Introduction to Stack Overflows on x86

Author: Yusef Karim
Instructor: Jacques Béland
Department of Computing & Information Systems

May 12, 2019

Contents

1 Purpose **1**

2 Learning a bit or two of assembly **1**

3 Introduction to stack overflows **2**
3.1 Exploiting overflow.c 2
3.2 Exploiting serial.c 4

4 Fixing the security flaws **6**
4.1 Fixing overflow.c 6
4.2 Fixing serial.c 6

5 Conclusions **6**

6 References **6**

1 Purpose

In this lab we will be reading chapter 1, and 2 of the course textbook. This will serve as an introduction to 32-bit x86 exploits and the various tools we will be using through out the course. All progress shown here will be within the Ubuntu 16.04 virtual machine setup in the previous lab.

2 Learning a bit or two of assembly

To begin, chapter 1 of the course text gives us a quick introduction to what is to expect, and a little snippet of code that we can look at to better understand assembly. Before that you should get the source code provided by the authors by running the command:

```
wget https://media.wiley.com/product_ancillary/3X/04700802/DOWNLOAD/080238%20code.zip
```

Now that we have the source code we can go into the ch01 directory and use gcc to compile our code, specify the -c command to tell gcc to only compile the current code and not link anything, then use gdb as shown below to disassemble the C code to see what it looks like in assembly.

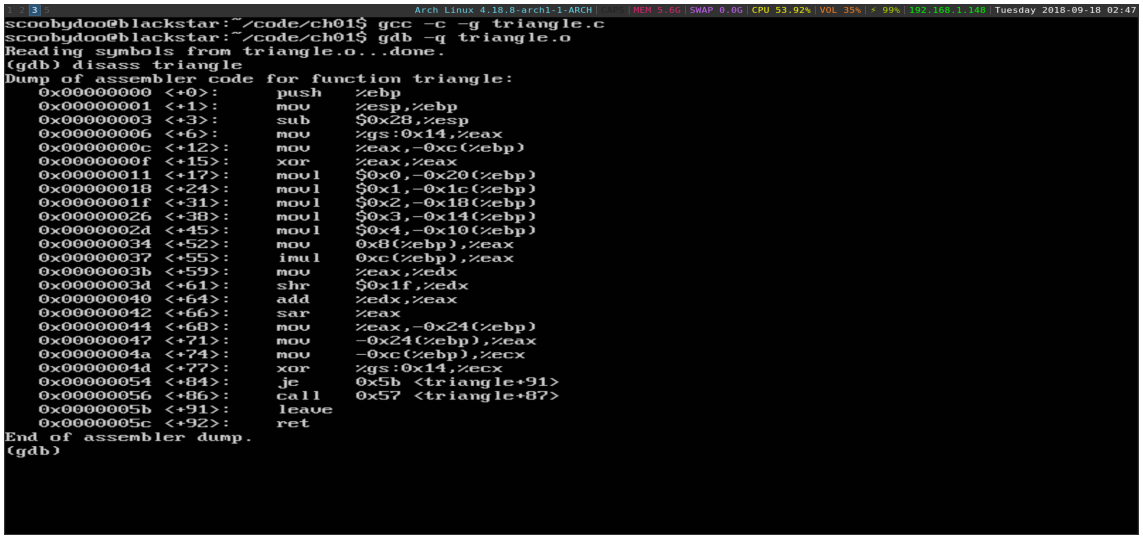

Figure 1: Compiling (without linking) then disassembling triangle.c

With this example, we can see how a C function actually looks in assembly. First, the *ebp* register (frame pointer) is pushed onto the stack so it can be used within the function then popped off the stack (via the *leave* instruction) after the function returns (a local function variable!). Next, room on the stack is made for local variables (width and height) by subtracting from the *esp* register (stacks grow downwards!). We see the array being initialized and values being stored in it with the *movl* instructions, we see the memory locations are being referenced by offsets with width 4 from *ebp*. We then see the area of the triangle being calculated with instructions like *imul*. This value is stored in the *eax* register, which is used to hold the functions return value. Finally, *ebp* is popped off the stack by the *leave* instruction and the function returns to the caller.

3 Introduction to stack overflows

Chapter 2 starts off by explaining what a stack overflow is then gives us a few examples to try out. In this section I will explain how I went about exploiting the *overflow.c* and *serial.c* programs through their buffer overflow vulnerabilities. Throughout this section I will be using Python scripts to help automate and understand the exploits I am carrying out, their will be directories including all the scripts I write and the vulnerable programs they pertain too.

3.1 Exploiting overflow.c

To start, we can compile *overflow.c* with the options described in 1ab 1.

```
gcc -fno-stack-protector -mpreferred-stack-boundary=2 -z execstack -ggdb
overflow.c -o overflow.out
```

When running *overflow.c* we see all it does is take user input then display it back to STDOUT. The flaw in this program is that it uses the *gets* function with a fixed sized array, meaning if user input is too long, the buffer will overflow and overwrite memory beyond it. The book holds our hand on how we can dig into this and eventually exploit this bug, I will briefly explain what I did when following the steps in the book. To begin, let's use gdb on the compiled executable so we can poke around:

```
gdb -q ./overflow.out
```

We know the vulnerability is in the *return_input* function so we can start by disassembling it. We can see the call instruction to the *gets* function, which is where it will ask for user input. To see the effects of our user input (especially when our input is larger than the buffer size) we can put a breakpoint right on the *gets* function and another one after it on the *ret* instruction (just before we return execution to the main function). We can then run our code, which will go until our first breakpoint is hit, as shown in Figure 2 below.

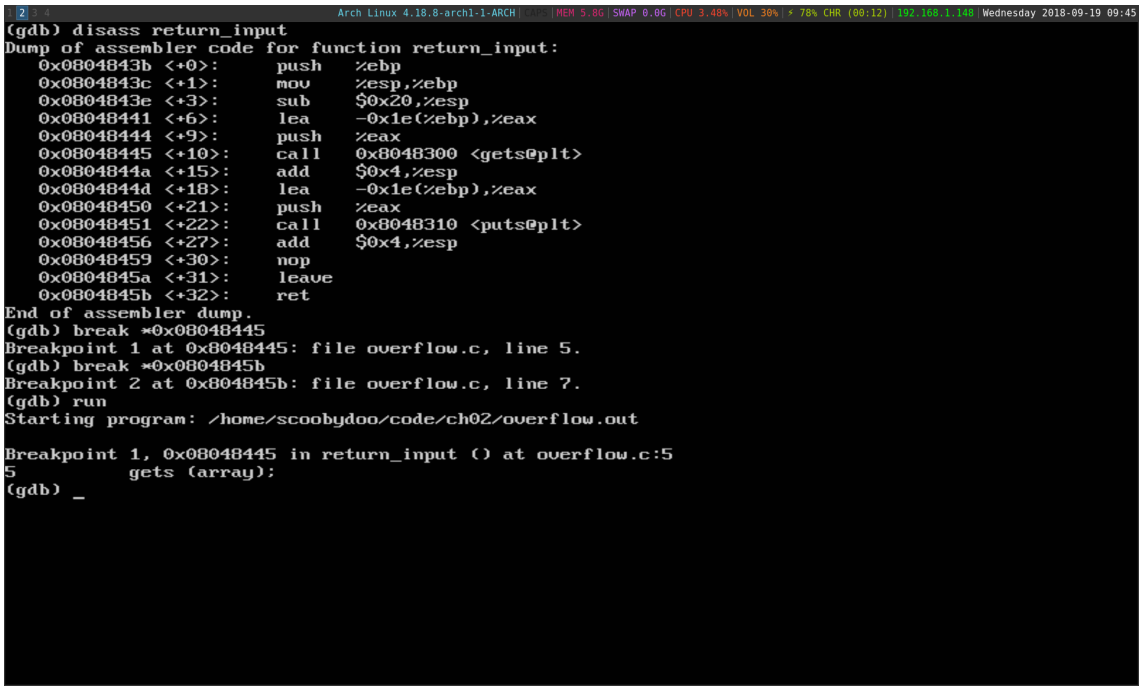

Figure 2: Disassembly of the return_input function in overflow.c

Our goal is to see the effect of user input on our registers, we know after we type our input and move onto our next breakpoint the function will return by referencing the address stored in the *EIP* register. By disassembling main we see the next instruction after returning (what *EIP* will be pointing to) is at address *0x08048465*, this value is also shown on the third row of values when printing the first 20 values beyond the current *ESP* address. Finally, if we give user input that is too long as shown below in Figure 3, we notice that the value stored in *EIP* has now been overwritten with the character's D (hexadecimal 0x44) due to a buffer overflow.

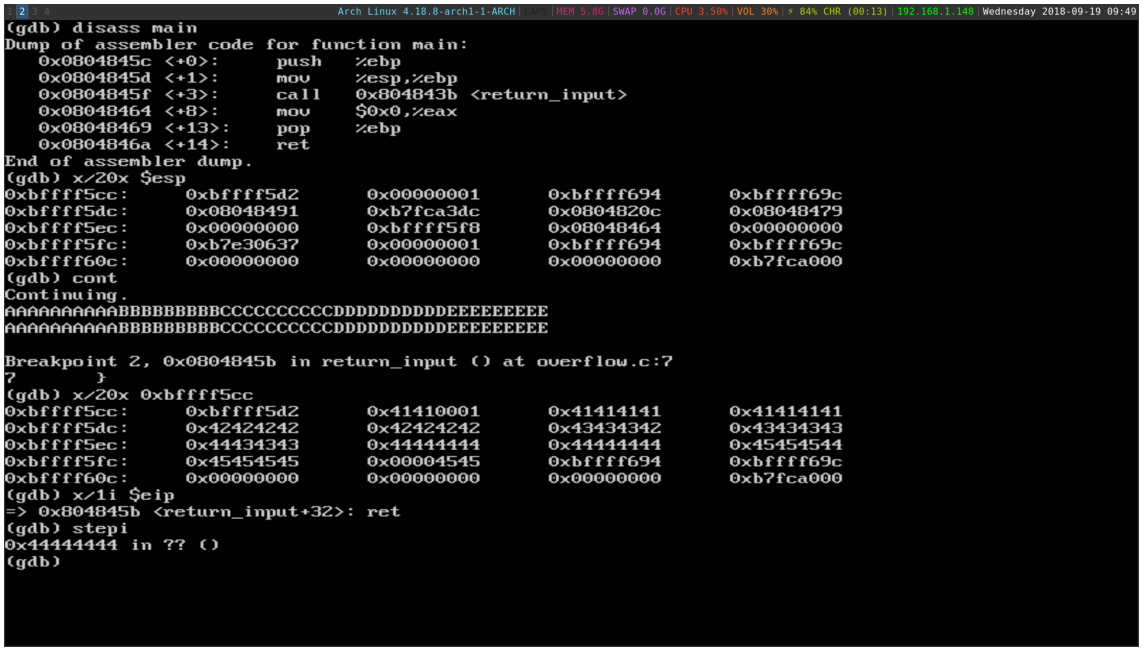

Figure 3: Finding the right amount of padding to overwrite/overflow *EIP*

In fact, if we count the number of characters, we can see that it took exactly 34 characters before our *EIP* register started getting overwritten by D's. To test this theory, the book recommends trying to store the address of the *return_input* function to see if we can get *EIP* (the return address) to call *return_input* for a second time. We can do this by putting the memory address of *return_input* at the end of our 34 bit padding. Looking back at Figure 3, we see the address of *return_input* is *0x0804845f* or `\x5f\x84\x04\x08` as little endian escaped bytes.

To carry out this exploit, a Python script called *exploit.py* was written, please refer to it for how the exploits were carried out for this section as well as section 3.2.

3.2 Exploiting serial.c

This is very similar to the above example, this time around we want to try and trick the program into thinking we put in a valid serial number by exploting a buffer overflow. Instead of looking for the exact amount of padding we need, we can make an estimate then use *exploit.py* to try multiple values for us. This saves a huge amount of effort and is very easy to carry out. First, compile *serial.c*:

```
gcc -fno-stack-protector -mpreferred-stack-boundary=2 -z execstack -ggdb
serial.c -o serial.out
```

We can then use gdb to disassemble the main function to find the address we want *EIP* to point to. Shown in Figure 4, the address we want to jump to is *0x08048601* (the *do_valid_stuff* function).

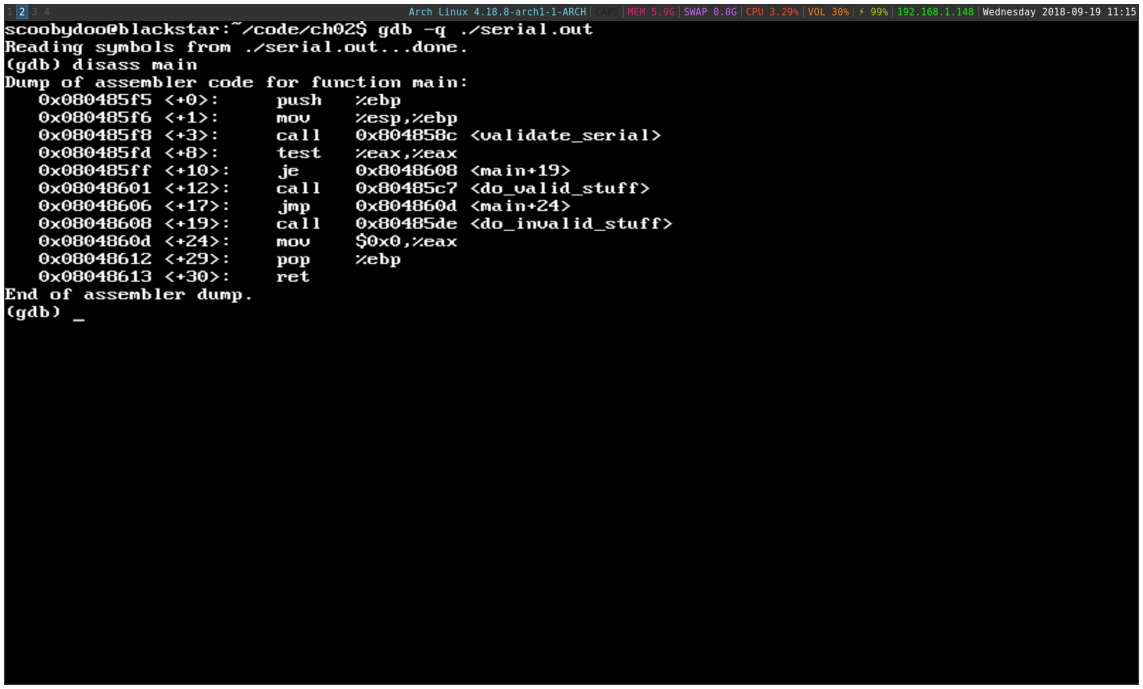

Figure 4: Finding the right amount of padding to overwrite/overflow *EIP*

We can use the function in *exploit.py* and a simple loop to find the correct padding to overwrite *EIP* successfully, shown below in Figure 5.

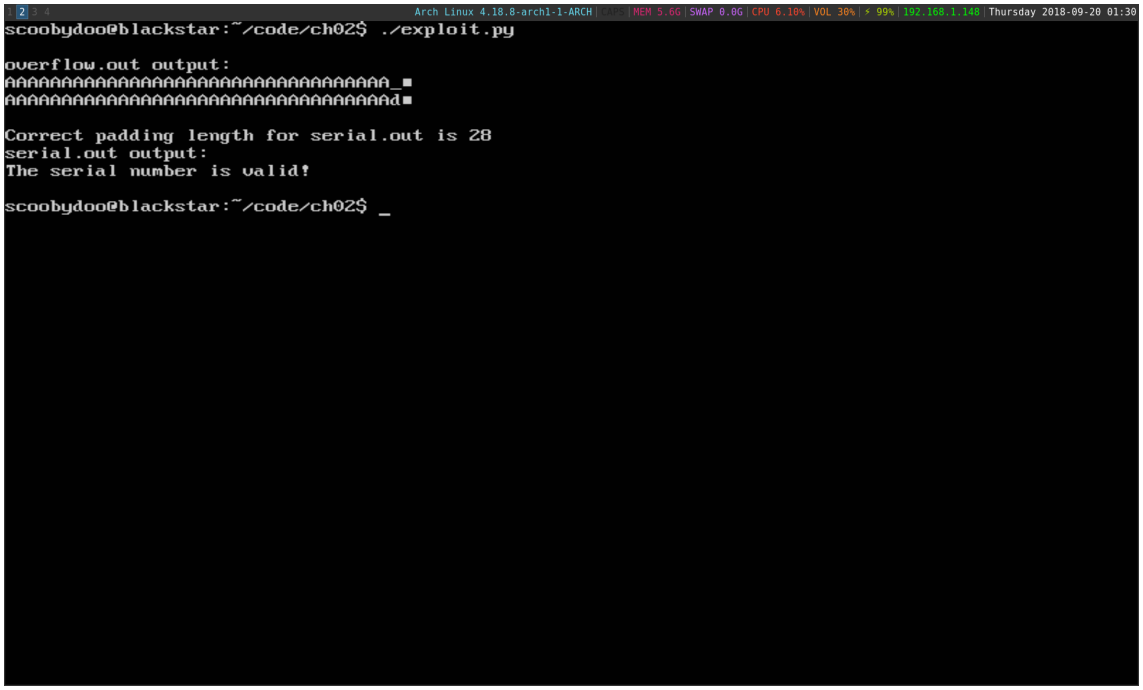

Figure 5: Bruteforcing the correct amount of padding to overwrite/overflow *EIP*

4 Fixing the security flaws

4.1 Fixing overflow.c

The exploitable piece of code in *overflow.c* is the *gets* function. Using the standard library function *gets* is now considered bad practice (reasons why shown above), it is now much safer to use the *fgets* function. This function makes sure you specify the size of the buffer you are storing data into. To fix *overflow.c* simply replace *gets* with a proper call of *fgets*:

```
fgets(array, sizeof(array), stdin);
```

4.2 Fixing serial.c

Fixing *serial.c* works in the same way as fixing *overflow.c* with one minor addition. In *serial.c* the *fscanf* function is used, which suffers the same bound checking problem as *gets*. This means we can fix the problem in the same way by replacing *fscanf* with *fgets*, but the problem with this is, *fgets* adds a newline character ('\n') to the end of the buffer, always making our serial number invalid. To fix this we can still call *fgets* but use the function *strcspn* to span the buffer and replace the newline character with a null value [3]:

```
fgets(serial, sizeof(serial), stdin);
serial(strcspn(serial, "\n")) = 0;
```

5 Conclusions

We now know a little more about x86 assembly as well as the basics of what a stack overflow is and how to exploit them. We have created a simple Python script to help exploit stack overflows in a more automated fashion and can extend it in the next labs, when we learn to create and execute useful shellcode.

6 References

1. **Python subprocess documentation**
   https://docs.python.org/3/library/subprocess.html#subprocess.Popen

2. **Smashing The Stack For Fun And Profit**
   http://phrack.org/issues/49/14.html#article

3. **Removing trailing newline character from fgets() input**
   https://stackoverflow.com/questions/2693776/removing-trailing-newline-character-from-fgets-input