Lab 2 – Buffer Overflow Attack lab

Report

**Due: Wed, Oct 13th, 2:30 pm.**

**Turn in: this report**

**Points: 100 pts**

**Section 1: Write a detailed report about the Buffer Overflow Attack lab. (60 points)**

* (40 points) Include step by step screenshots and explanations to show your exploit work.

We start the lab by disabling these countermeasures:



This command disables randomization of the starting address of the heap and stack, making it easy to know the addresses we need.

When compiling our programs, we will use the flag **-fno-stack-protector** to disable Stack-Guard, another precaution.

Also, when compiling we use **-z execstack -o** to make the program stack executable.

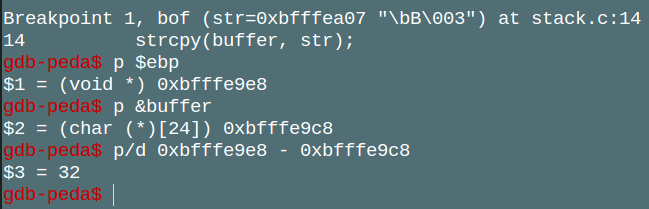
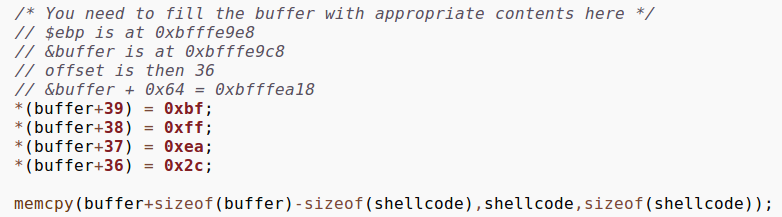
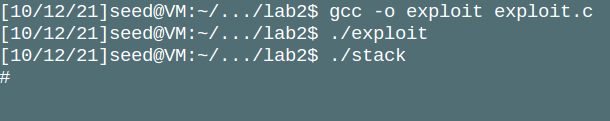
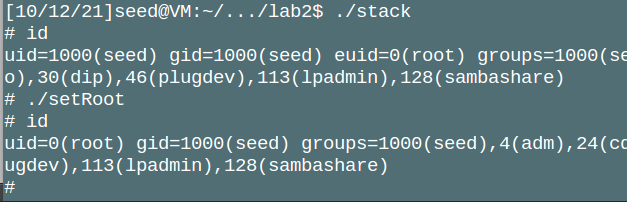
Finally, we need to link /bin/sh to /bin/zsh, instead of /bin/dash. /bin/dash includes countermeasures that we don’t want to deal with just yet.

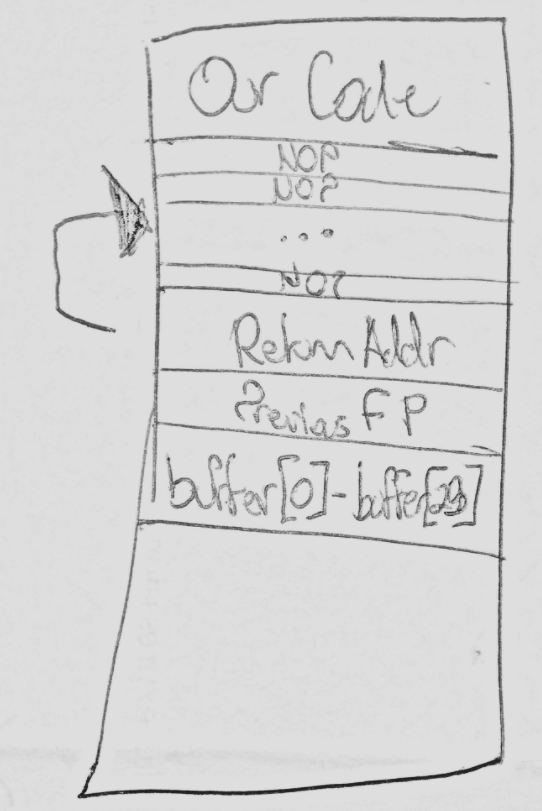


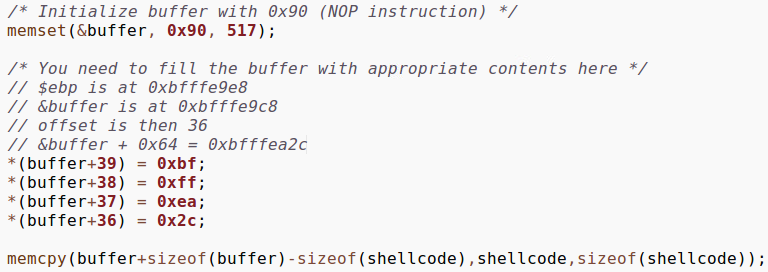
Next, we compile our progams:

1. call\_shellcode.c function using this command:
   1. gcc -z execstack -o call\_shellcode call\_shellcode.c
2. stack.c
   1. gcc -z execstack -o stack -fno-stack-protector stack.c
   2. sudo chown root stack
   3. sudo chmod 4755 stack

In the exploit.c program, we need to figure out how to modify the buffer to exploit the buffer overflow vulnerability. We can use gdb to help here. So, we need to recompile stack.c with the additional -g flag: ‘gcc -z execstack -fno-stack-protector -g -o stack\_dbg stack.c’

1. Start gdb
   1. gdb stack-db
2. Set breakpoint at bof
   1. b bof
3. Run the program
   1. run
4. Use p &buffer and p $ebp to find the position of the buffer and of EBP.
   1. Results: 
   2. So, our distance is 32+4 (to account for frame pointer) = 36.
   3. 
   4. This means our target location is 36 more than the buffer address.
5. Compile and run exploit.c, then run stack:
6. 
7. If successful, you should be in a root shell, as shown above. However, your user id is still the same. To change this, we compile this simple program:
   1. 
   2. This sets the id to root:
   3. 

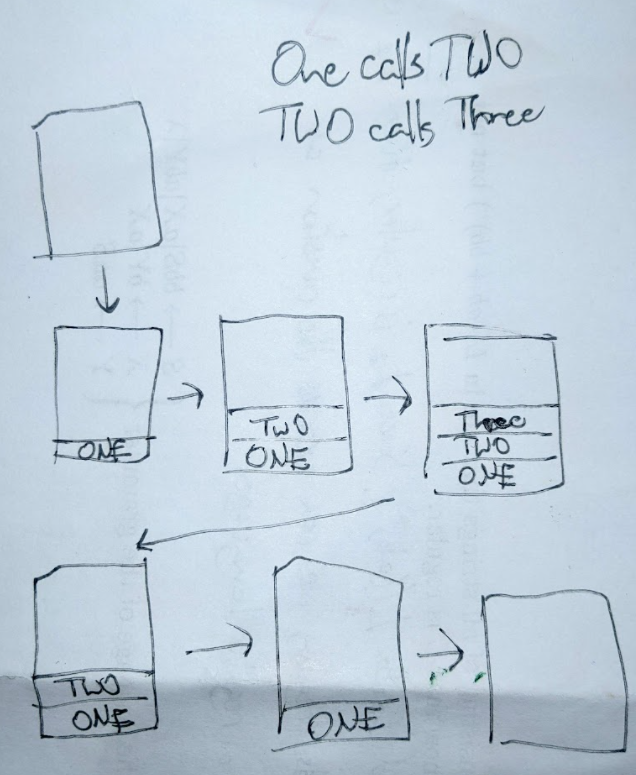
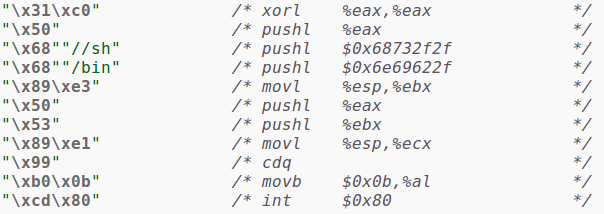
* (10 points) Include illustrations of the stack to show what you did.
  + 
  + We overwrote the bounds of the buffer array, overwriting the Return Address with a new pointer to a list of NOPs, which the computer then runs until it hits our malicious code.
* (10 points) Include the listing of your code and give a line-by-line explanation of what it does and how it works.
  + I’m not entirely sure what the listing of my code means, I assume it means the code that I wrote in the exploit function.



First, the memset function fills the buffer with nops. This will make it easier to run our malicious code, as we don’t have to get its exact address, just an address somewhere in the nops.

Next, we set the value at the buffer+36,37,38, and 39 to the address bfffea2c. This is an address to a NOP in the buffer. Then, when the program tries to run bfffea2c, it runs NOPs until it hits our malicious code.

**Section 2: Answer these questions in your own words. (30 points)**

* Define buffer overflow. What are the two key elements that must be identified in order to implement a buffer overflow?
  + A buffer overflow is an attack that makes use of functions that don’t check length when writing to a buffer. We need to know the address in memory that stores the Return Address, as well as the starting point for the malicious code. Ideally, you can debug the program to find the addresses, but there are even ways you can guess the addresses in many cases.
* Define stack frame, what are included in a stack frame.
  + The is a collection of data needed to perform a function. It contains the return address, as well as arguments and local variables.
* Illustrate the changes in a stack when a function is called.
* 
* In this example, function ONE calls TWO, then TWO calls Three. This is a Last-In-First-Out (LIFO) structure, as function ONE can’t complete until TWO finishes, and TWO can’t complete until three finishes.
* Define shellcode.
  + Shellcode is the Assembly code of code we want to inject with our attack. In our case, it was the code that started a root shell:
  + 
* Define what a NOP instruction is and how it is used in a buffer overflow attack.
  + A NOP instruction just tells the CPU to do nothing for that cycle, then move on to the next instruction. This is helpful in a buffer overflow attack, as you can append a bunch of them before your malicious code. Then, when setting the return address, you don’t have to get it *exactly* to where your code (that is, shellcode) is, you just have to get the address of one any of the NOPs.
* List two defenses against buffer overflows and explain how they work.
  + Check the length of the users input before writing to a buffer. If user input cannot be longer than the buffer, then they cannot write outside the bounds of the buffer.
  + Address randomization: Change where the stack and heap are located in memory at runtime. This way, it will be harder for the attacker to find the memory addresses they need to perform the attack.

**Section 3: Conclusion (10 points)**

What did you learn from the lab?

This lab has reinforced the idea of treating all user input as malicious. If there is a way to exploit your code for some personal gain, there’s a good chance someone will find it.

I learned what shellcode is in this lab. That was really interesting to me, that you can manually input assembly instructions as an array and with some creative code the computer will run it.

One thing that I’m curious about is how hard it would actually be to get this attack to work in a real-world scenario. In our lab, we had to disable multiple security features before we were able to actually perform the buffer overflow attack.