r)_profile.zip ./volatility/tools/linux/module.dwarf
/boot/System.map-$(uname -r)
```

On obtient un fichier zip, qu'on rapatrie vers notre machine de travail, puis on supprime la VM Debian toute neuve (puisque je suis déjà en rade d'espace disque... et ce n'est que le début du challenge).

Il existe beaucoup de plugins Linux pour Volatility. Voici ceux qui m'ont été utiles pour résoudre ce challenge :

linux_psaux pour lister les processus. On repère ainsi que le script `backup.sh` est en cours d'exécution.

linux_enumerate_files pour lister tous les chemins de fichiers contenus dans la capture mémoire. On découvre le système de fichiers d'un GNU/Linux classique. La seule chose qui attire l'attention est le dossier `/home` de l'utilisateur bakeup, dans lequel on trouve le script `backup.sh`, mais aussi un fichier `backup.log` et des fichiers de configuration SSH et GPG (dont des clés privées).

linux_find_file pour extraire un à un les fichiers d'intérêt. À noter que ce plugin fonctionne mal dans la version de Volatility issue des dépôts de Kali (v2.6). Utiliser la dernière version de Volatility depuis github (v2.6.1) règle ce problème.

```
$ ./vol.py -f .mem --profile=LinuxDebian_4_19_0-6-amd64_profilex64 linux_psaux
Pid    Uid    Gid    Arguments
[...]
533    1000   1000   /bin/bash ./backup.sh
650    1000   1000   sleep 1d
[...]
$ ./vol.py -f mem --profile=LinuxDebian_4_19_0-6-amd64_profilex64 linux_enumerate_files
[...]
0xffff97531b202b18                    131075 /home/bakeup/backup.log
0xffff97531b201e70                    131093 /home/bakeup/backup.sh
[...]
$ ./vol.py -f mem --profile=LinuxDebian_4_19_0-6-amd64_profilex64 linux_find_file -i 0xffff97531b201e70
-O backup.sh
```

Voici le script `backup.sh` récupéré :

```
#! /bin/bash
# Custom backup script

set +e

# Logging stuff
LOG_FILE="${HOME}/backup.log"
exec &> ${LOG_FILE}

log() {
        echo "$(LC_ALL=C date --utc "+%F %T") $*"
}

log "Starting backup daemon..."

# Check SSH/PGP config
export SSH_AUTH_SOCK="${XDG_RUNTIME_DIR}/gnupg/S.gpg-agent.ssh"
ssh-add -l

while true
do
        log "Starting new backup"
        WORKDIR=$(mktemp -d)
        BACKUPCMD="KHBnX2R1bXAgc3luYXBzZSA7IHRhciBjeiB+L21lZGlhX3NOb3JlKSB8IGd6aXAgLTEgfCBvcGVuc3NsIGVuYyAt
ZSAtYWVzMjU2IC1pdiA1ZDExNWEyOWQxZTcwZmM3Y2VmODQ0MWUzYzRmY2I1MyAtSyBlZDM1OGFmNmNlODIyNDgwZDM4NGRmMmNiOTc
4NTc2Y2Q1MTI5NzM3MzUzOTkzOGVhOTlmMjY4MTNmZmY1MjVmID4gfi9zeW5hcHNlLmJhawo="
        ssh synapse-node "$(echo ${BACKUPCMD} | base64 -d)"

        log "Downloading backup"
        scp synapse-node:synapse.bak ${WORKDIR}/tmp
        split -b 256k -d ${WORKDIR}/tmp ${WORKDIR}/tmp
        rm ${WORKDIR}/tmp

        log "Copying on external drive"
        sudo mount /dev/disk/by-label/external /mnt/external
        backup_date=$(date "+%F-%T")
        for part in $(ls ${WORKDIR}); do
                gpg --encrypt --armor --recipient "Sivi Ha Kerez" -o /mnt/external/backup/${backup_date}.
${part}.backup ${WORKDIR}/${part}
        done
        sha256sum ${WORKDIR}/* > ${WORKDIR}/tmp.sha256sum
        cp ${WORKDIR}/tmp.sha256sum /mnt/external/backup/${backup_date}.sha256sum
        sudo umount /mnt/external # Sync on disk

        # Cleanup
        log "Cleanup"
        srm -r ${WORKDIR}

        # Wait 1 day before next backup
        log "Wait"
        sleep 1d
done
```

Ce script se connecte en SSH à une machine nommée synapse-node, y exécute des commandes puis en rapatrie un fichier `synapse.bak`. Ce fichier est ensuite découpé en morceaux de 256Ko qui sont chiffrés avec `gpg` et sauvegardés sur un stockage externe. Pour finir, le script dort pendant 24 heures avant de recommencer les mêmes opérations. Les commandes exécutées sur la machine synapse-node sont encodées en base64. Les voici décodées :

```
'(pg_dump synapse ; tar cz ~/media_store) | gzip -1 | openssl enc -e -aes256 -iv 5d115a29d1e70fc7cef8
441e3c4fcb53 -K ed358af6ce822480d384df2cb978576cd51297373539938ea99f26813fff525f > ~/synapse.bak\n'
```

Le fichier synapse.bak est donc la concaténation de deux fichiers :

— Le dump au format texte d'une base PostgreSQL nommée `synapse`

— Une archive contenant le dossier `media_store`

Avant d'être rapatrié, ce fichier est chiffré avec une clé AES 256 codée en dur.

1.3 Récupération de synapse.bak

On se doute qu'il va falloir récupérer ce fichier `synapse.bak`... Mais comment faire, puisqu'il n'est pas conservé sur la machine kazbng-backup ? De plus, la machine synapse-node n'est pas accessible pour nous puisqu'elle est sur le réseau local de kazbng-backup. La réponse se trouve dans le fichier `backup.log`.

```
2020-04-02 10:57:09 Starting backup daemon...
2048 SHA256:71as+UT4U87vlrPZ78t1vIowemSPCWqR8117x9zk8qU cardno:000612042555 (RSA)
2020-04-02 10:57:10 Starting new backup
2020-04-02 10:57:10 Downloading backup
2020-04-02 10:57:11 Copying on external drive
2020-04-02 10:57:12 Cleanup
./backup.sh: ligne 44: srm : commande introuvable
2020-04-02 10:57:12 Wait
```

La commande `srm` (secure remove) n'est pas installée sur la machine kazbng-backup, la suppression des fichiers de travail de `synapse.bak` n'a donc pas fonctionné. En regardant de plus près la sortie du plugin linux_enumerate_files, on trouve en effet un dossier temporaire `/tmp/tmp.lbDY6KLaOE` qui contient 86 morceaux chiffrés de `synapse.bak` ainsi qu'un fichier `tmp.sha256sum` qui contient le hash de ces derniers. Voici un script qui utilise Volatility pour récupérer tout le contenu du dossier temporaire.

```python
import subprocess
import os

TARGET = "tmp.lbDY6KLaOE"
CMD = ["python", "vol.py", "-f", "mem", "--profile=LinuxDebian_4_19_0-6-amd64_profilex64"]
os.mkdir(TARGET)

def listdir(dirname):
    cmd = CMD + ["linux_enumerate_files"]
    txt = subprocess.check_output(cmd)
    for line in txt.splitlines():
        if dirname in line:
            s = line.strip().split()
            inode = s[0]
            fname = s[-1].rsplit("/", 1)[1]
            yield inode, fname

for inode, fname in listdir(TARGET+"/"):
    cmd = CMD + ["linux_find_file", "-i", inode, "-O", TARGET +  "/" + fname]
    subprocess.call(cmd)
```

Ce script est assez peu satisfaisant puisqu'il appelle 87 fois Volatility et répète à chaque fois l'initialisation du framework et le chargement de l'image. Mais Volatility ne semble pas vraiment conçu pour être chargé directement depuis Python (l'exemple donné à la toute fin de la documentation [2] est assez peu enthousiasmant). Si quelqu'un a un solution plus élégante, je suis intéressé !

2. https://github.com/volatilityfoundation/volatility/wiki/Volatility-Usage

On vérifie ensuite que tous les fichiers récupérés ont bien le bon hash sha256

```
$ cp -r tmp.lbDY6KLaOE/ /tmp
$ sha256sum /tmp/tmp.lbDY6KLaOE/tmp?? > extracted.sha256sum
$ diff extracted.sha256sum /tmp/tmp.lbDY6KLaOE/tmp.sha256sum
```

Puis on reconstitue le fichier `synapse.bak` original en concaténant les 86 morceaux et en les déchiffrant. Un `grep` sur ce fichier nous permet de trouver le premier flag.

```
$ cat tmp?? > synapse.bak.gz.enc
$ openssl enc -d -aes256 -iv 5d115a29d1e70fc7cef8441e3c4fcb53 -K ed358af6ce822480d384df2cb978576
cd51297373539938ea99f26813fff525f -in synapse.bak.gz.enc > synapse.bak.gz
$ gunzip synapse.bak.gz > synapse.bak
$ grep "SSTIC" synapse.bak
"Le mot de passe qui sera demand\\u00e9 est SSTIC{f8693a5a340b67c4ac13abb93e57da55b935577fc76cb
e14c223fb5912ccc795}."
```

```
SSTIC{f8693a5a340b67c4ac13abb93e57da55b935577fc76cbe14c223fb5912ccc795}
```

2 Niveau 2

2.1 Analyse de synapse.bak

Le fichier `synapse.bak` dont nous sommes entré en possession est la concaténation de deux fichier. Pour retrouver la frontière entre ceux-ci, on utilise `binwalk`. La première partie du fichier est un dump de base PostgreSQL au format texte. La seconde partie est une archive gzip qui contient une arborescence `var/lib/synapse/media_store/` renfermant de nombreux fichiers indexés (mais principalement des images de chat).

Le dump PostgreSQL est relativement volumineux (1.2Mo). Quand on regarde aux alentours du premier flag, on trouve des phrases échangées entre les utilisateurs d'une messagerie instantanée. Les messages sont enrobés de nombreuses métadonnées et formatés en JSON. Avec quelques `grep`, on récupère le contenu de plusieurs messages qui vont nous éclairer pour la suite.

```
* Bienvenue sur Kazh-Boneg, la nouvelle messagerie chiffree de la Bierique !
* Elle utilise des logiciels libres (matrix.org et Riot), donc si vous voulez contribuer a son evolution,
c'est completement possible :)
* ah, il semble que l'on puisse creer des salons de discussion chiffres, ce qui n'est pas le cas de #public
* En effet, le chiffrement est optionnel. Quand il est desactive, les administrateurs de Kazh-Boneg
(dont je fais partie) peuvent lire les messages.
* Je viens de creer le salon #cache pour les discussions generales qui seront chiffrees.
* Gwrizienn : Comment je rejoins ton nouveau salon ?
* cache.megolm_keys.txt
* Du coup, voici un export de la cle Megolm de #cache
* Le mot de passe qui sera demande est SSTIC{f8693a5a340b67c4ac13abb93e57da55b935577fc76cb
e14c223fb5912ccc795}.
```

Nous sommes donc bien en présence d'un dump de Kazh-Boneg, le logiciel de messagerie instantanée qu'il faut analyser d'après la consigne initiale du challenge. Grâce aux messages échangés sur le salon `#public`, on apprend que Kazh-Boneg utilise les technologies open-source `matrix` et `Riot` (tout comme Tchap, la messagerie de l'État français).

Le but de ce niveau est d'accéder au salon chiffré `#cache`, dont la clé Megolm[3] a été échangée sur le salon `#public` et est probablement quelque part dans le media_store. Mais pour pouvoir y accéder, il va falloir reconstruire le serveur Kazh-Boneg en local.

Ce deuxième niveau consiste donc principalement en de l'administration système, puisqu'il va falloir déployer un seveur `Synapse` (le serveur de référence pour le protocole Matrix), une base `PostgreSQL` et un client `Riot`.

2.2 Synapse

Le serveur `synapse` est directement disponible dans les dépôts de Kali (paquet matrix-synapse). Il faut ensuite personnaliser le fichier de configuration `homeserver.yaml`. On peut par exemple autoriser l'accès invité, ou l'enregistrement de nouveaux comptes. Mais la seule modification indispensable est l'utilisation de PostgreSQL comme base de données, plutôt que SQLite par défaut.

```
# Database configuration
database:
  name: "psycopg2"
  args:
      user: synapse
      password: synapse
      database: synapse
      host: kazh-boneg.brq
      cp_min: 5
      cp_max: 10
```

Pour avoir accès à toutes les images et fichiers échangés sur Kazh-Boneg, il faut également copier le media_store précédemment récupéré, vers le bon endroit (par défaut dans `/var/lib/matrix-synapse/media/`).

2.3 PostgreSQL

On s'attaque ensuite à l'installation et à la configuration de PostgreSQL, qu'on trouve également dans les dépôts Kali. On crée un utilisateur nommé `synapse`, puis une base de données `synapse` qu'on attribue à cet utilisateur. On peut ensuite restaurer le dump issu de `synapse.bak`.

```
$ sudo -u postgres bash
$ createuser --pwprompt synapse
$ psql
postgres=# CREATE DATABASE synapse
    ENCODING 'UTF8'
    LC_COLLATE='C'
    LC_CTYPE='C'
    template=template0
    OWNER synapse;
postgres=# quit
$ psql synapse < dump.sql
```

3. un algorithme dédié au chiffrement des conversations groupées

En explorant brièvement le contenu de la base PostgreSQL de Kazh-Boneg, on trouve 16 utilisateurs enregistrés, dont Gwrizienn [4] qui est l'administrateur. Assez rapidement, je me suis rendu compte qu'il allait être pratique de pouvoir se connecter sur le compte de chacun de ces utilisateurs, notamment pour lire les messages privés échangés entre deux personnes (s'ils ne sont pas chiffrés... ou alors quand nous auront récupéré les clés). J'ai donc créé un nouvel utilisateur quelconque pour générer un hash de mot de passe que je maîtrise, puis j'ai attribué son hash à tout le monde.

```
$ psql synapse
synapse=# update users SET password_hash ='$2b$12$nHv/lcPsMjTDsqPiEEvPG.lsuD36YCQ.Zg/STxcHlZQWwKw9D1vF2';
UPDATE 16
synapse=#
```

2.4 Riot

Le dernier logiciel à déployer est `Riot`, un client web pour Matrix. J'ai utilisé la version 1.5.15 de Riot, disponible sur github [5], avec une petite personnalisation du fichier `config.json` afin de se connecter par défaut à notre serveur synapse.

```
"default_server_config": {
    "m.homeserver": {
        "base_url": "http://localhost:8008",
        "server_name": "kazh-boneg.brq"
    },
```

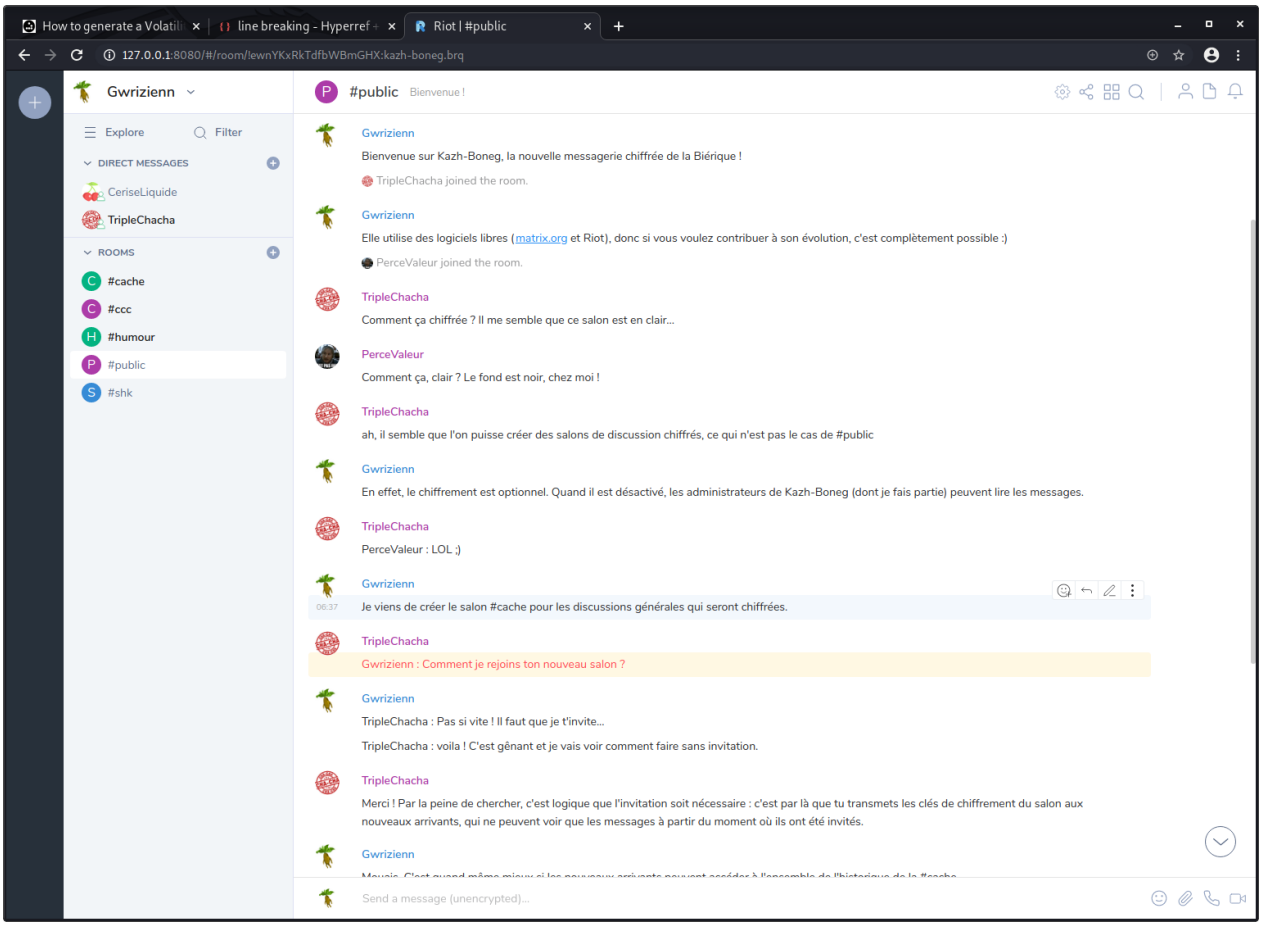
Pour lancer Riot, j'ai utilisé `live-server`, une sorte de SimpleHTTPServer python, mais orienté node.js. Tout est prêt, on peut démarrer tous les services et explorer Kazh-Boneg.

```
$ sudo service postgresql start
$ sudo service matrix-synapse start
$ cd riot-v1.5.15
$ live-server
```

2.5 À nous la #cache

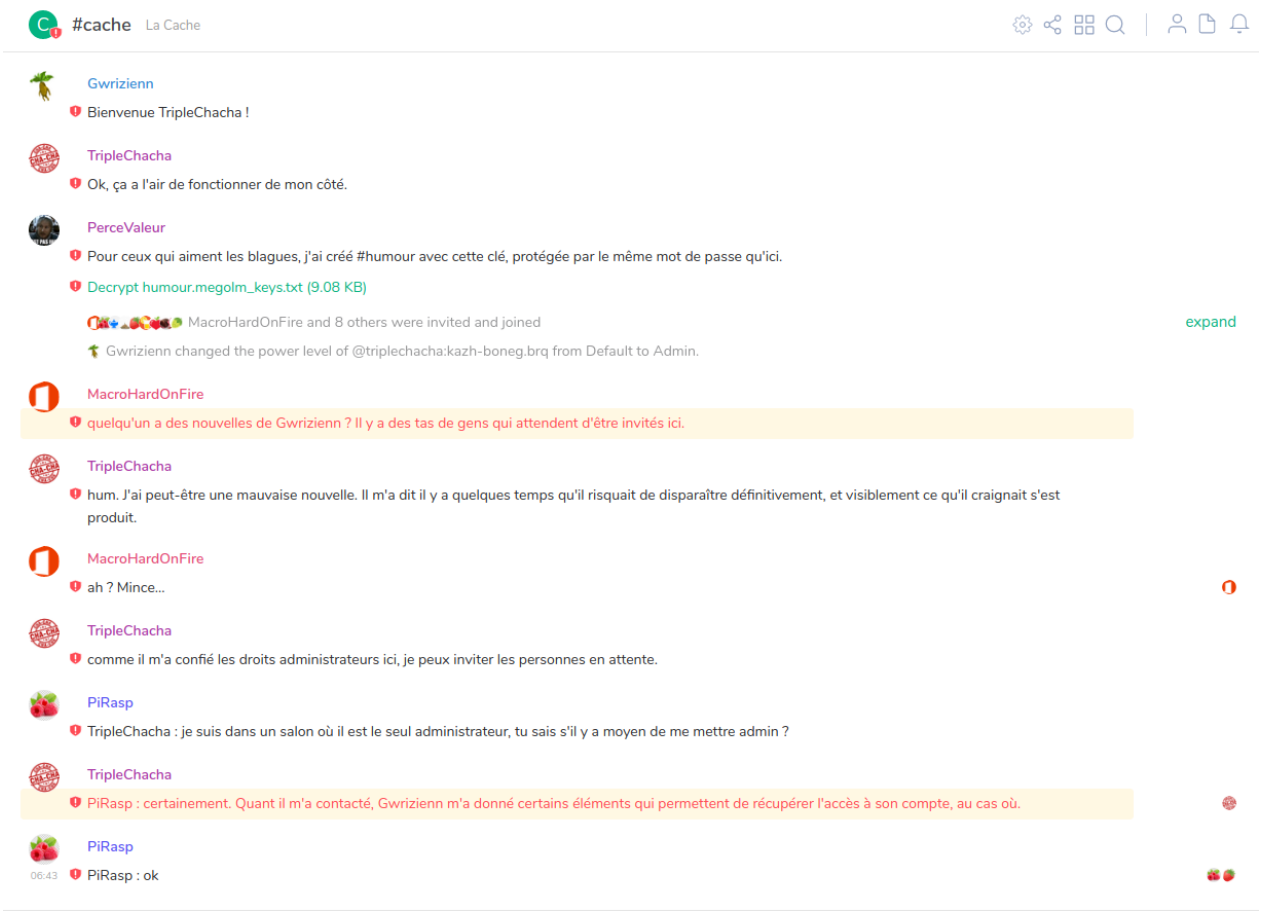
On décide de se connecter avec le compte de Gwrizienn. Voici ce à quoi ressemble le salon `#public`.

4. « racine » en breton
5. https://github.com/vector-im/riot-web/releases



On constate que Gwrizienn communique également dans d'autres salons et via des messages privés, mais pour l'instant, toutes ces communications sont chiffrées. Le seul autre salon en clair est `#ccc` (Comité Contre les Chats), sur lequel s'échangent des photos de chats (mal) générés par une IA.

Depuis le salon `#public`, on télécharge la clé `cache.megolm_keys.txt`, qu'on charge dans le compte Riot de Gwrizienn et qu'on déverrouille avec le flag du premier niveau. Tous les échanges du salon `#cache` sont désormais en clair !



Dans le salon `#cache`, on récupère le fichier `humour.megolm_keys.txt` qui permet de déchiffrer le salon `#humour`, qui contient beaucoup de mèmes, mais pas de flag. Dans la suite des discussions du salon `#cache`, on apprend que Gwrizienn a disparu de manière suspecte, des gens en ayant après lui. Au cas où, il avait laissé des instructions à l'utilisateur TripleChaCha pour récupérer l'accès à son compte.

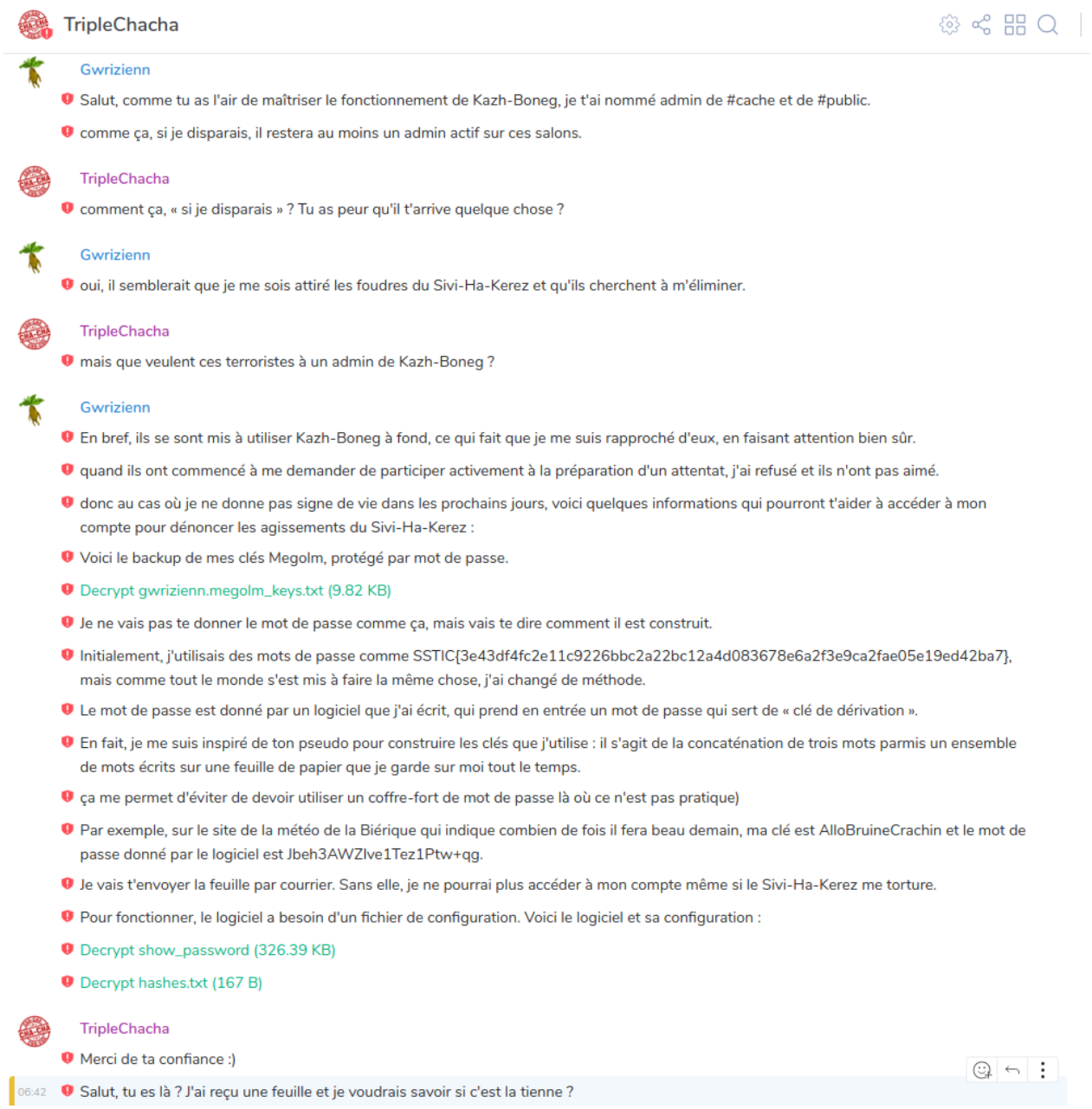
Et en effet, grâce aux clés Megolm déjà chargées, on constate que la conversation privée entre Gwrizienn et TripleChaCha est désormais en clair. L'une des phrases contient le deuxième flag.

`SSTIC{3e43df4fc2e11c9226bbc2a22bc12a4d083678e6a2f3e9ca2fae05e19ed42ba7}`

3 Niveau 3

3.1 Un accès de secours

Voici la conversation entre Gwrizienn et TripleChaCha.



La conversation se termine par TripleChaCha qui a reçu une feuille étrange. Mais Gwrizienn ne répond déjà plus.

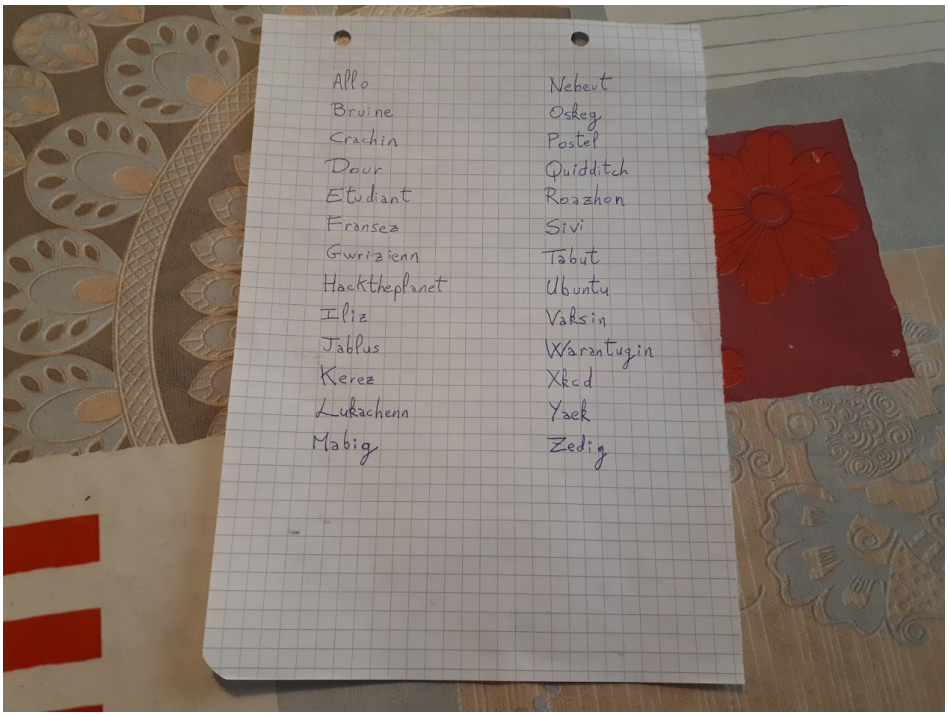
Gwrizienn a des problèmes avec le Sivi-Ha-Kerez, l'organistation soupçonnée de préparer un attentat contre la Biérique. En prévision de son éventuelle disparition, il a donné ses clés Megolm à TripleChaCha, protégées par un mot de passe qui est dérivé à partir d'un programme de sa conception. Pour cette troisième étape du challenge, nous sommes en présence de 4 fichiers :

show_password Un exécutable ELF x64 servant à dériver des mots de passe.

hashes.txt Un fichier contenant deux hashs de mot de passe au format *sstic*. Un hash pour l'entrée `météo` dont on connaît le mot de passe d'origine (AlloBruineCrachin), l'autre pour `chat`, qu'il va falloir casser.

gwrizienn.megolm_keys.txt les clés Megolm protégées par le mot de passe de l'entrée `chat`.

photo.jpg Une feuille manuscrite qui contient les 26 mots que Gwrizienn utilise pour générer ses mots de passe.



3.2 Analyse en boîte noire

On commence par tester le logiciel `show_password` dans un environnement jetable. On constate rapidement un comportement assez suspect. Quand on fournit le bon mot de passe d'une entrée, celui-ci est dérivé en environ 5 secondes. Mais quand on fournit un mauvais mot de passe, le programme met plus de 45 secondes avant de s'arrêter !

```
$ ./show_password
Usage: show_password hashes.txt name
$ time ./show_password hashes.txt meteo
Password for meteo: AlloBruineCrachin
Good password
Welcome
Your password is Jbeh3AWZIve1Tez1Ptw+qg

real        0m5.148s

$ time ./show_password hashes.txt meteo
Password for meteo: CrachinBruineAllo
Wrong password

real        0m46.712s
```

Les mots de passe de Gwrizienn sont construits à partir de la concaténation de trois mots issus de sa liste qui en contient 26. Le nombre de mots de passe possible est donc très faible, à savoir 26^3 (17576), voire même 15600 si on considère qu'il ne réutilise pas deux fois le même mot dans un mot de passe. Il semble donc envisageable de bruteforcer le mot de passe de l'entrée `chat`... si toutefois on parvient à tester les mots de passe à un rythme plus soutenu que 1mdp/45sec.

Pour comprendre pourquoi la dérivation est si longue dans le cas d'un mauvais mot de passe, on ouvre le programme dans Ghidra. Puis on le referme. En effet, d'après le nom de certaines fonctions, il est écrit en Rust. Et les chaînes de caractères sont toutes concaténées sans `0x00`, ce qui ne facilite pas la recherche de références croisées. Donc avant de se plonger davantage dans la rétroconception, on va adopter une démarche boîte noire.

```
__rust_alloc              0010e930
__rust_dealloc            0010e940
__rust_realloc            0010e950
__rust_alloc_zeroed       0010e960
rust_oom                  00124390
rust_begin_unwind         00124cf0
rust_panic                00125370
__rust_maybe_catch_panic  00127580
__rust_start_panic        00127590
```

Pour comprendre la différence significative de temps entre un bon et une mauvais mot de passe, j'ai utilisé `Valgrind` pour faire du profiling et `kcachegrind` pour visualiser les résultats.

```
$ valgrind --tool=callgrind ./show_password hashes.txt meteo
[...]
$ kcachegrind callgrind.out.53672
```

Voici le profil du programme show_password lorsque l'on fournit le bon mot de passe

Incl.	Self	Called	Function	Location
100.00	0.00	4	<cycle 1>	show_password
98.74	0.00	1 665	__rust_maybe_catch_panic <cycle 1>	show_password
98.74	0.00	1 664	0x0000000000014820	show_password
98.73	0.00	1 664	0x0000000000014080	show_password
98.73	1.31	1 664	0x00000000000012dc0	show_password
89.22	81.03	5 342 048	0x000000000000110e0	show_password
17.54	17.54	34 297 692	memcpy@GLIBC_2.2.5	libc-2.30.so: memmove-vec-unaligned-erms.S
1.25	0.00	(0)	0x0000000000001090	ld-2.30.so
1.25	0.00	1	0x000000000000ad50	show_password
1.25	0.00	1	(below main)	libc-2.30.so: libc-start.c
1.25	0.00	1	0x000000000000cd80	show_password
1.25	0.00	1	std::rt::lang_start_internal	show_password
1.18	0.00	22	0x000000000000fdf0	show_password
1.18	0.00	22	0x0000000000014eb0	show_password
1.18	0.02	22	0x0000000000014c90	show_password
1.18	0.00	21	argon2::argon2::verify_raw <cycle 1>	show_password
0.06	0.00	1	argon2::argon2::hash_encoded <cycle 1>	show_password
0.05	0.00	374	0x0000000000013370	show_password
0.05	0.00	10 956	0x0000000000010e20	show_password
0.05	0.00	22	0x0000000000010670	show_password
0.04	0.00	10 956	blake2b_simd::State::finalize	show_password
0.04	0.04	11 000	0x0000000000016ea0	show_password
0.01	0.00	69 866	free	libc-2.30.so: malloc.c
0.01	0.01	40 615	_int_free	libc-2.30.so: malloc.c
0.01	0.00	1 642	start_thread'2 <cycle 1>	libpthread-2.30.so: pthread_create.c, allocatestack.c, exit-thread.h
0.01	0.00	21 652	malloc	libc-2.30.so: malloc.c
0.01	0.00	1 663	__libc_thread_freeres	libc-2.30.so: thread-freeres.c
0.01	0.00	1 664	__malloc_arena_thread_freeres	libc-2.30.so: malloc.c, arena.c

Et voici le profil pour un mauvais mot de passe

Incl.	Self	Called	Function	Location
100.00	0.00	1	<cycle 1>	show_password
98.51	0.00	16 385	__rust_maybe_catch_panic <cycle 1>	show_password
98.51	0.00	16 384	0x0000000000014820	show_password
98.51	0.00	16 384	0x0000000000014080	show_password
98.51	1.31	16 384	0x00000000000012dc0	show_password
89.01	80.82	52 629 504	0x00000000000110e0	show_password
17.74	17.74	341 900 646	memcpy@GLIBC_2.2.5	libc-2.30.so: memmove-vec-unaligned-erms.S
1.47	0.00	(0)	0x0000000000001090	ld-2.30.so
1.47	0.00	1	0x000000000000ad50	show_password
1.47	0.00	1	(below main)	libc-2.30.so: libc-start.c
1.47	0.00	1	0x000000000000cd80	show_password
1.47	0.00	1	std::rt::lang_start_internal	show_password
1.45	0.00	256	argon2::argon2::verify_raw <cycle 1>	show_password
1.39	0.00	256	0x000000000000fdf0	show_password
1.39	0.00	256	0x0000000000014eb0	show_password
1.39	0.03	256	0x0000000000014c90	show_password
0.06	0.00	4 352	0x0000000000013370	show_password
0.06	0.00	127 488	0x0000000000010e20	show_password
0.06	0.00	256	0x0000000000010670	show_password
0.05	0.00	127 488	blake2b_simd::State::finalize	show_password
0.05	0.05	128 000	0x0000000000016ea0	show_password
0.01	0.00	689 628	free	libc-2.30.so: malloc.c
0.01	0.01	400 803	_int_free	libc-2.30.so: malloc.c
0.01	0.00	16 361	start_thread'2 <cycle 1>	libpthread-2.30.so: pthread_create.c, allocatestack.c, exit-thread.h
0.01	0.00	213 284	malloc	libc-2.30.so: malloc.c
0.01	0.00	16 383	__libc_thread_freeres	libc-2.30.so: thread-freeres.c
0.01	0.00	16 384	__malloc_arena_thread_freeres	libc-2.30.so: malloc.c, arena.c

Les deux profils sont assez similaires. On trouve des appels à Argon2 (une fonction de dérivation de clé) et à BLAKE2 (une fonction de hachage cryptographique). La grosse différence est le nombre d'appels à certaines fonctions. Là où certaines fonctions sont appelées 22 fois et 1664 fois pour un bon mot de passe, elles sont respectivement appelées 256 et 16384 fois pour un mauvais mot de passe.

3.3 Analyse en boîte blanche

Aidé par ces informations, on ré-ouvre Ghidra et on trouve la boucle responsable de ce comportement, qui est en fait inlinée dans un très gros `main`. À chaque tour de boucle, on fait un `sleep` d'une milliseconde (donc parfaitement négligeable) puis une vérification Argon2. Au bout de 20 tours de boucle, on examine le résultat de la vérification Argon2 pour savoir si le mot de passe est le bon, auquel cas on quitte la boucle. Sinon, au bout de 256 tours, on quitte la boucle en signalant que le mot de passe est mauvais.

Ce qui est amusant (et qu'on pourra vérifier en dynamique avec `gdb`), c'est que la vérification Argon2 est exactement la même lors de chacun de ces tours de boucle. Inutile de passer 20 fois dans la boucle pour savoir si le mot de passe est bon (et encore moins 256 fois pour savoir s'il est mauvais). Un seul passage suffit pour tester un mot de passe.

Donc plutôt que de reverser complètement et de réimplémenter l'algorithme de dérivation de clé, j'ai choisi une approche beaucoup plus pragmatique, à savoir patcher deux instructions du programme `show_password` pour ne faire qu'un seul tour de boucle, que le mot de passe soit bon ou mauvais.

```python
with open("show_password", "rb") as f:
    x = bytearray(f.read())

# Do not wait 20 rounds to check if the password is good
# 72 BA / jb loop_start  ==> 90 90 / nop nop
x[50564:50566] = b"\x90\x90"

# Do not wait 256 rounds to check if the password is bad
# 72 B2 / jz loop_start ==> 72 B9 / jz loop_start+7 (will directly jumps to bad_password code)
x[50573] = 0xB9

with open("show_password_patched", "wb") as f:
    f.write(x)
```

Avec notre version patchée, tester un mot de passe ne prend plus que 0.3 secondes. On écrit donc un script qui va tester les 15600 candidats possibles.

```python
import itertools
from pwn import *

WL = ["Allo", "Bruine", "Crachin", "Dour", "Etudiant", "Fransez", "Gwrizienn", "Hacktheplanet", "Iliz",
      "Jablus", "Kerez", "Lukachenn", "Mabig", "Nebeut", "Oskeg", "Postel", "Quidditch", "Roazhon", "Sivi",
      "Tabut", "Ubuntu", "Vaksin", "Warantugin", "Xkcd", "Yaek", "Zedig"]

context(arch="amd64", os="linux")

for x in itertools.permutations(WL, r=3):
    c = "".join(x)
    p = process(["./show_password_patched", "hashes.txt", "chat"])
    p.sendline(c)
    r = p.recvall()
    if b"Welcome" in r:
        print(c)
        print(r.decode())
        break
```

```
$ python3 solve.py
[...]
OskegLukachennMabig
Password for chat: Good password
Welcome
Your password is SSTIC{556c34304f6279736e516f3267794e4963637a637051}
```

Le bon mot de passe était donc OskegLukachennMabig [6]. À noter que dans show_password, on observe un traitement spécial lorsque l'entrée `chat` est demandée : une fois dérivé, le mot de passe est encodé en hexadécimal et précédé de SSTIC{}. Voilà qui conclut le troisième niveau.

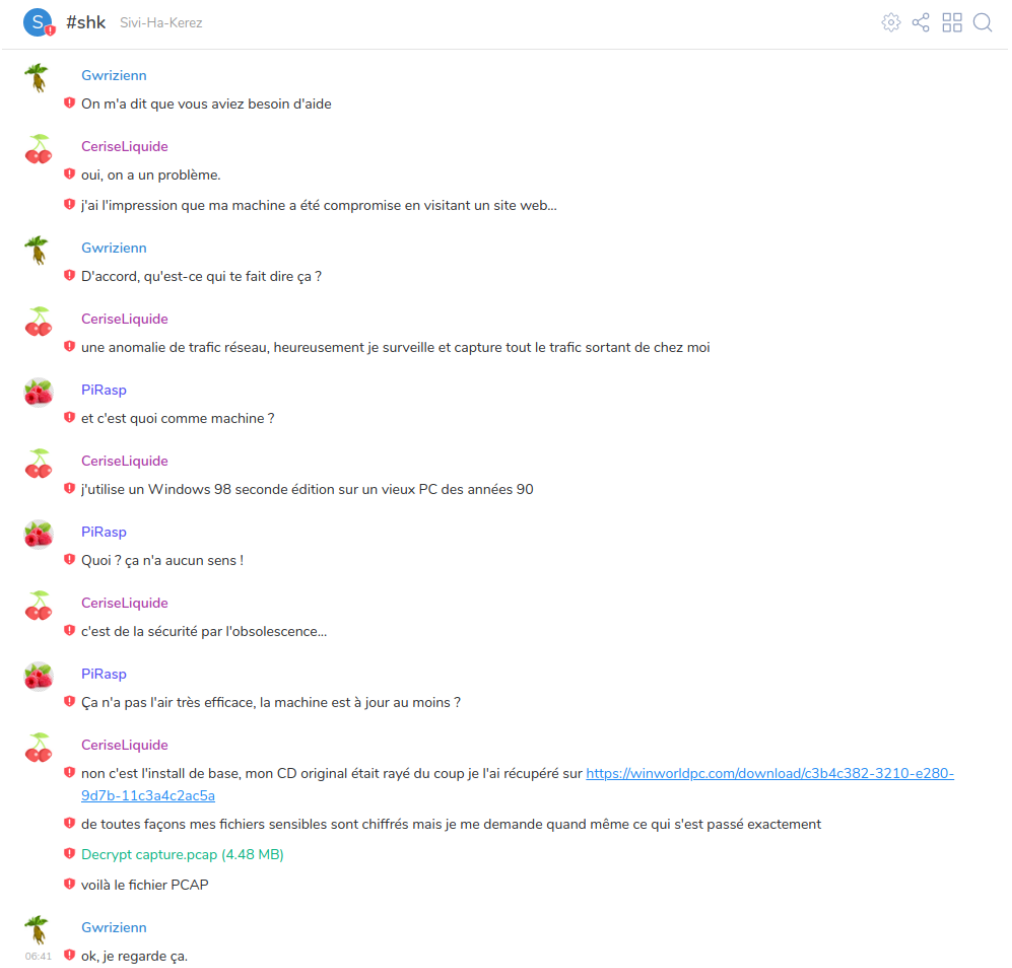
```
SSTIC{556c34304f6279736e516f3267794e4963637a637051}
```

6. Après traduction de breton vers français, on obtient Osque-Crachin-Enfant

4 Niveau 4

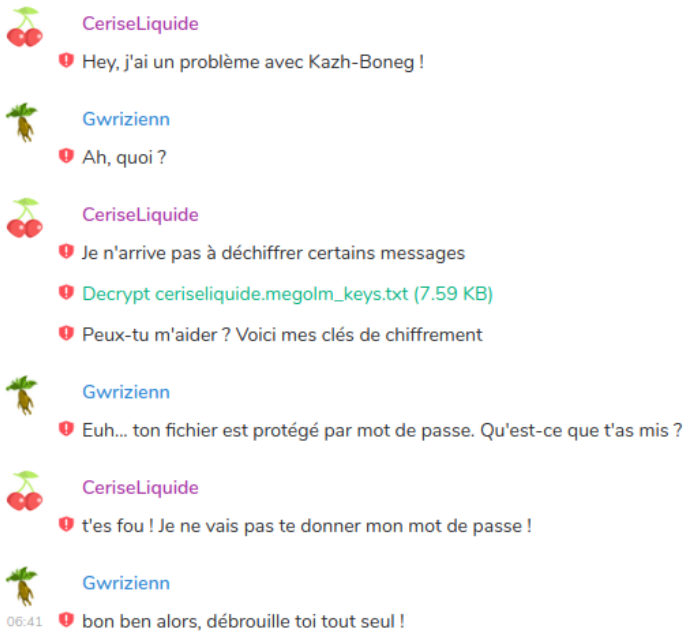
4.1 Retour en 1998

Grâce aux clés Megolm de Gwrizienn, le salon `#shk` (pour Sivi-Ha-Kerez) est désormais en clair. Sur celui-ci se tient une conversation entre Gwrizienn, CeriseLiquide et PiRasp.



CeriseLiquide utilise un vieux PC sous Windows 98 et pense s'être fait compromettre en visitant un site web. Il donne à Gwrizienn un fichier `capture.pcap` à analyser.

Les clés Megolm permettent également de déchiffrer une conversation privée entre CeriseLiquide et Gwrizienn.



CeriseLiquide nous donne ses clés Megolm, mais pas son mot de passe. On ne peut donc rien en faire... pour l'instant.

4.2 Analyse du pcap

Dans le fichier `capture.pcap`, on observe deux parties distinctes. Tout d'abord, il y a du trafic HTTP qui correspond à la visite de la page « *La carotte est-elle un légume* » sur le site web cookismo.fr [7] par CeriseLiquide. Un article *passionnant* qui nous apprend que depuis 1988, sur décision de la Communauté Européenne, la carotte est considéré à la fois comme un fruit et un légume, pour des sombres histoires de réglementation sur les confitures.

Dans la seconde partie du PCAP, on trouve du trafic non identifié sur le port 443. Ce trafic n'est pas du SSL mais il semble malgré tout chiffré. De plus, il est à sens unique, du PC de CeriseLiquide vers l'extérieur. À noter qu'il n'y a que deux adresses IP (privées) dans la capture, 192.168.20.128 pour le PC de CeriseLiquide et 192.168.20.1 pour sa passerelle par défaut.

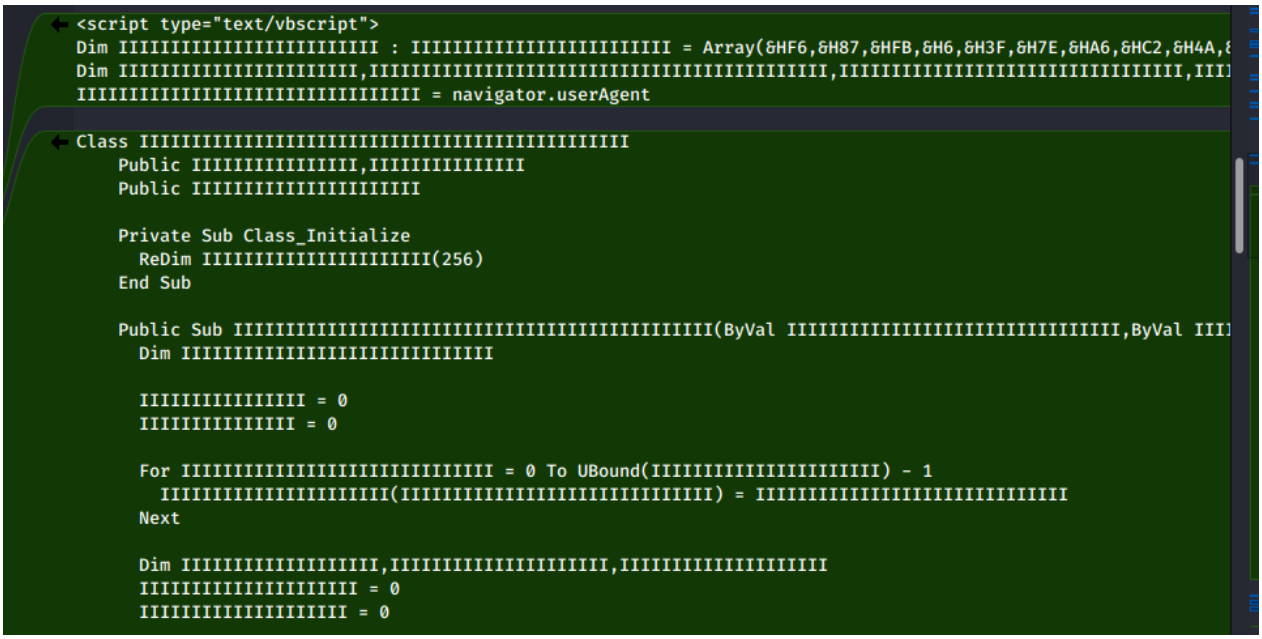
Avec l'option File->Export Objects->HTTP de Wireshark, on extrait tous les fichiers qui ont transité via HTTP, dont `16254.html` qui contient le code de la page visitée sur cookismo. Pour identifier plus facilement une potentielle compromission, j'ai récupéré la page d'origine à l'aide d'un `wget` afin de faire un `diff` [8].

On trouve trois principales différences entre la page 16254.html visitée par CeriseLiquide et la page 16254.html réelle.

1. Un TRÈS gros morceau de vbscript a été rajouté à la page visitée par CeriseLiquide. Le code est très suspect puisqu'il est obfusqué, toutes les variables s'appelant IIIIIIIIII. Il s'agit probablement de l'exploit que l'on recherche et qu'il va falloir désobfusquer.

7. http://www.cookismo.fr/la-carotte-est-elle-un-legume/16254
8. Enfin... plutôt un `meld` !

2. Un argument `onload="a()"` a été rajouté dans la balise `body` de CeriseLiquide, ce qui va avoir pour effet de déclencher l'exécution de l'exploit.
3. Du code JavaScript a été supprimé (lié notamment à Google Analytics), probablement pour éviter du trafic vers des sites externes pour les besoins du challenge.



4.3 Désobfuscation vbscript

Pour désobfusquer et analyser le code vbscript, je n'ai pas utilisé d'outil sophistiqué, simplement un éditeur de texte et de la documentation vbscript.

La première étape a été de renommer toutes les variables dans des noms plus ergonomiques, ne serait-ce que pour pouvoir faire des Ctrl+F. Pour ceci, l'astuce est de faire des Ctrl+H en commençant d'abord par le nom le plus long, puis on retire les I un par un et on recommence. Une fois le renommage terminé, on identifie trois parties dans le code VBSscript.

1. Un grand tableau de 25Ko qui contient des données en hexa, qui semblent chiffrées.
2. Une classe qui fait de la cryptographie sur ce tableau de données.
3. Une fonction a() qui va manifestement exploiter une vulnérabilité dans la vieille version d'Internet Explorer 5.0 présente sur le Windows 98 de CeriseLiquide.

On peut d'ores et déjà faire la supposition que le tableau de données va contenir le shellcode dans lequel la fonction a() va sauter, une fois l'exploitation réussie. Pour avancer, on va donc essayer de déchiffrer le tableau.

4.4 Déchiffrement du shellcode

Pour comprendre comment la classe vbscript déchiffre le tableau de données, j'ai – comme souvent – entrepris de recoder l'équivalent en Python.

Et une fois la traduction vbscript vers Python presque terminée, je me suis rendu compte que j'étais en train de recoder le célèbre algorithme de chiffrement par flux RC4.

La clé utilisée pour chiffrer en RC4 est le user-agent de l'utilisateur qui visite le site web. De cette façon, l'attaque devient ciblée puisqu'elle ne fonctionnera que pour un user-agent bien précis.

Ce user-agent utilisé par CeriseLiquide, on le retrouve facilement dans le pcap :

```
Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)
```

Mais malheureusement pour moi, le pragmatisme m'avait momentanément quitté et je n'ai pas tout de suite pensé à regardé dans le pcap...À la place, j'ai utilisé le lien de téléchargement de l'ISO Windows 98 SE qui nous est fourni au début du niveau, et j'ai procédé à l'installation de Windows 98 sous VMWare (en m'aidant de ce tutoriel [9]). J'ai lancé un SimpleHTTPServer sur mon hôte, que je suis allé visiter avec l'Internet Explorer 5.0 de ma machine virtuelle. J'étais alors tout content de trouver le bon User-agent visé par l'exploitation... Puis j'ai réalisé qu'en fait il suffisait de regarder dans le pcap initial... Mais au moins, la bonne nouvelle c'est que l'installation de la VM Windows 98 était faite, et que ça sera utile pour la suite.

On a l'algorithme de chiffrement et la clé, on peut déchiffrer le gros tableau de données situé au début du code vbscript. Il y a quand même une petite subtilité à prendre en compte, puisque l'algorithme consomme sa propre clé *à blanc* avant d'utiliser la suite du flux pour chiffrer/déchiffrer. Voici un petit script Python qui déchiffre le tableau de données.

```python
from Crypto.Cipher import ARC4
with open("payload", "rb") as f:
    payload = f.read()

ua = b"Mozilla/4.0 (compatible; MSIE 5.0; Windows 98; DigExt)"

rc4 = ARC4.new(ua)
rc4.decrypt(ua)
plain = rc4.decrypt(payload)
with open("shellcode", "wb") as f:
    f.write(plain)
```

La tableau déchiffré commence par une chaîne de caractères qui sert à obfusquer le code de la fonction a().

```
v:shape_shape_vgRuntimeStyledashstylearrayitemlength
```

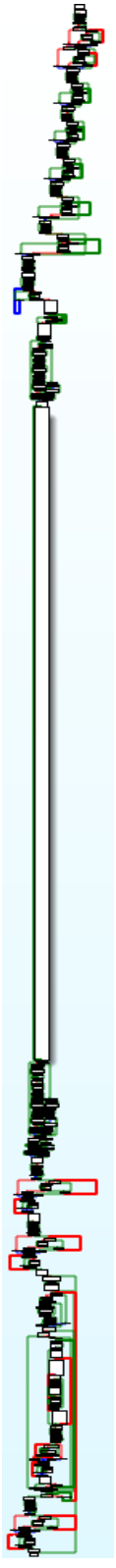
Dans cette fonction a(), le reste du tableau déchiffré n'est utilisé qu'à un seul endroit. L'attribut `.src` d'un objet va pointer sur le tableau à partir de l'offset 58. Or, il se trouve qu'à partir ce cet offset, le tableau déchiffré contient le code x86 valide d'un shellcode. Notre ami CeriseLiquide semble bel et bien avoir été victime d'une RCE. Cette fois, on va donc retrouver un peu de pragmatisme et faire abstraction de toute la partie exploitation de la

9. https://dans-things.com/index.php/2019/08/22/how-to-install-windows-98-second-edition-using-vmware-player/

vieille vulnérabilité Internet Explorer. On jette donc tout le reste du code vbscript pour plonger directement dans le shellcode.

4.5 Première analyse statique du shellcode

Pour cette partie, j'ai utilisé à la fois Ghidra et IDA Free. L'analyse statique promet d'être assez laborieuse, puisque le shellcode est particulièrement gros. Il est composé de 3 fonctions (une gigantesque de 20Ko et deux minuscules) puis de 5Ko de données. Je ne résiste pas au plaisir de vous montrer ce à quoi ressemble le graphe de la fonction principale du shellcode.



Dans ce graphe qui pique les yeux, on peut visuellement découper le shellcode en plusieur morceaux. Au début, des traitements similaires sont effectués plusieurs fois. Ensuite, le gros bloc linéaire contient sans aucun doute de la cryptographie. Et le shellcode se termine par des traitements qui ont lieu dans une grande boucle. Regardons tout ça de plus prêt.

Pour fonctionner, un shellcode doit avoir (au moins) deux propriétés :
— Il doit être *position independant*, c'est à dire qu'il doit pouvoir s'exécuter correctement quelle que soit l'adresse mémoire à laquelle il est chargé.
— S'il a besoin d'utiliser des API système (typiquement Kernel32.dll) alors il doit d'abord trouver un moyen de récupérer un pointeur vers les DLL dont il a besoin, puis résoudre tous les symboles à la main. C'est une opération assez pénible puisqu'il faut, entre autres, parser soi-même le format PE. Dans le cas d'une exploitation à distance sous Windows, le point de départ est souvent la structure opaque PEB (Process Environment Block).

On retrouve bien ces deux propriétés dans notre shellcode. L'une des premières instruction est un `call $+5`. C'est une prise de vue qui sert à récupérer la valeur actuelle de EIP, et ainsi pouvoir calculer l'adresse absolue des données situées en bout de shellcode. L'instruction qui suit la prise de vue est `mov eax,fs:[0x30]`, qui a pour effet de récupérer un pointeur vers le PEB qui sera ensuite beaucoup réutilisé dans la suite du shellcode.

En faisant une première passe de reverse sur l'ensemble du shellcode, on parvient à isoler plusieurs parties :

1. Le shellcode commence par une résolution de symboles répétée une dizaine de fois. Le shellcode ne contient pas directement de chaines de caractères avec le nom des fonctions qu'il souhaite importer à la main. A la place, il utilise un hash (qu'on va étudier plus en détail dans la section suivante).
2. On trouve ensuite un très gros bloc cryptographique qui manipule des données du shellcode. Ces données se révèlent être des tables AES précalculées. À ce stade, on peut donc supposer être en présence d'un chiffrement AES ou d'une dérivation de clé.
3. Le bloc AES est à la fois précédé et succédé par une fourberie à base de `far call` dont on reparlera plus tard.
4. Dans la dernière partie du shellcode, les fonctions importées sont utilisées au sein d'une boucle. On reconnait aussi des blocs qui ressemblent à de

la crypto. Mais pour analyser cette partie, il faut d'abord résoudre les imports.

Cependant, bien qu'ayant réussi à séparer ces différents traitements, l'analyse reste ardue du fait que tout le code est inliné dans la même fonction. Ainsi, il n'est pas trivial de repérer quelles sont les entrées/sorties des différents blocs. Typiquement, trouver ce qui rentre et ce qui ressort du gros bloc AES ne saute pas aux yeux. Pour ne rien simplifier à la rétro-conception, les même zones de variables locales sont utilisées par les différents morceaux du shellcode pour y stocker des choses différentes. On perd donc du temps à déclarer des Unions dans IDA et à se mélanger les pinceaux.

4.6 Résolution des imports

il y a plusieurs moyens de récupérer un pointeur sur Kernel32.dll (ou d'autres DLL utiles) à partir d'un PEB. Mais je n'ai pas trouvé de documentation pertinente sur le PEB de Windows 98, donc je n'ai pas cherché à savoir le chemin exact emprunté par le shellcode pour arriver à ses fins. J'ai plutôt *guessé* ce comportement.

Une fois qu'on a récupéré un pointeur vers Kernel32.dll, il faut parser le format PE à la main pour parcourir les fonctions contenue dans la DLL et récupérer des pointeurs vers celles qui nous intéressent. Ici, il y a deux écoles. Le plus simple est de récupérer un pointeur vers les deux fonctions LoadLibrary et GetProcAddress. Ensuite on peut les utiliser pour importer de manière beaucoup plus ergonomique toutes les autres fonctions dont le shellcode aura besoin. Ou sinon, on peut récupérer à la main toutes les fonctions dont on a besoin. Dans ce shellcode, c'est cette deuxième approche qui a été adoptée.

Pour résoudre ses imports, le shellcode parcourt des DLL et calcule un hash pour chaque nom de fonction, puis il compare chaque hash avec des valeurs codées en dur. J'ai reconnu la fonction de hachage non cryptographique utilisée. Il s'agit de `djb2`, dont la constante d'initialisation (5381) a été remplacée par 0x53544943 (STIC). Cette fonction de hachage non cryptographique produit des condensats de 4 octets.

Mon plan pour retrouver les fonctions importées par le shellcode est de récupérer la liste des fonctions des principales DLL de Windows, de calculer leur hash `djb2` modifié, puis de les comparer avec les valeurs codées en dur dans le shellcode.

Plutôt que de choisir et de parser des DLL, j'ai estimé (peut être à tord) qu'il serait plus rapide de trouver des listes de fonctions sur internet. Je me suis donc retrouvé à parser (de façon un peu sale) des pages HTML. Je remercie au passage Geoff Chappell qui héberge sur son site [10] des listes de fonctions pratiques, puisque toutes les fonctions d'une DLL sont sur la même page.

10. https://www.geoffchappell.com/studies/windows/win32/kernel32/api/index.htm

```python
import re

def sstic_djb2(s):
    h = 0x53544943
    for x in s:
        h = (h << 5) + h + ord(x)
    return h & 0xFFFFFFFF

def get_funcs(path):
    with open(path, "rb") as f:
        x = f.read()
    funcs = re.findall(r"span [^>]*.>([^<]+)</span", x)
    if "winsock" in path:
        funcs = re.findall(r"absolute-path.>([^>]+)</a>", x)
    return funcs

hardcoded =  [0x7C0B9C64, 0xEF5DD668, 0x5211FF8D, 0x74CAE544,
  0x4C531778, 0x992D1B85, 0x556D2B5F, 0xB07290D5, 0x3789542D,
  0x1776EDEC, 0x3779904D, 0x0B4DFE8D, 0xAA090282]

for f in ["kernel32.html", "ntdll.html", "kernelbase.html", "native.html", "shell32.html", "winsock2.html"]:
    funcs = get_funcs(f)
    for s in funcs:
        h = sstic_djb2(s)
        if h in hardcoded:
            print("%08X = %s" % (h, s))
            hardcoded.remove(h)
```

```
$ python3 sstic_djb2.py
992D1B85 = CloseHandle
4C531778 = CreateFileA
5211FF8D = FindFirstFileA
74CAE544 = FindNextFileA
556D2B5F = ReadFile
B07290D5 = VirtualAlloc
EF5DD668 = SHGetFolderPathA
AA090282 = closesocket
3779904D = connect
0B4DFE8D = send
1776EDEC = socket
```

Bingo ! On va maintenant pouvoir regarder ce que le shellcode va faire de toutes ces fonctions. On remarquera que cette résolution d'import aurait pu être faite plus facilement en dynamique, mais à ce moment je n'avais pas encore fait les démarches pour être capable de débugger le shellcode dans Windows 98.

4.7 Analyse de l'exfiltration

Pour l'instant, on va donc ignorer le gros bloc AES et regarder ce qui se passe ensuite, quand les fonctions importées commencent à être utilisées.

Le shellcode commence par ouvrir une socket TCP, puis appelle SHGetFolderPathA avec l'argument `CSIDL_PERSONAL`. CSIDL signifie *Constant special item ID list*. Cette fonction permet de récupérer le chemin vers certains dossiers spéciaux. En l'occurrence, ici le shellcode va accéder au dossier `"My Documents"` de CeriseLiquide.

Ensuite, avec FindFirstFileA et FindNextFileA, le shellcode itère sur tous les fichiers contenus dans le dossier personnel de CeriseLiquide, les ouvre, les chiffre puis les envoie dans la socket.

CeriseLiquide s'est donc fait siphonner ses données dans les régles de l'art, et le trafic à sens unique qu'on observe sur le port 443 du pcap correspond aux données chiffrées qui ont été exfiltrées. Notre mission va donc être – si cela est possible – de déchiffrer le flux exfiltré afin de rentrer en possession des fichiers personnels de CeriseLiquide. Pour ce faire, on a besoin de trois choses :

— Extraire le flux chiffré à partir du pcap.
— Comprendre l'algorithme de chiffrement utilisé et l'inverser.
— Trouver la clé qui a été utilisée par cet algorithme.

4.8 Extraction du flux chiffré

C'est clairement l'étape la plus facile donc je commence par celle-ci. Gràce à Scapy, on extrait proprement du pcap le flux que le malware exfiltre. On le stocke dans un fichier, en espérant pouvoir le déchiffrer un jour.

```python
from scapy.all import *

a = rdpcap("capture.pcap")
s = a.sessions()
pkts = s["TCP 192.168.20.128:1041 > 192.168.20.1:443"]
out = bytearray()
for p in pkts:
    if "Raw" in p:
        out += p["Raw"].load
with open("exfiltrate.enc", "wb") as f:
    f.write(out)
```

4.9 Rétroconception de l'algorithme de chiffrement des fichiers

À première vue, L'algorithme qui sert à chiffrer les fichiers avant qu'ils soient exfiltrés est en trois parties. La première et la troisième se ressemblent. Elles font des opérations à partir d'une table contenue dans les données du shellcode. Cette table contient 256 entrées de 3 octets chacune. Entre ces deux parties se trouve un traitement un peu plus obscur [11].

Quand on cherche à identifier un algorithme de cryptographie, le premier réflexe est de googler ses constantes. Ici, quand on cherche la première entrée de la table sur google, à savoir `8AED2A`, on atterri sur Google Books, à la page 270 du livre du symposium STACS 98, dans un paragraphe qui parle de l'algorithme COCONUT98 et des tables qu'il utilise. La table correspond parfaitement à celle présente dans notre shellcode. On a donc très probablement identifié l'algorithme de chiffrement.

On part donc en quête de plus d'informations sur COCONUT98 et sur son implémentation de référence. Mais on constate assez rapidement que cet algorithme au nom cocasse n'est pas

11. c'est un euphémisme, puisqu'en définitive cette partie se révélera être une multiplication dans un corps de Galois

vraiment passé à la postérité... puisqu'on ne trouve aucune implémentation sur Internet. Pas même une implémentation de référence. La publication initiale qui décrit l'algorithme contient juste une description mathématique de l'algorithme [12]. Quelle belle surprise ... :/

COCONUT98 possède malgré tout un court article Wikipedia qui lui est dédié [13]. C'est un algorithme créé en 1998 [14] par Serge Vaudenay, un cryptologue français. C'est un algorithme de chiffrement par bloc qui travaille sur des blocs de 64 bits et des clés de 256 bits. Il consiste en 8 tours d'un réseau de Feistel avec une opération de décorrélation au bout de 4 tours. En pratique, cette décorrélation se traduit par une multiplication modulaire dans un corps fini GF(2^{64}) (GF signifiant *Galois Field*).

L'article Wikipédia parle de l'attaque boomerang qui a été développée en 1999 contre cet algorithme. Mais pour la mettre en oeuvre, on a besoin de *adaptively-chosen plaintexts and ciphertexts*, ce qui ne correspond pas du tout à la situation dans laquelle on est (déchiffrer un flux unique, dont le clair est inconnu). L'article pointe aussi vers la publication originale de Serge Vaudenay qui décrit COCONUT98, *Provable Security for Block Ciphers by Decorrelation*, qui date de février 1998. Cette publication est uniquement disponible au format PostScript [15]. On va donc la convertir en PDF et la lire attentivement.

Ce papier est en fait le même que celui du symposium STACS 98 trouvé sur Google Books, sauf que sur Google Books il manque plusieurs pages cruciales, notamment celle contenant les vecteurs de test. Pour une clé et un clair donné, on dispose des valeurs intermédiaires que doit renvoyer l'algorithme après chaque tour et l'étape de décorrélation. Pour le premier tour, on a également un descriptif détaillé des opérations mathématiques à effectuer. Dans le shellcode, il n'y a que le code x86 de la partie chiffrement de COCONUT98, or on souhaite pouvoir déchiffrer. Comme les fonctions de chiffrement et de déchiffrement ne sont pas identiques, on va forcément devoir réimplémenter COCONUT98. Pour ce faire, j'ai choisi Python.

Pour l'étape de préparation de la clé et les 8 tours de réseau de Feistel, c'est assez simple, il suffit de suivre les explications du papier. L'étape plus pénible est la fameuse décorrélation, puisqu'il faut calculer une multiplication dans un corps fini. Voici l'intégralité de la marche à suivre pour coder cette étape :

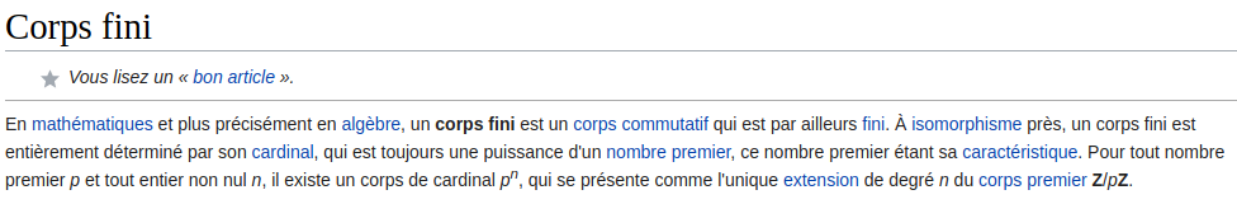
> xor it with K_3 or K_4 alternately. The decorrelation module is a function M defined by $(K_5, \ldots, K_8)$ by
>
> $$M(xy) = (xy \oplus K_5K_6) \times K_7K_8$$
>
> where the product is performed in GF(2^{64}). The Galois Field representation is chosen so that a 64-bit string $b_{63} \ldots b_1b_0$ represents the polynomial
>
> $$b_{63}.x^{63} + \ldots + b_1.x + b_0$$
>
> modulo $x^{64} + x^{11} + x^2 + x + 1$ modulo 2.

Il faut savoir que les mathématiques sont un peu ma Némésis et que je fais un blocage psychologique sur la notation mathématique. Donc traduire cette description mathématique

12. Ce qui n'a probablement pas aidé à sa démocratisation
13. https://en.wikipedia.org/wiki/COCONUT98
14. On pourra donc affirmer que pour un malware qui cible Windows 98, il s'agit d'un algorithme à l'état de l'art !
15. Mais...

en code Python me semblait hors de portée de mes modestes capacités. Puisqu'il faut d'abord comprendre ce qu'est un corps fini.

Heureusement, wikipédia est la !



Corps fini

★ *Vous lisez un « bon article ».*

En mathématiques et plus précisément en algèbre, un **corps fini** est un corps commutatif qui est par ailleurs fini. À isomorphisme près, un corps fini est entièrement déterminé par son cardinal, qui est toujours une puissance d'un nombre premier, ce nombre premier étant sa caractéristique. Pour tout nombre premier p et tout entier non nul n, il existe un corps de cardinal p^n, qui se présente comme l'unique extension de degré n du corps premier $\mathbf{Z}/p\mathbf{Z}$.

Ah... c'est tout de suite beaucoup plus clair ! Je suis vraiment content d'apprendre que je suis en train de lire un « bon article » ! Car pour comprendre cette phrase d'introduction, je n'ai plus qu'a comprendre ce qu'est un corps commutatif... un corps tout court d'ailleurs, un isomorphisme, un cardinal, une extension de degré n, un corps premier, et enfin Z/pZ.

Je suis donc allé pleurer ma détresse sur l'épaule virtuelle d'un polytechnicien, un certain L.F., qui a essayé tant bien que mal de m'inculquer quelques notions d'algèbre et de notation mathématique.

4.10 Implémentation de COCONUT98

Un fois que j'ai grossièrement compris ce qu'était un corps et un corps fini, et ce que voulait dire polynôme dans ce contexte, j'ai pu coder une implémentation de COCONUT98 en python.

Pour la représentation du corps fini, j'ai utilisé le module Python3 pyfinite [16]. Pour que ce module fonctionne bien avec GF(2^{64}), il faut préciser l'argument useLUT=0. On utilise comme générateur de notre corps fini, le nombre calculé à partir du polynôme décrit dans la publication. En effet, quand on parle de GF(2^{64}) modulo $x^{64} + x^{11} + x^2 + x + 1$ modulo 2, en pratique on va créer un corps qui contient 2^{64} élements qui sont tous des multiples de $2^{64} + 2^{11} + 2^2 + 2^1 + 1$, c'est à dire 0x10000000000000807. D'ailleurs, dans le code x86 présent dans le shellcode, on retrouve un peu partour l'utilisation des bits de poids faible de ce nombre (0x807). Une fois notre GF(2^{64}) ainsi créé dans pyfinite, on peut directement calculer la multiplication modulaire.

Pour coder la fonction de déchiffrement, on dispose d'un petit paragraphe explicatif :

We note that the decryption can be performed by using a key K^{-1} defined by

$$K^{-1} = (K_2 \oplus K_4, K_1 \oplus K_4, K_3, K_4, K'_5, K'_6, K'_7, K'_8)$$

with

$$\begin{aligned} K'_5 K'_6 &= K_5 K_6 \times K_7 K_8 \\ K'_7 K'_8 &= 1/K_7 K_8 \end{aligned}$$

in GF(2^{64}).

16. https://pypi.org/project/pyfinite

Il faut donc calculer une inversion modulaire dans GF(2^{64}), ce que, fort heureusement, pyfinite sait également faire. Voici donc mon implémentation de COCONUT98 en python3, qui passe le test vector.

```python
from pyfinite import ffield

P = 2**64 + 2**11 + 2**2 + 2**1 + 1
C = 0xb7e15162
S = [
0x8aed2a, 0x6abf71, 0x58809c, 0xf4f3c7, 0x62e716, 0x0f38b4, 0xda56a7, 0x84d904,
0x5190cf, 0xef324e, 0x773892, 0x6cfbe5, 0xf4bf8d, 0x8d8c31, 0xd763da, 0x06c80a,
0xbb1185, 0xeb4f7c, 0x7b5757, 0xf59584, 0x90cfd4, 0x7d7c19, 0xbb4215, 0x8d9554,
0xf7b46b, 0xced55c, 0x4d79fd, 0x5f24d6, 0x613c31, 0xc3839a, 0x2ddf8a, 0x9a276b,
0xcfbfa1, 0xc877c5, 0x6284da, 0xb79cd4, 0xc2b329, 0x3d20e9, 0xe5eaf0, 0x2ac60a,
0xcc93ed, 0x874422, 0xa52ecb, 0x238fee, 0xe5ab6a, 0xdd835f, 0xd1a075, 0x3d0a8f,
0x78e537, 0xd2b95b, 0xb79d8d, 0xcaec64, 0x2c1e9f, 0x23b829, 0xb5c278, 0x0bf387,
0x37df8b, 0xb300d0, 0x1334a0, 0xd0bd86, 0x45cbfa, 0x73a616, 0x0ffe39, 0x3c48cb,
0xbbca06, 0x0f0ff8, 0xec6d31, 0xbeb5cc, 0xeed7f2, 0xf0bb08, 0x801716, 0x3bc60d,
0xf45a0e, 0xcb1bcd, 0x289b06, 0xcbbfea, 0x21ad08, 0xe1847f, 0x3f7378, 0xd56ced,
0x94640d, 0x6ef0d3, 0xd37be6, 0x7008e1, 0x86d1bf, 0x275b9b, 0x241deb, 0x64749a,
0x47dfdf, 0xb96632, 0xc3eb06, 0x1b6472, 0xbbf84c, 0x26144e, 0x49c2d0, 0x4c324e,
0xf10de5, 0x13d3f5, 0x114b8b, 0x5d374d, 0x93cb88, 0x79c7d5, 0x2ffd72, 0xba0aae,
0x7277da, 0x7ba1b4, 0xaf1488, 0xd8e836, 0xaf1486, 0x5e6c37, 0xab6876, 0xfe690b,
0x571121, 0x382af3, 0x41afe9, 0x4f77bc, 0xf06c83, 0xb8ff56, 0x75f097, 0x9074ad,
0x9a787b, 0xc5b9bd, 0x4b0c59, 0x37d3ed, 0xe4c3a7, 0x939621, 0x5edab1, 0xf57d0b,
0x5a7db4, 0x61dd8f, 0x3c7554, 0x0d0012, 0x1fd56e, 0x95f8c7, 0x31e9c4, 0xd7221b,
0xbed0c6, 0x2bb5a8, 0x7804b6, 0x79a0ca, 0xa41d80, 0x2a4604, 0xc311b7, 0x1de3e5,
0xc6b400, 0xe024a6, 0x668ccf, 0x2e2de8, 0x6876e4, 0xf5c500, 0x00f0a9, 0x3b3aa7,
0xe6342b, 0x302a0a, 0x47373b, 0x25f73e, 0x3b26d5, 0x69fe22, 0x91ad36, 0xd6a147,
0xd1060b, 0x871a28, 0x01f978, 0x376408, 0x2ff592, 0xd9140d, 0xb1e939, 0x9df4b0,
0xe14ca8, 0xe88ee9, 0x110b2b, 0xd4fa98, 0xeed150, 0xca6dd8, 0x932245, 0xef7592,
0xc703f5, 0x32ce3a, 0x30cd31, 0xc070eb, 0x36b419, 0x5ff33f, 0xb1c66c, 0x7d70f9,
0x391810, 0x7ce205, 0x1fed33, 0xf6d1de, 0x9491c7, 0xdea6a5, 0xa442e1, 0x54c8bb,
0x6d8d03, 0x62803b, 0xc248d4, 0x14478c, 0x2afb07, 0xffe78e, 0x89b9fe, 0xca7e30,
0x60c08f, 0x0d61f8, 0xe36801, 0xdf66d1, 0xd8f939, 0x2e52ca, 0xef0653, 0x199479,
0xdf2be6, 0x4bbaab, 0x008ca8, 0xa06fda, 0xce9ce7, 0x048984, 0x5a082b, 0xa36d61,
0x1e99f2, 0xfbe724, 0x246d18, 0xb54e33, 0x5cac0d, 0xd1ab9d, 0xfd7988, 0xa4b0c4,
0x558aa1, 0x194177, 0x20b6e1, 0x50ce2b, 0x927d48, 0xd7256e, 0x445e33, 0x3cb757,
0x2b3bd0, 0x0fb274, 0x604318, 0x9cac11, 0x6cedc7, 0xe771ae, 0x0358ff, 0x752a3a,
0x6b6c79, 0xa58a9a, 0x549b50, 0xc58706, 0x90755c, 0x35e4e3, 0x6b5290, 0x38ca73,
0x3fd1aa, 0xa8dab4, 0x0133d8, 0x0320e0, 0x790968, 0xc76546, 0xb993f6, 0xc8ff3b]

def ror(x, n, bits = 32):
    mask = (2**n) - 1
    mask_bits = x & mask
    return (x >> n) | (mask_bits << (bits - n))

def rol(x, n, bits = 32):
    return ror(x, bits - n, bits)

def phi(x):
    return ((S[x & 0xFF] << 8) + x) & 0xFFFFFFFF

def round(x, y, k):
    a = y ^ k
    a = phi(a)
    a = rol(a, 11)
    a = (a + C) & 0xFFFFFFFF
    a = phi(a)
    a = a ^ x
    return y, a

def decorrelation(x, y, key):
    a = x ^ key[4]
    b = y ^ key[5]
    c = (a << 32) | b
    d = (key[6] << 32) | key[7]
    f = ffield.FField(64, gen=P, useLUT=0)
    r = f.Multiply(c, d)
    x = (r >> 32)
    y = r & 0xFFFFFFFF
```

```
68      return x, y
69
70  def encrypt(plaintext, key):
71      x, y = plaintext >> 32, plaintext & 0xFFFFFFFF
72      k = key[0]
73      x, y = round(x, y, k)
74      k ^= key[2]
75      x, y = round(x, y, k)
76      k ^= key[3]
77      x, y = round(x, y, k)
78      k ^= key[2]
79      y, x = round(x, y, k)
80      x, y = decorrelation(x, y, key)
81      k = key[1]
82      x, y = round(x, y, k)
83      k ^= key[2]
84      x, y = round(x, y, k)
85      k ^= key[3]
86      x, y = round(x, y, k)
87      k ^= key[2]
88      y, x = round(x, y, k)
89      return (x << 32) | y
90
91  def decrypt(ciphertext, key):
92      a = (key[4] << 32) | key[5]
93      b = (key[6] << 32) | key[7]
94      f = ffield.FField(64, gen=P, useLUT=0)
95      r = f.Multiply(a, b)
96      c = f.Inverse(b)
97      invkey = [
98          key[1] ^ key[3],
99          key[0] ^ key[3],
100         key[2],
101         key[3],
102         r >> 32,
103         r & 0xFFFFFFFF,
104         c >> 32,
105         c & 0xFFFFFFFF]
106     return encrypt(ciphertext, invkey)
107
108 if __name__ == "__main__":
109     # run test vector
110     K = [0x7c44a4ad, 0x56f6bb77, 0x220e96e8, 0x401694e1,
111          0x6c469dbc, 0x516decc5, 0x17929e9b, 0x226ddd64]
112     plaintext = 0x6d8779e078ac5f02
113     ciphertext = 0x3b2ae895037910f8
114     assert(ciphertext == encrypt(plaintext, K))
115     assert(plaintext == decrypt(ciphertext, K))
116     print("Test vector passed !")
```

Maintenant, on a envie d'exécuter le chiffrement COCONUT98 provenant du shellcode pour s'assurer qu'on obtient bien le même résultat qu'avec notre implémentation. Il serait vraiment fourbe que l'attaquant utilise une version customisée de COCONUT98...

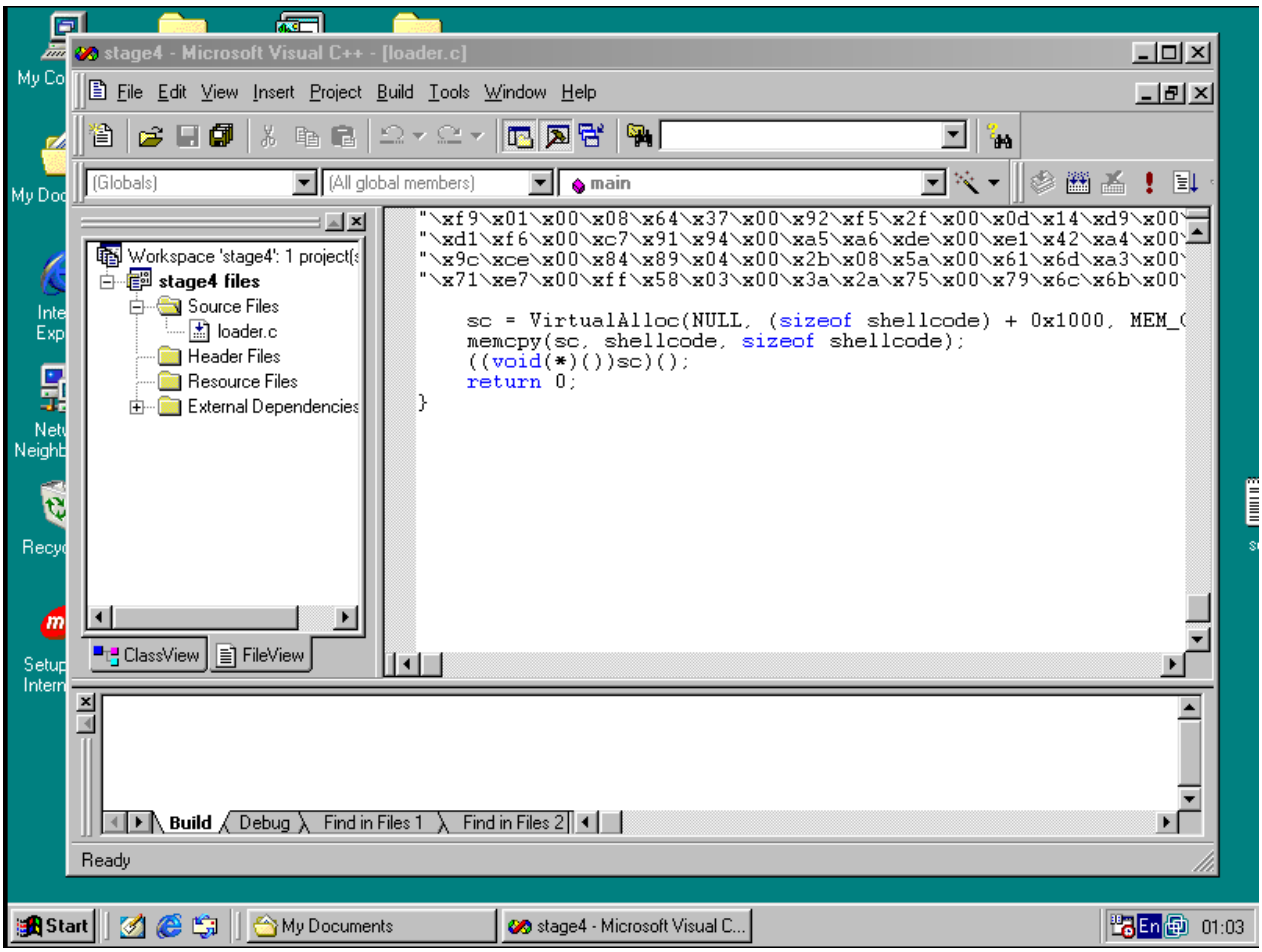
Le code x86 de COCONUT98 du shellcode n'utilise rien de spécifique à Windows 98 (rien qui dépende du PEB), on pourrait donc imaginer l'exécuter sur un Windows 10 ou l'émuler. Mais je me suis dit que pour trouver la bonne clé, j'aurais de toute façon besoin de débugger le shellcode complet directement dans un vrai Windows 98. Donc autant faire le nécessaire pour vérifier par la même occasion COCONUT98 dans Windows 98.

4.11 Débugger le shellcode dans Windows 98

J'ai testé plusieurs possibilités pour pouvoir débugger le shellcode dans ma machine virtuelle Windows 98, dont notamment essayer d'utiliser IceBox (et donc réinstaller Windows 98 dans un Virtualbox patché, plutôt que dans VMware), mais ça n'a pas fonctionné.

Finalement, le plus confortable [17] a été d'installer Visual Studio 6 pour compiler un loader de shellcode directement dans la VM, et ensuite d'utiliser le débugger intégré à vs6.

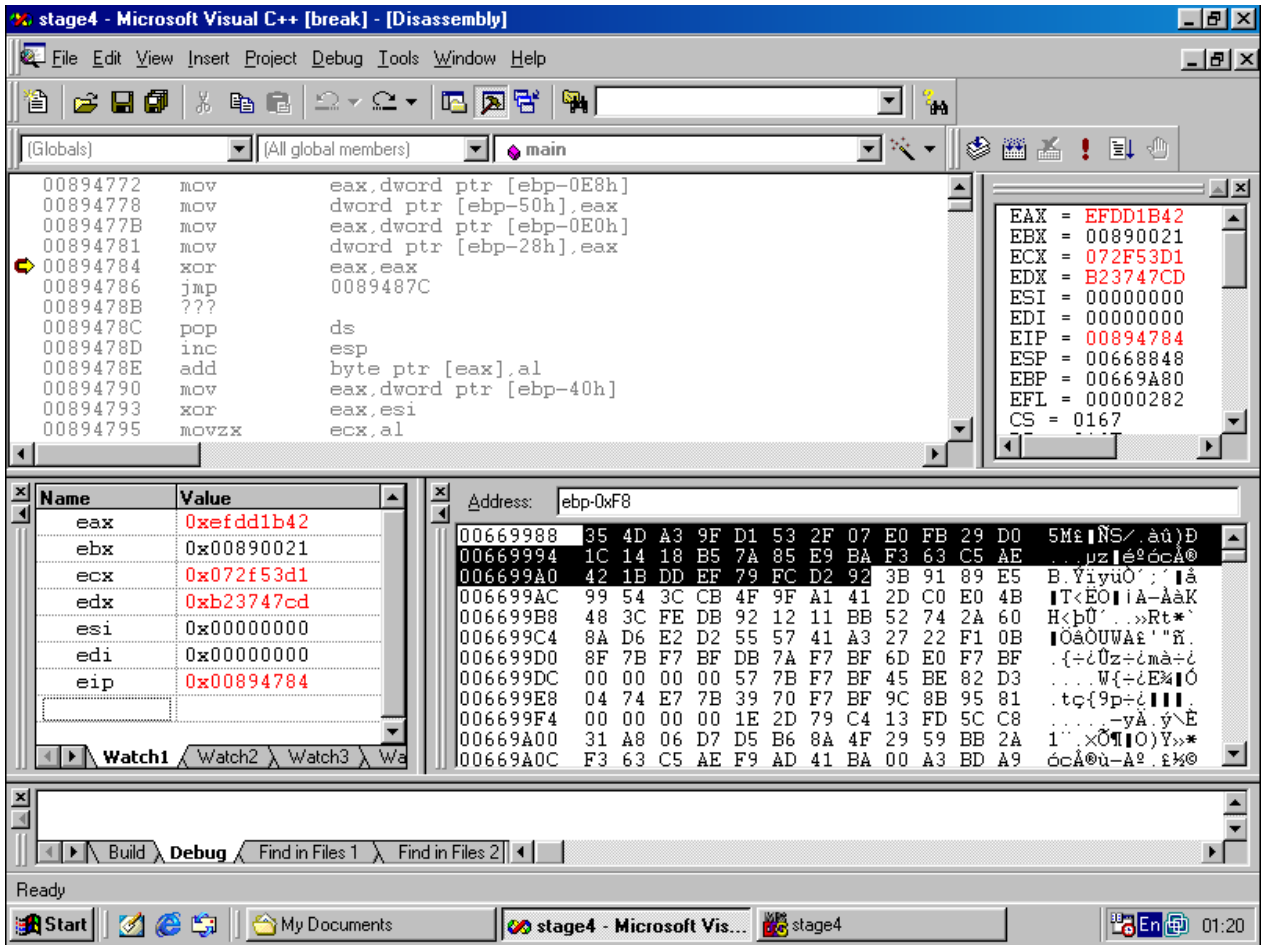
On écrit donc un petit loader en C, dans un IDE que j'ai trouvé étonnement évolué pour l'époque. En 1998, Visual Studio avait déjà beaucoup de fonctionnalités avancées [18].



Le loader consiste simplement en un VirtualAlloc() d'une zone RWX dans laquelle on memcpy() le shellcode, avant d'y sauter. On peut alors entrer dans la vue désassemblée du mode debug et examiner le shellcode en dynamique.

Quand on met un point d'arrêt au début du code COCONUT98, on peut lire la clé de 32 octets qu'il utilise pour chiffrer les fichiers.

17. tout est relatif
18. ce n'est pas de l'ironie



En exécutant le code pas à pas, on constate qu'il s'agit bien d'une implémentation *standard* de COCONUT98 puisque notre implémentation Python en utilisant cette clé donne les même résultats, il faut juste faire attention à l'endianness. En effet, la clé ci-dessous va être lue en little-endian.

```
35 4D A3 9F D1 53 2F 07
E0 FB 29 D0 1C 14 18 B5
7A 85 E9 BA F3 63 C5 AE
42 1B DD EF 79 FC D2 92
```

4.12 Trouver la bonne clé

Maintenant qu'on a récupéré le flux chiffré et recodé l'algorithme permettant de le déchiffrer, il ne manque plus qu'à trouver la bonne clé. Et c'est là qu'on a une très bonne surprise (pour une fois), puisque la clé qu'on trouve en dynamique est constante entre chaque exécution, et qu'il s'agit directement de la bonne clé.

Dans le shellcode, l'algorithme COCONUT98 est utilisé en mode CBC avec un IV nul. Voici donc le script permettant de déchiffrer le flux exfiltré.

```python
import struct
from coconut98 import decrypt

key = [0x9fa34d35, 0x72f53d1, 0xd029fbe0, 0xb518141c, 0xbae9857a, 0xaec563f3, 0xefdd1b42, 0x92d2fc79]

with open("exfiltrate.enc", "rb") as f:
    enc = f.read()

with open("exfiltrate.dec", "wb") as f:
    iv = 0
    for i in range(0, len(enc), 8):
        c = struct.unpack("<Q", enc[i:i+8])[0]
        r = decrypt(c, key)
        r = r ^ iv
        iv = c
        f.write(struct.pack("<Q", r))
```

On trouve le flag de validation du quatrième niveau dans le flux déchiffré.

```
$ strings exfiltrate.dec | grep -i sstic
[...]
SSTIC{e5a35d6b392b2bbe4af78614076b0fbb03f33aa5dd5bf92fd247a7d2a00a5247}
```

4.13 Mais d'où provient cette clé ?

Maintenant qu'on a la capacité d'analyser dynamiquement le shellcode, on peut plus facilement comprendre d'où provient la clé COCONUT98 utilisée pour chiffrer les fichiers de CeriseLiquide, et qui semble codée en dur.

Cette clé est la sortie du gros bloc de dérivation de clé AES. En dynamique, on constate que l'entrée de cette dérivation de clé AES sont les 9 premiers octets de la chaîne de caractères « *A20 hardware error. Contact technical support to identify the problem* ».

Mais la question qui en découle, c'est où le shellcode va t-il chercher cette chaîne ? C'est là que l'astuce des `far call` intervient. Juste avant la dérivationde clé AES, le shellcode fait un appel à VirtualAlloc, puis exécute deux `far call` qui vont atterrir dans un handler présent à la fin du shellcode.

C'est une sorte de mécanisme de syscall. Le handler gère 3 syscalls différents. Le syscall 2 retourne la valeur contenue dans le registre CR3, mais n'est a priori pas appelé. Les syscalls 0 et 1 prennent en paramètre l'adresse du buffer précédemment alloué avec VirtualAlloc. Ils font de sombres manipulations avec les plages mémoire 0xFF800000 et 0xFFBFE000, avant de flusher l'entrée de la TLB correspondant à la page de notre buffer alloué.

Par ces petits tours de magie, le buffer alloué (vide) pointe désormais vers un buffer système. Il contient toute sortes de données propres à Windows 98. On y retrouve quelques chaînes de caractères, dont le nom local de la machine et les noms de ports COM. Quand j'ai vu la fameuse *Erreur hardware A20* dans ce buffer, j'ai d'abord pensé qu'il s'agissait d'une défaillance liée à la VM qui ne supportait pas certaines opérations bas niveau. Mais non, c'est

bien cette chaîne que le shellcode va venir chercher, va dériver avec AES et l'utiliser comme clé de chiffrement COCONUT98.

En pratique, toute la partie syscall et AES semble exister dans le seul but de nous empêcher de résoudre l'épreuve de manière statique, de nous forcer à installer une VM Windows 98 et de trouver un moyen de débugger le shellcode à l'intérieur. Ce qui a plutôt bien fonctionné !

Voilà qui conclut la quatrième épreuve de ce challenge, qui était nettement plus longue et plus compliquée que les trois précédentes.

`SSTIC{e5a35d6b392b2bbe4af78614076b0fbb03f33aa5dd5bf92fd247a7d2a00a5247}`

5 Niveau 5

5.1 Récupération des fichiers de CeriseLiquide

Dans le niveau 4, la machine de CeriseLiquide, qui tourne sous un vieux Windows 98, a subi une RCE qui a mené à l'exfiltration de ses fichiers personnels contenus dans son dossier `"My Documents"`. Le flux a pu être déchiffré grâce à l'algorithme COCONUT98. Il faut désormais le parser correctement pour reconstituer les fichiers d'origine.

Dans le flux, chaque fichier est précédé de sa structure `WIN32_FIND_DATAA` d'origine, qui contient toutes les métadonnées nécessaires à leur reconstruction (le nom du fichier, sa taille, les dates de création, d'accès et d'écriture, ...).

On écrit donc un script qui parse les structures `WIN32_FIND_DATAA` et qui extrait les fichiers du flux.

```python
import struct
import os
from datetime import datetime, timedelta

EPOCH_AS_FILETIME = 116444736000000000
OUTDIR = "CeriseLiquide"

if not os.path.exists(OUTDIR):
    os.mkdir(OUTDIR)

with open("exfiltrate.dec", "rb") as f:
    stream = f.read()

def filetime_to_dt(ft):
    us = (ft - EPOCH_AS_FILETIME) / 10
    return datetime(1970, 1, 1) + timedelta(microseconds = us)

def get_qword(i, le=True):
    data_hi = struct.unpack("<I", stream[i:i+4])[0]
    data_lo = struct.unpack("<I", stream[i+4:i+8])[0]
    if le:
        data_hi, data_lo = data_lo, data_hi
    data = (data_hi << 32) + data_lo
    return data

def parse_file(idx):
    filename = stream[idx+0x2C:idx+0x130]
```

```
28      filename = filename.split(b"\x00", 1)[0].decode()
29      size = get_qword(idx+0x1C, False)
30      t_create = get_qword(idx+4)
31      t_access = get_qword(idx+12)
32      t_write = get_qword(idx+20)
33      if filename:
34          print('File "%s" (%i bytes)' % (filename, size))
35          print("  creation time   : %s" % filetime_to_dt(t_create))
36          print("  last write time : %s" % filetime_to_dt(t_write))
37          print("  last access time: %s" % filetime_to_dt(t_access))
38          with open(os.path.join(OUTDIR, filename), "wb") as f:
39              f.write(stream[idx+0x140:idx+0x140+size])
40      return size + 0x140
41
42  idx = 0
43  while idx < len(stream):
44      idx += parse_file(idx)
45      # align on 8 bytes
46      if idx % 8:
47          idx += (8 - idx % 8)
```

Je m'étais embêté à parser les dates de création, d'accès et de dernière écriture des fichiers car j'avais le pressentiment que ça allait aider pour la suite (en particulier pour casser la base Keepass). Mais en fait non :)

```
$ python3 parse_files.py
File "desktop.ini" (125 bytes)
  creation time   : 2019-11-02 07:02:35.900000
  last write time : 2019-11-02 07:02:36
  last access time: 2020-03-29 08:00:00
File "screensaver.iso" (2488416 bytes)
  creation time   : 2020-03-24 13:36:23.280000
  last write time : 2020-03-29 19:51:36
  last access time: 2020-03-29 08:00:00
File "KeyProvider.dll" (9105 bytes)
  creation time   : 2020-03-24 13:36:27.120000
  last write time : 2020-03-29 19:51:36
  last access time: 2020-03-29 08:00:00
File "wallpaper.jpg" (69149 bytes)
  creation time   : 2020-03-24 13:36:27.590000
  last write time : 2020-03-29 19:51:38
  last access time: 2020-03-29 08:00:00
File "KeePass.exe" (719872 bytes)
  creation time   : 2020-03-24 13:36:28.120000
  last write time : 2020-03-29 19:51:40
  last access time: 2020-03-29 08:00:00
File "KeePass.ini" (92 bytes)
  creation time   : 2020-03-24 13:36:30.740000
  last write time : 2020-03-29 19:51:40
  last access time: 2020-03-29 08:00:00
File "Database.kdb" (1452 bytes)
  creation time   : 2020-03-29 19:51:57.850000
  last write time : 2020-03-29 19:51:58
  last access time: 2020-03-29 08:00:00
File "flag.txt" (72 bytes)
  creation time   : 2020-03-28 15:58:34.560000
  last write time : 2019-03-29 19:51:40
  last access time: 2020-03-29 08:00:00
```

Nous obtenons 8 fichiers à analyser :

Keepass.exe une vieille version du célèbre gestionnaire de mots de passe.

KeyProvider.dll , un plugin Keepass chargé, comme son nom l'indique, de fournir une clé maître à Keepass.exe.

Keepass.ini un fichier de configuration Keepass qui sert juste à charger le plugin KeyProvider.dll.

Database.kdb une base de donnée chiffrée Keepass qui contient les mots de passe de CeriseLiquide.

wallpaper.jpg un très beau fond d'écran Lobster Dog, certifié running gag officiel du challenge SSTIC depuis 2011.

screensaver.iso un fichier non-identifié, qui n'est pas reconnu comme une ISO par la commande `file`. Initialement, vu son nom, je l'ai délaissé à tord en pensant qu'il s'agirait d'un écran de veille Lobster Dog conçu pour Windows 98, mais que pour une raison obscure, il ne fonctionnait pas. Après avoir examiné plus en détail les fichiers Keepass, j'ai compris qu'il manquait un morceau pour résoudre le challenge et que ce fichier était capital. Car en réalité, il s'agit d'une image CD pour Playstation 1 !

desktop.ini le fichier caché (inutile pour nous) qui est présent par défaut dans le dossier personnel de CeriseLiquide.

flag.txt le flag de validation du 4ème niveau.

5.2 Analyse de Keepass et de son plugin

CeriseLiquide utilise Keepass en version 1.10, une version qui date de janvier 2008 [19]. D'après les changelog, la version 1.10 est la toute première à supporter des plugins de type KeyProvider, c'est donc pour cette raison qu'elle a été choisie. À noter que Keepass 1.10 fonctionne aussi bien sous Windows 98 que sous Windows 10.

J'ai commencé par récupérer Keepass 1.10 par mes propres moyens [20] pour vérifier que le Keepass de CeriseLiquide n'était pas, par exemple, une version backdoorée. Mais non, les deux fichiers sont identiques.

J'ai ensuite fait la rétroconception du plugin KeyProvider.dll. Pour nous aider, on peut s'appuyer sur une documentation assez complète [21] qui détaille les structures manipulées par le plugin. Un plugin de type KeyProvider est en charge de fournir le mot de passe maître d'une base Keepass par un moyen alternatif. En l'occurrence, ce plugin va récupérer ce mot de passe depuis le port série COM1 de la machine de CeriseLiquide. Il ouvre le port (vitesse 115200 bauds), envoie le caractère « **?** », puis reçoit en réponse un mot de passe de 20 octets. Mais à quoi peut bien être raccordé le port série du PC de CeriseLiquide ?

Comme j'avais négligé le fichier `screensaver.iso`, j'ai pensé qu'il devait y avoir un bug critique connu dans cette vieille version de Keepass, qui pourrait aboutir à un mauvais chiffrement de la base en cas d'utilisation d'un plugin KeyProvider. Par exemple on pourrait imaginer qu'un mot de passe récupéré par ce moyen sera mal dérivé et donc plus facilement bruteforçable. Ou plus simplement, un plugin mal codé qui ferait que le mot de passe renvoyé par le KeyProvider soit toujours 0x00.

En parallèle, j'ai également utilisé `keepass2john` pour tenter un bruteforce de la base. Mais non, le plugin semble bien codé. Il n'y a pas de bug critique identifié dans cette version

19. CerisieLiquide nous avait habitué à moins de modernité !
20. https://sourceforge.net/projects/keepass/files/KeePass%201.x/1.10/
21. https://keepass.info/help/v1_dev/plg_keyprov.html

de Keepass. Et le bruteforce ne donne rien. On est donc dans une impasse et on regarde d'un oeil plus attentif les autres fichiers exfiltrés.

5.3 Analyse du Screensaver

Le format du fichier Screensaver.iso n'est pas reconnu par ma distribution Linux. Mais en l'explorant avec `strings` et `hexdump`, on trouve des indices permettant de l'identifier.

```
$ strings screensaver.iso
[...]
BOOT=cdrom:\MAIN.EXE;1
TCB=4
EVENT=10
STACK=801FFFF0
,hv~
PS-X EXE
[...]
```

PS-X EXE s'avère être le magic présent au début des exécutables Playstation 1. Nous sommes en présence d'un CD de Playstation.

Pour récupérer le fichier MAIN.EXE présent sur le CD, j'ai utilisé le logiciel CDmage qui supporte les ISO de playstation. C'est un exécutable au format PS-X EXE, pour architecture MIPS32 little endian.

Après avoir chargé MAIN.EXE dans Ghidra, j'ai changé mon fusil d'épaule et je suis plutôt parti en quête d'un émulateur PSX orienté debug. Je suis alors rapidement tombé sur `No$psx`. La documentation relative au projet `No$psx` est impressionnante tant elle est détaillée. Sur la page http://problemkaputt.de/psx-spx.htm on trouve toutes les spécifications de la Playstation. Une vraie mine d'or.

Quand on charge le fichier `screensaver.iso` dans l'émulateur `No$psx`, on obtient un écran noir avec des étoiles qui défilent. Un économisateur d'écran qui n'est pas sans rappeler ce qu'on pouvait trouver sur Windows 98 (Flying Windows, mais avec des points à la place des logo Windows).

5.4 Trouver le port série

Ma première intuition est de regarder si le jeu PSX contient du code relatif à la manipulation du port série de la Playstation. Si tel est le cas, alors on peut imaginer que la Playstation est connectée via une liaison série à l'ordinateur de CeriseLiquide, et que c'est l'économisateur d'écran qui va répondre à la demande de clé du plugin Keepass. Sachant que le plugin Keepass se contente d'envoyer le caractère **?** puis de recevoir 20 octets, il suffit potentiellement d'envoyer un point d'interrogation sur le port série de la Playstation pour récupérer le mot de passe Keepass.

Grâce aux spécifications incroyables du site problemkaputt.de, dans la section I/O Map, on trouve les adresses qui correspondent aux interaction avec le périphérique série de la Playstation.

```
  1F801050h 1/4  SIO_DATA Serial Port Data (R/W)
  1F801054h 4    SIO_STAT Serial Port Status (R)
  1F801058h 2    SIO_MODE Serial Port Mode (R/W)
  1F80105Ah 2    SIO_CTRL Serial Port Control (R/W)
  1F80105Ch 2    SIO_MISC Serial Port Internal Register (R/W)
  1F80105Eh 2    SIO_BAUD Serial Port Baudrate (R/W)
```

Dans le debugger `No$psx`, j'ai généré un dump texte du code désassemblé, dans lequel j'ai cherché des utilisations de ces adresses. Et bingo, on trouve notre bonheur à l'adresse 0x8001C7D8.

```
8001C7D8 1F801050
```

Et on trouve ensuite de multiples références croisées sur 0x8001C7D8 qui tombent dans du code qui correspond à nos attentes. Cette fois j'ai donc généré un dump binaire de la mémoire de l'émulateur entre les adresses 0x80010000 et 0x80020000, que j'ai chargé dans Ghidra. On tombe ainsi directement dans la fonction 0x8001a4c4 qui s'occupe de toutes les manipulations liées au port série (initialisation, configuration, envoi et réception de données). On renomme cette fonction `serial_port_stuff` et on regarde qui l'appelle. On tombe alors sur du code décompilé qui nous fait très plaisir.

```
if (((uVar4 & 2) != 0) && (uVar4 = serial_port_stuff(0,4,0), (uVar4 & 0xff) == 0x3f)) {
  uVar4 = 0;
  do {
    do {
      uVar5 = serial_port_stuff(0,0,0);
    } while ((uVar5 & 1) == 0);
    serial_port_stuff(1,4,(uint)*(byte *)((int)&local_220 + uVar4));
    uVar4 = uVar4 + 1;
  } while (uVar4 < 0x14);
}
```

Ici, le jeu Playstation va lire des données sur son port série et les comparer avec 0x3f, le code ASCII du caractère **?**. Si c'est bon, il répond sur le port série par 20 octets qui proviennent d'une variable locale. Ce code est clairement conçu pour discuter avec le plugin KeyProvider de Keepass.

Cependant, au départ, la variable locale censée contenir le mot de passe Keepass ne

continent pas grand-chose... jusqu'à ce qu'on commence à appuyer sur des boutons de notre manette Playstation virtuelle. À chaque appui, la zone se voit attribuer 20 octets qui semblent uniformément distribués.

On a donc une assez bonne vision de la manière *disruptive* utilisée par CeriseLiquide pour déverrouiller sa base Keepass.

1. Il démarre sa playstation 1, avec le CD screensaver.iso à l'intérieur.
2. Il tape une combinaison de boutons sur sa manette Playstation.
3. Il ouvre Keepass et choisit d'utiliser le plugin KeyProvider.
4. Le plugin envoie un point d'interrogation sur le port série COM1.
5. Ce port série est branché à celui de la Playsation, qui va traiter la demande et répondre avec 20 octets.
6. Si (et seulement si) la combinaison de boutons était bonne, la base Database.kdb va être déverrouillée.

Il reste encore à faire de la rétroconception pour comprendre comment le mot de passe est calculé en fonction des boutons appuyées, mais au moins nous sommes rassurés puisque, grâce à l'adresse du port série, nous avons trouvé facilement où se situe le code intéressant du jeu. Et comme il contient beaucoup de code mais aucune chaîne de caractères pour se repérer, une de mes craintes était de perdre beaucoup de temps au milieu du code qui gère les graphismes.

5.5 Documentation des syscalls

Dans Ghidra, on constate de nombreux sauts vers des adresses très hautes (0xA0 par exemple). Grâce à la section, *BIOS Function Summary*, on comprend qu'il s'agit de syscall vers le BIOS de la PSX.

Il y a trois groupes de syscall, les A-Functions, les B-Functions et les C-Functions, pour lesquelles on jump respectivement à l'adresse 0xA0, 0xB0 ou 0xC0 avec le numéro de la fonction à appeler dans le registre R9 (le registre t1 dans Ghidra, puisque la PSX utilise son propre nommage des registres MIPS32).

On documente ainsi dans Ghidra les appels à des fonctions classiques d'une libc, telles que strlen, memcpy, memcmp, rand, srand ou calloc. Mais aussi des fonctions spécifiques à la Playstation, comme InitPad, StartPad, OutdatedPadInitAndStart ou OutdatedPadGetButton qui s'occupent de gérer les manettes de Playstation.

La documentation détaillée des fonctions OutdatedPadInitAndStart et OutdatedPadGetButton sur le site de `No$psx`, vaut vraiment le coup d'œil ! C'est tellement alambiqué qu'on comprend mieux le choix du préfixe *Outdated*.

B(15h) - OutdatedPadInitAndStart(type, button_dest, unused, unused)
This is an extremely bizarre and restrictive function - don't use! The function fails unless type is 20000000h or 20000001h (the type value has no other function). The function uses "buf1/buf2" addresses that are located somewhere "hidden" within the BIOS variables region, the only way to read from that internal buffers is to use the ugly "OutdatedPadGetButtons()" function. For some strange reason it FFh-fills buf1/buf2, and does then call InitPad(buf1,22h,buf2,22) (which does immediately 00h-fill the previously FFh-filled buffers), and does then call StartPad().
Finally, it does memorize the "button_dest" address (see OutdatedPadGetButtons() for details on that value). The two unused parameters have no function, however, they are internally written back to the stack locations reserved for parameter 2 and 3, ie. at [SP+08h] and [SP+0Ch] on the caller's stack, so the function MUST be called with all four parameters allocated on stack. Return value is 2 (or 0 if type was disliked).

B(16h) - OutdatedPadGetButtons()
This is a very ugly function, using the internal "buf1/buf2" values from "OutdatedPadInitAndStart" and the "button_dest" value that was passed to that function.
If "button_dest" is non-zero, then this function is automatically called by the PadCardIrq handler, and stores it's return value at [button_dest] (where it may be read by the main program). If "button_dest" is zero, then it isn't called automatically, and it <can> be called manually (with return value in R2), however, it does additionally write the return value to [button_dest], ie. to [00000000h] in this case, destroying that memory location.
The return value itself is useless garbage: The lower 16bit contain the pad1 buttons, the upper 16bit the pad2 buttons, of which, both values have reversed byte-order (ie. the first button byte in upper 8bit). The function works only with controller IDs 41h (digital joypad) and 23h (nonstandard device). For ID=23h, the halfword is ORed with 07C7h, and bit6,7 are then cleared if the analogue inputs are greater than 10h. For ID=41h the data is left intact. Any other ID values, or disconnected joypads, cause the halfword to be set to FFFFh (same as when no buttons are pressed), that includes newer analogue pads (unless they are switched to "digital" mode).

5.6 Rétroconception de la boucle principale

A l'adresse 0x80012de se trouve la fonction principale, dans laquelle une boucle infinie va effectuer tous les traitments qui nous intéressent. Le code se décompile bien dans Ghidra, mais comme la fonction est très longue et que beaucoup de code a été inliné, elle reste malgré tout assez difficile à analyser.

On peut découper cette boucle infinie en quatre morceaux :

serial le code qui regarde si on a reçu un **?** sur le port série, auquel cas il répond avec l'état actuel du mot de passe.

graphic du code avec beaucoup d'appels à rand(), on suppose fortement qu'il est en charge de faire défiler aléatoirement les étoiles de l'économisateur d'écran, on ignore donc ce morceau de la boucle.

pad du code qui récupère les boutons appuyés sur la manette de Playstation et qui va remplir un buffer circulaire de 8 octets. Il n'y a que 4 boutons qui sont acceptés : la croix, le carré, le rond et le triangle. Les combinaisons de deux boutons (croix+rond) sont ignorées. En fonction du bouton appuyé, le buffer circulaire de 8 octets va être alimenté par la valeur **z**, **x**, **s** ou **d**.

crypto du code qui va dériver un mot de passe de 20 octet à partir du buffer circulaire de 8 octets qui contient les boutons.

Avec ces informations en tête, on remarque que le bruteforce devient possible. Car bien que le mot de passe final fasse 20 octets, il est généré à partir d'une combinaison de 8 appuis de boutons, parmi 4 boutons possibles. Il n'y a donc que $4^8 = 65536$ mots de passe possibles. Mais encore faut-il être capable de les calculer.

5.7 Rétroconception superficielle de la crypto

L'algorithme qui dérive les boutons en mot de passe a la structure d'un HMAC. On reconnaît les buffers IPAD et OPAD remplis de 0x36 et de 0x5C qui vont être xorés avec une clé.

La clé du HMAC est composée de 43 octets qui ne sont clairement pas uniformément distribués :

```
97 8b 8b 8f 8c c5 d0 d0 88 88 88 d1 86 90 8a 8b
8a 9d 9a d1 9c 90 92 d0 88 9e 8b 9c 97 c0 89 c2
bc 94 89 aa be 93 c8 a6 cb 92 88 00 00 00 00 00
```

En effet, si on applique un NOT sur chaque octet de la clé, on obtient un lien youtube vers la chanson *Toujours là pour toi*[22] des 2be3. CeriseLiquide a décidément des goûts... surprenants :)

```
https://www.youtube.com/watch?v=CkvUAl7Y4mw
```

Enfin, l'élément principale d'un HMAC, c'est une fonction de hachage. Ici, d'après les constantes d'initialisation et la taille de la sortie (20 octets), on croit tout d'abord reconnaître SHA-1.

Mais le HMAC-SHA1 ne donne pas les mêmes résultats que ce qu'on observe en émulation. Le problème est que le code étant très inliné, il n'est pas si facile à analyser et on soupçonne que la construction du HMAC puisse ne pas être standard pour nous piéger.

Plutôt que de reverser scrupuleusement cette partie du code, j'ai donc tenté d'être pragmatique et j'ai utilisé l'émulateur intégré à Ghidra pour exécuter 65536 fois ce HMAC (potentiellement *maison*) avec toutes les entrées possibles.

5.8 Génération des mots de passe

C'était la première fois que j'utilisais l'émulateur de Ghidra, et j'ai été assez impressionné par sa facilité de prise en main et par ses performances. Dans la mesure où il y a quand même beaucoup de code cryptographique à émuler, on pouvait craindre le passage à l'échelle puisqu'il va falloir émuler 65536 fois le code du HMAC. Mais non, en pratique, la génération n'a pris « que » 3 heures, dans une VM, sur un PC portable standard.

Pour faire fonctionner l'émulation dans Ghidra, on va avoir besoin de répondre à la place du BIOS lors des appels des fonctions memset et memcpy. Voici le code du plugin Ghidra qui génère les 65536 mots de passe.

```python
#HMAC emulation of screensaver.iso
#@category SSTIC

from ghidra.app.emulator import EmulatorHelper
import itertools

KEY = "\x97\x8b\x8b\x8f\x8c\xc5\xd0\xd0\x88\x88\x88\xd1\x86\x90\x8a\x8b\x8a\x9d\x9a\xd1\x9c\x90\x92\xd0" +
"\x88\x9e\x8b\x9c\x97\xc0\x89\xc2\xbc\x94\x89\xaa\xbe\x93\xc8\xa6\xcb\x92\x88"

def get_addr(offset):
    return currentProgram.getAddressFactory().getDefaultAddressSpace().getAddress(offset)

def syscall(emu, num):
    a0, a1, a2 = (emu.readRegister(x) for x in ["a0", "a1", "a2"])
    if num == 0x2a:
        # memcpy
        src = emu.readMemory(get_addr(a1), a2)
        emu.writeMemory(get_addr(a0), src.tostring())
    elif num == 0x2b:
```

22. Probablement la version française, compatible ANSSI, du rick roll *Never gonna Give You Up*

```
        # memset
        emu.writeMemory(get_addr(a0), chr(a1)*a2)
    else:
        raise Exception("unknown syscall 0x%x" % num)
    emu.writeRegister(emu.getPCRegister(), emu.readRegister("ra"))

def hmac(emu, buttons):
    # set initial memory/registers state
    emu.writeRegister("gp", 0x8001C7E0)
    emu.writeRegister("sp", 0x807FFDB8)
    emu.writeMemory(get_addr(0x807FFDD0), buttons)
    emu.writeMemory(get_addr(0x80010040), KEY)
    emu.writeMemory(get_addr(0x80010000), "\x80" + "\x00"*0x3f)

    start_addr = 0x80013604 # just after PadGetButton
    end_addr = get_addr(0x80013f78) # end of hmac computation
    emu.writeRegister(emu.getPCRegister(), start_addr)

    while not monitor.isCancelled():
        pc = emu.getExecutionAddress()
        if pc == end_addr:
            # returns hmac result
            h = emu.readMemory(get_addr(0x807FFDD8), 20)
            return h.tostring()

        if pc == get_addr(0xa0):
            t1 = emu.readRegister("t1")
            syscall(emu, t1)

        success = emu.step(monitor)
        if not success:
            raise Exception("Emulation error: %s" % emu.getLastError())

emu = EmulatorHelper(currentProgram)
monitor.initialize(65536)
monitor.setMessage("CeriseLiquide pwn in progress...")
buttons = ["z", "x", "s", "d"]
with open("/tmp/psx_hashes.txt", "wb") as f:
    for i, p in enumerate(itertools.product(buttons, repeat=8)):
        b = "".join(p)
        h = hmac(emu, b)
        f.write(h.encode("hex") + "\n")
        monitor.incrementProgress(1)
```

Une fois le challenge terminé, ma curiosité m'a poussé à reverser ce HMAC à tête reposée. J'ai alors compris que la fonction de hachage utilisée était RIPEMD160 et non SHA-1. Mais puisque les deux algorithmes partagent les même constantes d'initialisation et la même taille de condensat, ça explique la confusion quand on les regarde de loin. Au niveau de la construction du HMAC, il n'y a aucun piège, c'est un HMAC-RIPEMD160 standard. Voici donc un script qui génère exactement la même sortie (les 65536 mots de passe) en 4 secondes, contre 3 heures pour l'émulation Ghidra.

```
from Crypto.Hash import HMAC, RIPEMD160
import itertools

key = "\x97\x8b\x8b\x8f\x8c\xc5\xd0\xd0\x88\x88\x88\xd1\x86\x90\x8a\x8b\x8a\x9d\x9a\xd1\x9c\x90\x92\xd0" +
"\x88\x9e\x8b\x9c\x97\xc0\x89\xc2\xbc\x94\x89\xaa\xbe\x93\xc8\xa6\xcb\x92\x88"

buttons = ["z", "x", "s", "d"]
with open("/tmp/psx_hashes.txt", "wb") as f:
    for p in itertools.product(buttons, repeat=8):
        b = "".join(p)
        h = HMAC.new(key, b, RIPEMD160).hexdigest()
        f.write(h + "\n")
```

Mais je ne regrette pas mon choix puisqu'il m'a permis de manipuler pour la première fois (et probablement pas la dernière !) l'émulateur de Ghidra.

5.9 Bruteforce de la base keepass

Nous avons un fichier avec tous les mots de passe possibles. Il ne reste plus qu'à les tester. Pour cela, j'ai utilisé le module Python `libkeepass` [23] qui supporte l'ancien format de base keepass utilisé par CeriseLiquide. J'ai juste eu besoin de modifier un tout petit peu le code pour que le module accepte les mots de passe qui ne sont pas ASCII (juste un `.encode('utf-8')` dans `common.py` à supprimer).

```python
import libkeepass
import sys

with open("/tmp/psx_hashes.txt", "rb") as f:
    hashes = f.read().split()

for i,h in enumerate(hashes):
    # progress bar
    sys.stdout.write("\r%02d%% (%05d/%d)" % (i*100/len(hashes), i, len(hashes)))
    try:
        with libkeepass.open("Database.kdb", password=h.decode("hex")) as db:
            print("\nwin: %s" % h)
            print(db.pretty_print())
            sys.exit(0)
    except IOError:
        pass
```

La base s'ouvre avec le mot de passe 2d3cdc09e039d88928b9e988b9653ce5c294fe4b et elle contient le flag de ce 5ème niveau.

```
$ python bruteforce_keepass.py
70% (46200/65536)
win: 2d3cdc09e039d88928b9e988b9653ce5c294fe4b
<?xml version='1.0' encoding='utf-8' standalone='yes'?>
<pwlist>
<pwentry>
        <group>General</group>
        <title>kazh-boneg</title>
        <username>CeriseLiquide</username>
        <url/>
        <password>SSTIC{02879b337f4987351d11e90570c91a696320a9985a7d6923dc2d1da3cfe6155b}</password>
        <notes/>
        <uuid>bc99add88fd98db760eb5dd26036ae0f</uuid>
        <image>0</image>
        <creationtime>2020-03-29T11:51:57</creationtime>
        <lastmodtime>2020-03-29T11:51:57</lastmodtime>
        <lastaccesstime>2020-03-29T11:51:57</lastaccesstime>
        <expiretime expires="False">2999-12-28T23:59:59</expiretime>
</pwentry>
</pwlist>
```

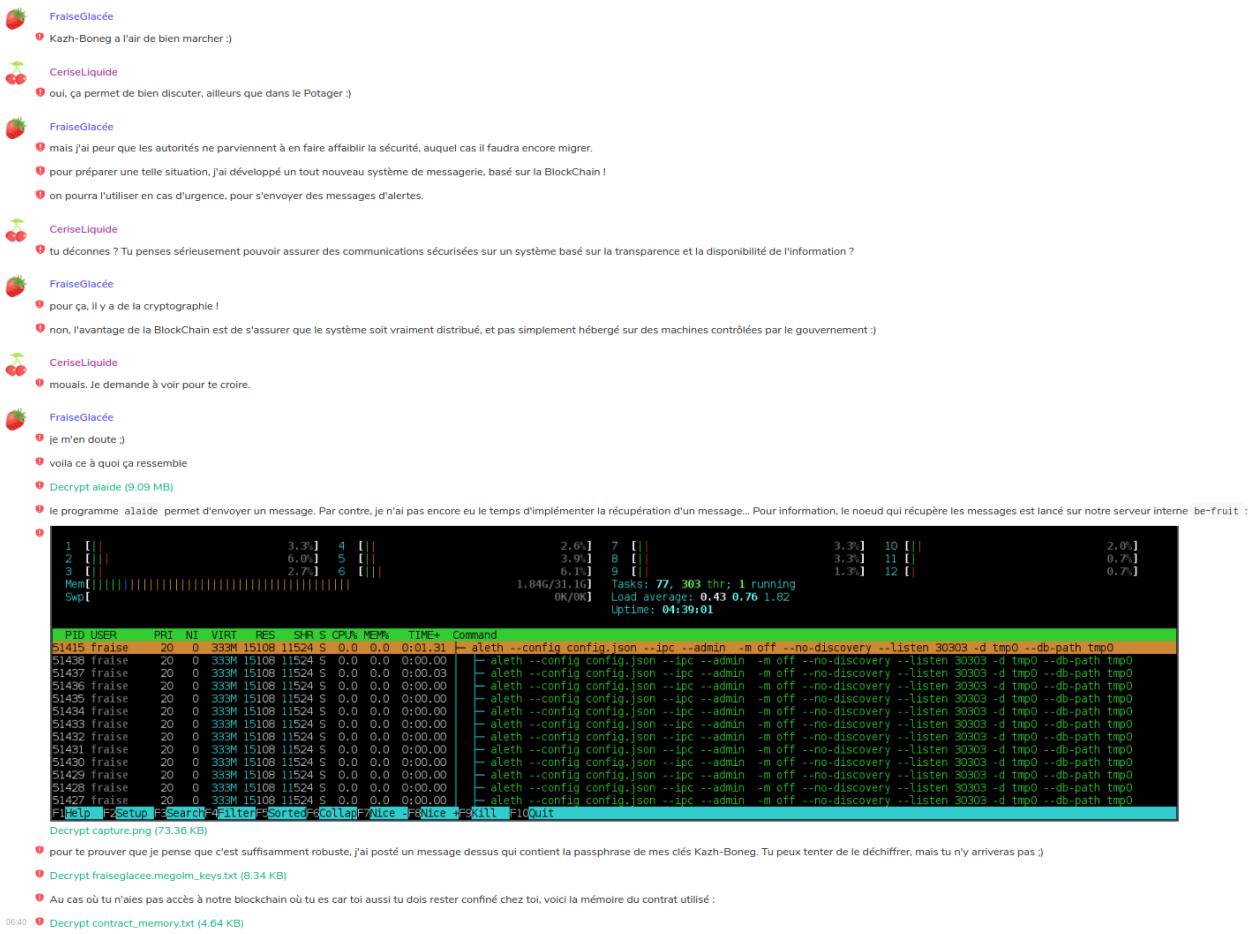
```
SSTIC{02879b337f4987351d11e90570c91a696320a9985a7d6923dc2d1da3cfe6155b}
```

23. https://github.com/libkeepass/libkeepass

6 Niveau 6

6.1 À l'aide

Grâce au 5ème flag, on peut déchiffrer les clés Megolm de CeriseLiquide. Pour être sur de ne pas rater de conversation privée, on se connecte sur notre miroir de Kazh-Boneg avec le compte de CeriseLiquide. Ceci débloque l'accès au salon `#lb`, qui contient des photos de Lobster Dog (et aussi d'un LobsterRabbit si mignon), l'accès au salon `#maths` dont on reparlera plus tard, ainsi qu'une conversation privée entre CeriseLiquide et FraiseGlacée.



Afin d'échapper au contrôle du gouvernement, FraiseGlacée a développé un système de messagerie basée sur la BlockChain. Il nous donne le programme `alaide` (9.09 Mo!!) qui permet d'envoyer des messages, et pour prouver que son système est sécurisé, il s'en est servi pour envoyer le mot de passe de sa clé Megolm.

Il va falloir casser son système de messagerie et voler sa clé Megolm. Au travail.

6.2 alaide

Le programme `alaide` peut fonctionner en mode local ou peer, et permet d'envoyer un message de 256 caractères maximum dans une blockchain. Quand on le lance, on apprend qu'il est basé sur `aleth` 1.8.0.

Quelle que soit la longueur du message qu'on souhaite envoyer, `alaide` va envoyer 32 morceaux de message, puis afficher tout un ensemble de memory slot.

```
$ ./alaide
Usage :

    alaide PEER MESSAGE

Where PEER is 'local' or 'enode://peer' and MESSAGE is an ASCII string (max. 256 characters)
If 'local' is used, message won't be synchronized with a remote peer.

$ ./alaide local bonjour
              __
             /--
      _______ //
     /_______'/|    Alaide !
     [        ]|    ========
     [ Jus de ]|
     [ cerise ]|    Envoi d'un SOS sur la blockchain du Sivi Ha Kerez
     [   .    ]|    Base sur aleth 1.8.0
     [ _/ \_  ]|
     [(_) (_) ]/
     '--------'

Deploying...
Smart contract has been deployed.
Sending message part 1/32 ...
Sending message part 2/32 ...
[...]
Sending message part 32/32 ...
Contract memory is now :
Memory slot 755311B9E2CEE471A91B161CCC5DEED933D844B5AF2B885543CC3C04EB640983 =
    000000000000000000000000000000190CB8F0C93F08D138E9088AD2D55B00
Memory slot 68FB8E7CAD479CCC9244A179D64897454189FD25DB04E15D3A5135327A17597B =
    000000000000000000000000000000D189FBBCB85A1F9F104DA2D3EB6AF280
[...]
Memory slot 2B00120B81607971383F6F5676C1551D6BB27BE3F263689FD3630E1A5BE14018 =
    000000000000000000000000000000BD1D4CBEBFDE2EB426BAAAE213D16889
Done.
```

Aleth regroupe un ensemble d'outils codés en C++ pour Ethereum. La blockchain du Sivi-Ha-Kerez se base donc sur Ethereum.

Étant novice en blockchain, une bonne étape de documentation a été nécessaire pour comprendre ce que j'avais entre les mains et me familiariser avec Ethereum et les Smart Contracts.

Quand on ouvre `alaide` avec Ghidra, on constate que c'est toujours aussi désagréable de reverser du C++. Dans les strings, on trouve des morceaux de JSON dont une longue chaîne hexa qui semble être un contrat intelligent. On remarque également qu'`alaide` contient un mode verbose mais qui est désactivé. On patche donc un octet du binaire pour l'activer par défaut.

```python
with open("alaide", "rb") as f:
    b = bytearray(f.read())

# enable verbose mode
# mov byte ptr [RDI], 0x0 => mov byte ptr [RDI], 0x1
b[0x6dda6] = 1

with open("alaide_verbose", "wb") as f:
    f.write(b)
```

Grâce au mode verbose, on voit passer le contrat qui est déployé sur la blockchain. On obtient également les transactions, c'est à dire les fonctions du contrat qui sont appelées et leurs arguments.

```
Sending message part 1/32 ...
Transaction data is 0xb5b6d2a800000000000000000000000000000000000000000000000004e6414dfc88d288300000000
0000000000000000000000000000000000000000000000000000000000000b4
Part 1/32 is sent. Hash is 481519fb5ed71e25795b19397a5884a8c3c05de20341722c4529903a912e447c
```

Quand on envoie deux fois le même message et qu'on fait un diff, on remarque que le contrat est presque identique (seul les 8 derniers octets changent). Par contre, les transactions et le contenu des memory slot sont très différents.

Comme `alaide` est basé sur `aleth` 1.8.0, j'ai compilé cette version d'`aleth` dans l'espoir de porter ses symboles dans la base Ghidra d'`alaide`. Ne parvenant pas à trouver les bons plugins Ghidra pour arriver à mes fins, je n'ai pas insisté. Cependant, maintenant qu'on a un `aleth` qui fonctionne, on peut utiliser `alaide` en mode peer, tel que décrit dans le screenshot `htop` partagé par FraiseGlacée.

```
$ ./aleth --config ../../../config.json --ipc --admin password -m off --no-discovery --listen 30303
-d tmp0 --db-path tmp0
```

J'ai vu deux grands avantages à utiliser `alaide` en mode peer plutôt qu'en mode local.

1. On peut tirer profit de l'API JSON-RPC de notre node aleth (par exemple, pour aller explorer l'état de la blockchain avec des outils dédiés)
2. On peut rajouter l'option `-v4 -log-vmtrace` à `aleth`, ce qui génère une trace de la machine virtuelle dans laquelle s'exécute le contrat. La trace est tellement verbeuse qu'on ne peut stocker qu'une trace partielle (une ou deux des 32 transactions). Mais ces traces partielles me seront très utiles plus tard pour m'aider à comprendre certaines subtilités du contrat.

Il est temps de trouver des outils pour examiner ce smart contract.

6.3 Trouver les bons outils

Quand on veut travailler sur la blockchain Ethereum, ce ne sont pas les outils qui manquent. Il y en a énormément. Peut être trop. Tout l'écosystème bouge très vite. La difficulté est plutôt de réussir à trouver l'outil qui va répondre efficacement à notre besoin très précis... et qui fonctionne encore (car comme tout bouge très vite et que beaucoup d'outils dépendent les uns des autres, on tombe souvent sur des dépendances cassées).

Un smart contract Ethereum s'écrit (généralement) en language Solidity, qui sera ensuite compilé en bytecode EVM (Ethereum Virtual Machine). La plupart des outils de l'écosystème sont conçus pour travailler sur du code source Solidity. Or, on dispose uniquement du bytecode EVM du contrat.

Une autre partie non négligeable des outils est faite pour travailler sur *la* blockchain Ethereum, et non sur une blockchain privée. Il y a donc pas mal de tri à faire pour trouver l'outil dont on a besoin.

Pour le fun, voici la liste des outils que j'ai installé et (plus ou moins vaguement) testé pour l'occasion :

— ethereum-dasm
— ethereum-lite-explorer
— ethjsonrpc
— evm-disassembler
— evmdasm
— geth
— manticore
— mythril
— panoramix
— piet
— rattle

Tout ça pour au final utiliser presque exclusivement un outil en ligne ! Car c'est sur https://ethervm.io/decompile que j'ai trouvé mon bonheur, ce site permettant de décompiler du bytecode EVM en code Solidity.

Quand on fournit notre contrat, on obtient le code suivant

```
contract Contract {
    function main() {
        memory[0x40:0x60] = 0x80;
        var var0 = msg.value;

        if (var0) { revert(memory[0x00:0x00]); }

        var temp0 = memory[0x40:0x60];
        var0 = temp0;
        var temp1 = code.length - 0x06c0;
        var var1 = temp1;
        memory[var0:var0 + var1] = code[0x06c0:0x06c0 + var1];
        memory[0x40:0x60] = var1 + var0;

        if (var1 < 0x20) { revert(memory[0x00:0x00]); }

        storage[0x00] = (storage[0x00] & ~0xffffffff) | 0x00;
        storage[0x00] = ((memory[var0:var0 + 0x20] ~ 0x4d30442053515541) & 0xffffffffffffffff)
             * 0x0100 ** 0x04 | (storage[0x00] & ~(0xffffffffffffffff * 0x0100 ** 0x04));
        storage[0x00] = (storage[0x00] / 0x0100 ** 0x04 & 0xffffffffffffffff)
            * 0x0100 ** 0x14 | (storage[0x00] & ~(0xffffffffffffffff * 0x0100 ** 0x14));
        storage[0x00] = (storage[0x00] & ~(0xffffffffffffffff * 0x0100 ** 0x0c)) | (0xffffffffffffffff &
            0x5245205230305421) * 0x0100 ** 0x0c;
        memory[0x00:0x059e] = code[0x0122:0x06c0];
        return memory[0x00:0x059e];
    }
}
```

Ce code se contente s'initialiser le storage[0x00] du contrat avec une valeur calculée à partir des constantes 0x4d30442053515541 et 0x5245205230305421, ce qui en ASCII donne « `MOD SQUARE ROOT!` », un indice pour la suite.

Mais il y a un piège car il reste encore beaucoup de code à décompiler. Les contrats semblent toujours commencer par les octets 0x608060. Or, dans notre contrat, cette séquence

apparaît deux fois. Quand on décompile la deuxième partie, on obtient 156 lignes supplémentaires de code Solidity.

Dans ce code, on trouve l'endroit qui va recevoir les transactions envoyées par `alaide`, effectuer des calculs mathématiques dessus, puis les stocker dans des storages [24]. La trace générée par l'option `-log-vmtrace` de la node `aleth` permet de comprendre comment le code s'exécute pas à pas, et de voir l'évolution de la memory et des storages. D'ailleurs, petite digression. Dans le vocabulaire Ethereum, `memory` désigne un équivalent de la mémoire RAM, qui va servir à effectuer des calculs temporaires et sera ensuite nettoyée (ça ne coûte pas cher). Au contraire, `storage` serait plutôt comparable à un disque dur et permet de stocker des données de façon persistance dans la blockchain (ça coûte cher).

6.4 Un plan en trois étapes

A ce stade, je commence à bien cerner ce qu'il va falloir faire pour terminer le niveau et un plan en trois étapes se déssine. Plan qui va étonnement se dérouler comme prévu.

La première étape, c'est de me débarasser complètement de la partie blockchain, c'est à dire recoder en Python l'équivalent du smart contract, donc un script qui prend 32 transactions en entrée et qui est capable de reproduire l'état des storages. Une fois cette étape passée, on pourra donc oublier tout ce qui touche à Ethereum et Solidity et toute la surcouche de complexité inhérente à la blockchain.

Ensuite, il faudra inverser le contrat, donc écrire un script qui à partir de l'état final des storages soit capable de retrouver les 32 transactions.

Enfin, il faudra reverser `alaide` pour comprendre comment le message texte qu'on envoie est converti en transactions et être capable de faire l'opération inverse. Alors, les clés Megolm de FraiseGlacée seront à nous !

6.5 Étape 1 - se débarrasser de la blockchain

On commence par parser les sorties de `alaide_verbose` afin de récupérer les 32 transactions envoyées, l'état final des storages et le nonce du contrat (les 8 derniers octets aléatoires du contrat). Le but est d'écrire un code qui soit capable de recalculer l'état des storages en fonction du nonce et des transactions.

Les transactions font 68 octets, mais il n'y en a que 13 utiles :
— L'identifiant de la fonction appelée (basé sur un hash), sur 4 octets. Chaque transaction appelle la même fonction (0xb5b6d2a8), on peut donc ignorer cette partie.
— Le premier argument et 8 octets.
— Le second argument d'un octet.

Chaque transaction va entraîner le stockage d'une valeur dans un storage. La clé de ce storage est un hash keccak256 d'un buffer de 64 octets (rempli de 0x00, hormis un compteur d'appels et un 0x01). Il faut savoir que l'opcode EVM qui calcule ce hash s'appelle SHA3, mais que ce n'est pas tout a fait un SHA3 standard qui est utilisé par Ethereum. En effet,

24. storage == memory slot

entre le moment où Keccak256 a été ajouté à Ethereum et le moment où l'algorithme a été standardisé SHA3 dans sa forme définitive, le padding utilisé a été modifié.

Le storage[0x00] est particulier et sert à stocker 3 valeurs :

— Un compteur d'appel, qui s'incrémente après chaque transaction de type 0xb5b6d2a8de (il varie donc entre 0 et 32)
— Une constante de 8 octets, initialisée au chargement du contrat.
— Une variable de 8 octets. Elle est initialisée en faisant un xor du nonce du contrat avec une constante. Ensuite, à chaque nouvelle transaction, cette valeur va être dérivée.

Voici le script Python3 qui reproduit le comportement du smart contract.

```python
from Crypto.Hash import keccak

M64  = (2**64)  - 1
M128 = (2**128) - 1
M160 = (2**160) - 1

def ror(x, n, bits = 32):
    mask = (2**n) - 1
    mask_bits = x & mask
    return (x >> n) | (mask_bits << (bits - n))

def rol(x, n, bits = 32):
    return ror(x, bits - n, bits)

def parse_logs(path):
    args = []
    storage = {}
    nonce = None
    with open(path, "r") as f:
        lines = f.read()
    for line in lines.splitlines():
        if line.startswith("Contract data is"):
            nonce = int(line[-16:], 16)
        elif line.startswith("Transaction data is"):
            arg0 = int(line[78:78+16], 16)
            arg1 = int(line[-2:], 16)
            args.append((arg0, arg1))
        elif line.startswith("Memory slot"):
            l = line.split(" ")
            k = int(l[2], 16)
            v = int(l[4], 16)
            storage[k] = v
    return nonce, args, storage

class Contract(object):
    def __init__(self, nonce):
        # nonce is the 8 bytes at the end of contract code
        self.nb_call = 0
        self.nonce = nonce
        r = (nonce ^ 0x4d30442053515541)
        init_s0 = (r << 160) | (0x5245205230305421 << 96) | (r << 32)
        self.s0 = []
        self.storage = {}
        self.storage_keys = []
        # precompute all s0 values and storage keys
        s0 = init_s0
        for i in range(32):
            s0 = self.deriv(s0) + i
            self.s0.append(s0)
            self.storage_keys.append(self.get_storage_key(i))

    def call(self, u64_arg0, u8_arg1):
        arg = self.mix_args(u64_arg0, u8_arg1)
        x = (self.s0[self.nb_call] >> 160) ^ arg
        storage_key = self.storage_keys[self.nb_call]
```

```
56          self.storage[storage_key] = ((x * x * 0x854e9fb4699ed8f22fd89ebe3f17f7f6)  +
57              (x * 0xd677105721b51a080288a52f7aa48517) ) & M128
58          self.nb_call += 1
59
60      def deriv(self, storage):
61          for i in range(0x40):
62              var1 = (storage >> 160) & ((storage >> 96) & M64)
63              var2 = var1
64              for j in range(0x40):
65                  var2 = var2 ^ (var1 >> (j+1))
66              x = storage >> 160
67              x = ((x >> 1) | (var2 << 0x3f)) & M64
68              storage = (x << 160) | storage & M160
69          return storage
70
71      def mix_args(self, x, y):
72          for i in range(8):
73              x = x ^ (y << ((i * 8) & 0xFF))
74              y = rol(y, 1, 8)
75          return x
76
77      def get_storage_key(self, nb_call):
78          h = keccak.new(digest_bits=256)
79          counter = ("%X" % nb_call).rjust(64, "0").ljust(127, "0") + "1"
80          return int(h.update(bytes.fromhex(counter)).hexdigest(), 16)
81
82  def check_contract(log_file):
83      nonce, args, storage = parse_logs(log_file)
84      ctc = Contract(nonce)
85      for i, (arg0, arg1) in enumerate(args):
86          ctc.call(arg0, arg1)
87          k = ctc.storage_keys[i]
88          assert ctc.storage[k] == storage[k]
89      print("Success. Farewell, blockchain !")
90
91  if __name__ == '__main__':
92      import sys
93      check_contract(sys.argv[1])
```

```
$ ./alaide_verbose local bonjour > /tmp/bonjour
$ python3 smart.py /tmp/bonjour
Success. Farewell, blockchain !
```

6.6 Étape 2 - inverser le contrat

Dans cette deuxième étape du plan, on veut être capable de récupérer le nonce et les transactions à partir de l'état final des storages.

Pour le nonce c'est facile, il se recalcule directement à partir de storage[0x00]. Pour récupérer les transactions, c'est plus complexe puisqu'il va falloir résoudre des équations quadratiques modulaires. Aie.

En effet, nous allons devoir résoudre les 32 équations suivantes :

$$177195561791243721643870281543305394166x^2 + 285073005845973382283673713514754901271x \equiv y (\mathrm{mod}\ 2^{128})$$

Avec y les valeurs stockées dans les storage, et x l'inconnue qui permettra de calculer la valeur des transactions.

Il s'agit d'un cas particulier d'équation quatratique modulo une puissance de 2, qu'on écrit sous la forme :

$$ax^2 + bx + c \equiv 0 \pmod{2^n}$$

Dans le salon `#maths` on nous donne justement un peu d'aide pour mieux comprendre l'arithmétique modulaire : un lien [25] vers des fichiers LaTeX écrits par Nicolas Iooss.

Je suis donc allé compiler les fichiers de Nicolas, qui ont des dépendances vers approximativement tous les packages LaTeX de la Terre... Ce qui n'a pas manqué de saturer une fois de plus mon disque dur. Cette fois-ci, j'ai du sacrifier mes derniers jeux Steam sur l'autel du SSTIC.

Après avoir consulté les fichiers de Nicolas, j'ai trouvé une publication [26] intitulée *Complete solving the quadratic equation mod* 2^n et qui est entièrement dédiée au sujet qui nous concerne.

L'idée générale pour résoudre cette famille d'équation est de tirer profit du modulo 2^n et de le transformer en modulo 2 au bout de n divisions et réécritures. À chaque itération, on va résoudre un bit de l'inconnue, qui va directement dépendre de la parité de c.

Le papier propose justement un algorithme en pseudo-code permettant de résoudre ces équations. Je l'ai adapté en Python. Voici le code permettant d'inverser l'algorithme des contrats. Il peut s'utiliser avec une trace complète `alaide_verbose` afin de vérifier que tous les calculs sont corrects. Il peut également s'utiliser avec une trace partielle contenant uniquement les storages (tel que dans le fichier `contract_memory.txt` fourni par FraiseGlacée), auquel cas il va recalculer les 32 transactions envoyées par FraiseGlacée.

```python
def break_contract(log_file, verbose=False):
    dbg_nonce, dbg_args, storage = parse_logs(log_file)
    # compute nonce from storage[0x00]
    nonce = (0x4d30442053515541 ^ (storage[0] >> 32)) & M64
    if dbg_nonce:
        assert nonce == dbg_nonce
    elif verbose:
        print("nonce: 0x%x" % nonce)
    a = 0x854e9fb4699ed8f22fd89ebe3f17f7f6
    b = 0xd677105721b51a080288a52f7aa48517
    n = 128
    ctc = Contract(nonce)
    out = []
    for i, k in enumerate(ctc.storage_keys):
        d = storage[k]
        c = -d & M128
        x = quadratic_solve(a, b, c, n)
        arg = x ^ (ctc.s0[i] >> 160)
        if dbg_args:
            arg0, arg1 = dbg_args[i]
            dbg_arg = ctc.mix_args(arg0, arg1)
            assert arg == dbg_arg
        elif verbose:
```

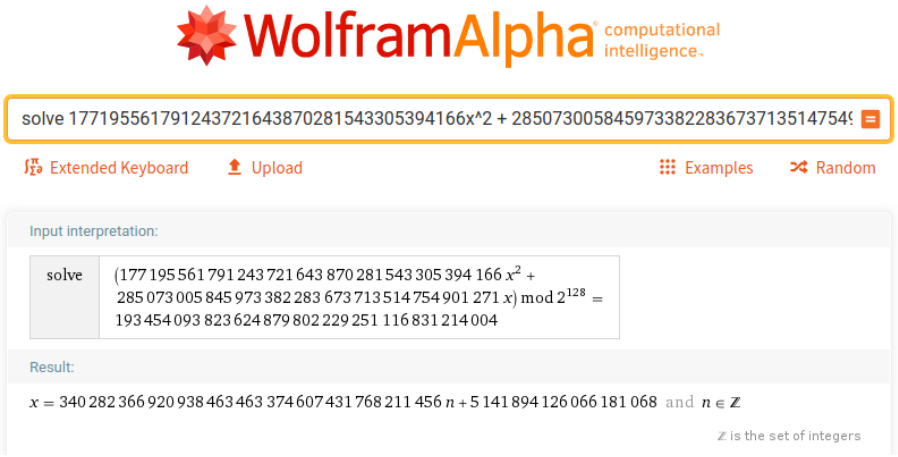
25. https://github.com/fishilico/shared/tree/4fa8c8225e027113a551a395db544cd118faa1df/latex
26. https://arxiv.org/pdf/1711.03621.pdf

```
24              print("0x%016x" % arg)
25          out.append(arg)
26      if dbg_args:
27          print("Success. Farewell, modular arithmetic !")
28      return out
29
30  def quadratic_solve(a, b, c, n):
31      def div2(a, b, c):
32          if c % 2 == 0:
33              return 2*a, b, c//2, 0
34          else:
35              return 2*a, 2*a+b, a//2 + b//2 + c//2 + 1, 1
36      bits = []
37      while n:
38          a, b, c, bit = div2(a, b, c)
39          bits.insert(0, str(bit))
40          n -= 1
41      return int("".join(bits), 2)
42
43  if __name__ == '__main__':
44      import sys
45      break_contract(sys.argv[1], verbose=True)
```

```
$ python3 smart.py /tmp/bonjour
Success. Farewell, modular arithmetic !
$ python3 smart.py contract_memory.txt
nonce: 0xf91c812941bb22c7
0x178789432c7bc26f
0x2cd9952958e79a65
0x15c02a2f2baa379f
0xff7b60885493f139
0x7167df0fce3a5733
0xb5de4352d5f336d2
0x4cdf31b73f926d0c
0xf7af0d7a1da775ce
0x767c733bf40c6388
0xbdd8fe1ecd06e16e
0x5eaf1852399a7bf6
0x6cbd587f8e90fae6
0xe66221a0a2006c26
0x19c18d80b803a23a
0x53190be88bf03f98
0x69399d6cd76bc1cd
0x03405cd8bd7ee70d
0x143db3dd3111d750
0x689aa34431522046
0x3eb810db119bfc16
0x5fee09732b5243b9
0x59bcaf7df995f43a
0x310b467463313a90
0xac1585cd357e0289
0x2414d20b752e10dc
0x9710873789c59955
0xdefd516b952b689f
0x94738c06da2aa19a
0xbea5aaec9b6eb8df
0x8266427ff58fbf96
0x8adc2a372571dc92
0x1c851be999b483be
```

Le seul problème c'est que mon niveau en maths étant ce qu'il est, je n'ai presque rien compris au papier *Complete solving the quadratic equation mod* 2^n au moment de la résolution du challenge. Je suis donc passé à coté de l'algorithme qu'il contenait. Le script ci-dessus, je viens de l'écrire à tête reposée au moment où je rédige cette solution. Ma vraie solution pour

résoudre ce niveau rapidement (et donc sans prendre le temps de combler mes lacunes pour comprendre le vocabulaire et l'écriture mathématique) est bien plus scandaleuse. J'ai tout simplement copié-collé mes 32 équations quadratiques, une par une, dans WolframAlpha, qui m'a bien gentiment donné les résultats. Je pense qu'on peut difficilement faire moins élégant !



6.7 Étape 3 - Convertir les transactions en message

On est prêt du but ! Il reste à trouver comment les messages qu'on envoi avec `alaide` sont découpés en transactions. Mais comme je n'avais plus très envie de reverser du C++, j'ai fait la majeure partie de cette tâche en boîte noire.

Mais pour ça, il faut d'abord se débarrasser du random. Dans `alaide`, on trouve une fonction qui va utiliser `/dev/urandom`. J'ai donc patché le début de cette fonction pour qu'elle renvoie toujours 9 (https://dilbert.com/strip/2001-10-25). Ainsi, l'aléatoire disparaît. Quand on envoie deux fois le même message, on obtient deux fois la même trace et les même storages.

Une fois qu'on a constaté que le nonce du contrat vaut désormais toujours 9, je me suis dit qu'il serait plus judicieux de patcher le random pour qu'il renvoie le nonce de FraiseGlacée.

```python
with open("alaide_verbose", "rb") as f:
    b = bytearray(f.read())

# disable random
# always returns nonce of FraiseGlacee contract (0xf91c812941bb22c7)
#
# 46D240 mov rax, ds:qword_46D251
# 46D24A mov [rsi], rax
# 46D24D xor rax, rax
# 46D250 retn
# 46D251 qword_46D251 0F91C812941BB22C7h

patch = bytes.fromhex("48a151d2460000000000004889064831c0c3c722bb4129811cf9")
b[0x6d240:0x6d240+len(patch)] = patch

with open("alaide_pwned", "wb") as f:
    f.write(b)
```

Je n'ai pas encore parlé des deux arguments qui composent les transactions. Il y a 8 octets de données utiles à envoyer, mais `alaide` envoie un 9ème octet généré aléatoirement qui sert à mélanger un peu la donnée envoyée (de manière réversible). Maintenant qu'on a fixé le random, `alaide_pwned` choisit systèmatiquement 0xc7 comme octet de mélange.

Sans random, l'analyse devient beaucoup plus facile. On peut chiffrer des messages de 256 caractères, faire varier les caractères un par un, faire un diff des logs générés et comprendre quelle transaction a été impactée par le changement.

On en tire deux conclusions importantes :

— Si on ne change qu'un seul octet du message envoyé, il n'y aura qu'un octet d'une seule transaction qui sera modifié.

— Les transactions ne sont pas envoyées dans l'ordre du message. Par exemple, les 8 premiers octets d'un message sont envoyés dans la 17ème transaction (et non pas dans la première comme on pouvait s'y attendre).

On peut donc tout résoudre en boîte noire. J'ai généré une trace d'exécution avec `alaide_pwned` et un message composé de 256 **A**. Quand on XOR les transactions obtenues avec celles de CeriseLiquide et 0x4141414141414141, on retrouve le texte clair. Ou presque. Car on ne sait pas lequel des 256 octets de mélage a été choisi. Comme on sait que la sortie est du texte imprimable, il suffit de tester les 256 possibilités.

```python
import string
import struct

def get_flag():
    transaction_order = [17, 6, 11, 15, 24, 12, 26, 30, 3, 14, 23, 18, 2, 22, 20, 19,
        8, 28, 27, 31, 29, 25, 16, 4, 7, 32, 10, 13, 1, 9, 21, 5]
    msg = bytearray()
    transactions = break_contract("contract_memory.txt")
    nonce, AAAA_transactions, _ = parse_logs("/tmp/AAAA")
    charset = string.printable + "\x00"
    ctc = Contract(nonce)
    AAAA = [ctc.mix_args(a0, a1) for a0, a1 in AAAA_transactions]
    for i in transaction_order:
        target = transactions[i-1] ^ AAAA[i-1] ^ 0x4141414141414141
        for j in range(0x100):
            p = ctc.mix_args(target, j)
            plaintext = struct.pack("<Q", p)
            if all(chr(c) in charset for c in plaintext):
                msg += plaintext
    print(msg.decode())

if __name__ == '__main__':
    get_flag()
```

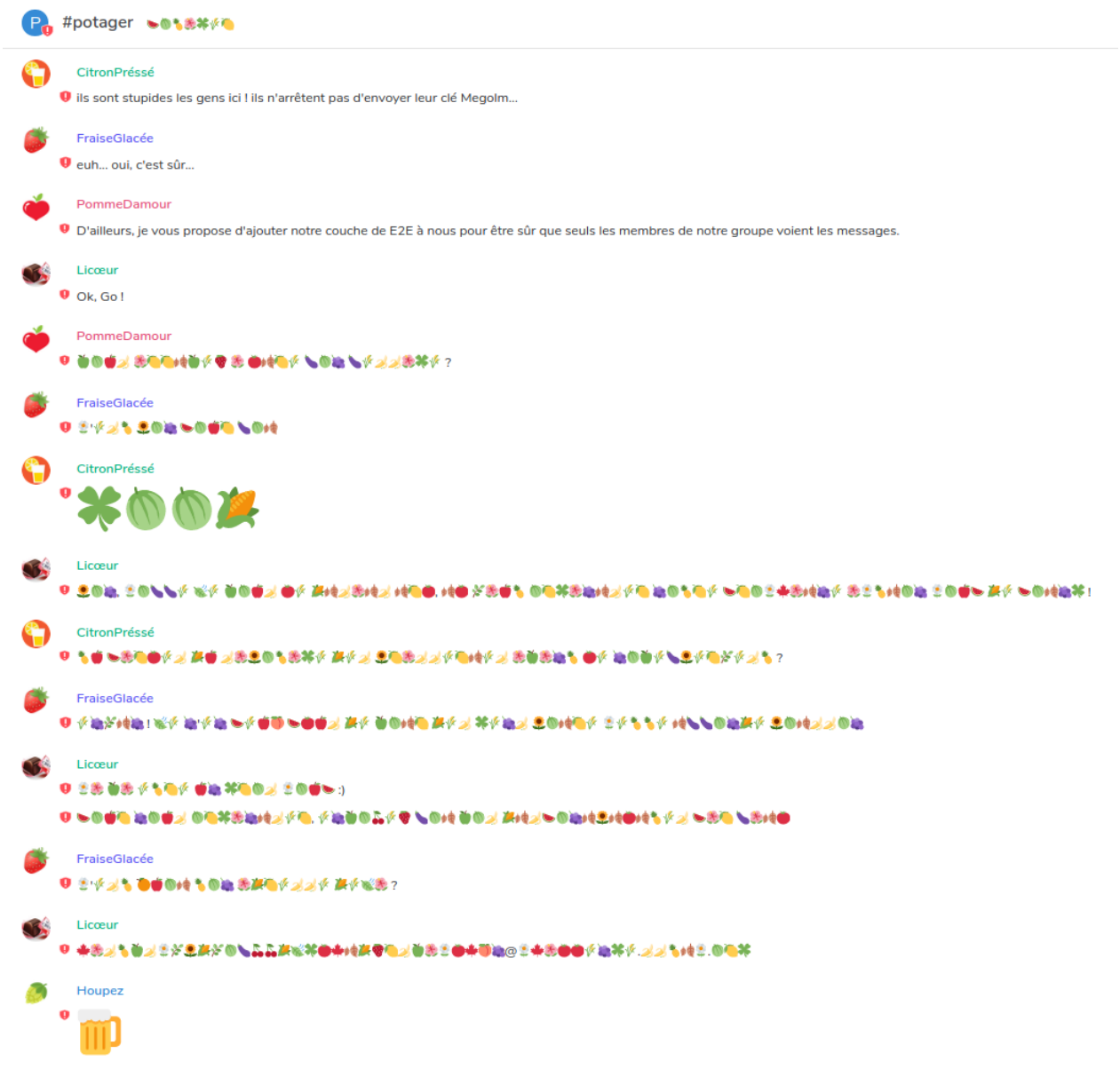
```
$ ./alaide_pwned local $(python -c "print 'A'*256") > /tmp/AAAA
$ python3 smart.py
NO WAY! Personne n'arrivera jamais a dechiffrer ce message sans mon programme de dechiffrement.
Je m'en sers donc de pense-bete xD. La passphrase de mon export megolm est
SSTIC{825145b95c18a865b78f750e153fcc1767f3954e9d629c781ef9150803f3f80f}...
```

Voilà qui conclue le dernier (dernier ?) niveau du challenge SSTIC 2020 !

```
SSTIC{825145b95c18a865b78f750e153fcc1767f3954e9d629c781ef9150803f3f80f}
```

7 Niveau bonus

La clé Megolm de FraiseGlacée permet de déchiffrer le salon `#potager`.



Pour casser ces messages codés en emoji fruits et légumes, on s'aide d'un message sur lequel on a un clair connu, puisqu'on sait que l'adresse email de validation est de la forme `@challenge.sstic.org`.

Licœur

Comme on pouvait le supposer, il s'agit d'une substitution monoalphabétique. On observe bien deux tomates à la places des deux `l` de challenge, et deux bananes pour les deux `s` de sstic.

J'avais imaginé qu'il faudrait deviner les caractères un par un en reconstruisant des phrases qui ont du sens. Mais en codant le script traduisant les emoji en caractères, j'ai remarqué que le `s` de sstic était remplacé par un emoji de valeur unicode 0x1F34C, tandis que le `t` avait un emoji 0x1F34D.

Autrement dit, les valeurs se suivent, il suffit donc de faire une sorte de César pour déchiffrer tous les messages. L'unicode est bien gérés par Python3.

```python
msg = [...]

def emoji_decode(c):
    if ord(c) > 0x1f2d9:
        return chr(ord(c) - 0x1f2d9)
    return c

for m in msg:
    print ("".join(emoji_decode(c) for c in m))
```

```
$ python3 emoji.py
vous arrivez a lire mon message ?
c'est bon pour moi
good
bon, comme je vous le disais irl, il faut organiser notre prochaine action coup de poing !
tu parles du sabotage des brasseries avant le novemberfest ?
enfin ! je n'en peux plus de voir des gens boire cette immonde boisson
ca va etre un gros coup :)
pour nous organiser, envoyez moi vos disponibilites par mail
c'est quoi ton adresse deja ?
hastvscfbdfomyydjglhidzrsvaclhxn@challenge.sstic.org
\T1\textexclamdown
intrusssssssssssssss
code rouge
mayday
```

```
hastvscfbdfomyydjglhidzrsvaclhxn@challenge.sstic.org
```

Conclusion

J'ai trouvé cette édition 2020 du challenge SSTIC excellente.

Le thème de fond était sympathique. On découvre les agissements du Sivi-Ha-Kerez qui promeut les jus de fruit et lutte contre la bière et les boissons alcoolisées. L'édition 2019 du challenge se terminait sur une blague complotiste mettant en scène intoxication alimentaire du social event du SSTIC 2018 comme étant un coup monté. Cette année, on peut presque se demander si l'annulation du social event 2020 n'est pas un sale coup du Sivi-Ha-Kerez.

Sur la forme, j'ai beaucoup apprécié l'idée d'utiliser Matrix/Synapse/Riot comme interface centrale du challenge. Quand on termine une épreuve, on a hâte de découvrir les salons qui seront déchiffrés et de surprendre de nouvelles conversations entre les différents protagonistes. Certes, j'ai un peu pesté lors de la deuxième épreuve du challenge, quand il s'agissait de faire de l'administration système pour déployer les différentes briques qui composent Kazh-Boneg, mais ça en valait clairement le coup. D'ailleurs, au fil des épreuves, j'ai trouvé Matrix de plus en plus intuitif.

Sur le contenu des épreuves en tant que telles, c'était varié et intéressant. J'ai particulièrement aimé le coté réaliste mais old-school du niveau 4, avec la RCE Windows 98 à analyser de la compromission jusqu'à l'exécution du malware et le vol de données. Je suis content d'avoir pu découvrir l'émulateur de Ghidra grâce au niveau 5, et d'avoir pu acquérir des connaissances techniques sur la Blockchain Ethereum. C'était également la première fois que j'utilisais Volatility ou que je reversais du Rust. Au final, bien ce soit le 8ème challenge SSTIC que je termine (le temps passe si vite !), j'ai pu explorer des domaines que je ne connaissais pas et apprendre beaucoup de nouvelles choses. Et le plaisir d'arriver au bout du challenge est toujours intact.

Bon, je n'irai pas jusqu'à dire que j'ai aimé la multiplication des épreuves de maths, puisque j'ai souffert de mes lacunes. Cependant, je tire une certaine satisfaction d'avoir affronté ma Némésis, de m'être confronté à mes traumatismes de Terminale, et de m'en être sorti malgré tout. Et maintenant que je comprends un peu mieux la notation mathématique, lire des papiers remplis d'équations me donne (un peu, vraiment un tout petit peu) moins la nausée.

Bref, c'est un challenge que j'ai trouvé bien conçu et instructif. Un fidèle successeur des éditions précédentes. Merci beaucoup à Nicolas Iooss et Vincent Dehors de l'avoir conçu !

En cette période de confinement si particulière, pouvoir focaliser mon attention sur la résolution d'un challenge SSTIC avait quelque chose de rassurant. Cependant, c'était un vrai défi de réussir à dégager du temps pour le challenge sans trop perturber l'école à la maison et la vie de famille. Merci donc à elles [27] pour leur patience !

Pour conclure, je tenais à vous dire ceci :

Merci pour tout et à l'année prochaine !

27. D'ailleurs, il y aura bientôt une *elle* de plus ! \o/