**Intro To Exploitation - SimpleOverflow Solution**

**Introduction:** SimpleOverflow is an easy level exploitation challenge intended to introduce students to the concept of buffer overflow exploits; As well as the potential dangers involved with not verifying the size of a user's input against the size of the buffer that input will be stored in. Buffer overflow exploits also can potentially allow an attacker to execute arbitrary code, modify values of variables within a program, or jump to functions within a program that are otherwise not executed.

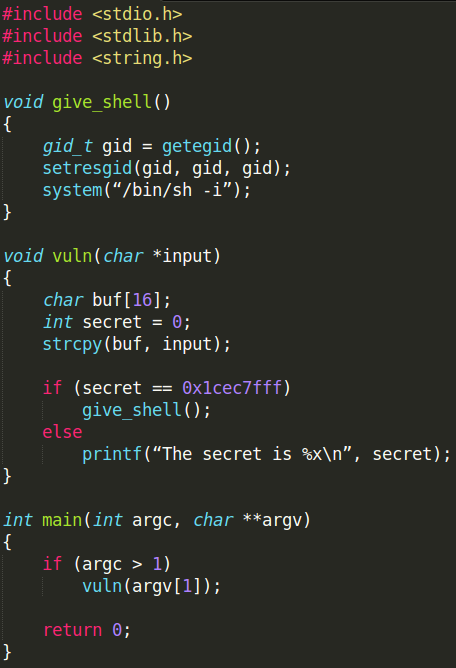
**Challenge:** The provided program is vulnerable to a buffer overflow exploit. Modify the value stored in the ‘secret’ variable to the required value to execute the give\_shell() function by executing a buffer overflow to retrieve the flag.

**Hints:** [**Wikipedia knows about buffer overflows, do you?**](https://en.wikipedia.org/wiki/Buffer_overflow)

**Solving:** Buffer overflow refers to any case in which a program writes beyond the end of memory allocated for any buffer. For example, if you have a program that can only store 250 characters of user input but the user is allowed to enter a 300 character string the program is vulnerable to a buffer overflow exploit. The extra 50 characters entered by the user will begin to overwrite adjacent memory values outside of the buffer.

In this challenge you are given access to a shell with standard user privileges (not root). You are expected to execute a buffer overflow against the running program and gain root privileges to retrieve the flag on the remote server. This writeup will execute the binary locally, but the steps are still the same.

Let’s begin by analyzing ‘overflow.c’ and find the vulnerability in the program.



We can see from the above source code that the main function of the program calls vuln() with the second command line argument (argv[1]) as a parameter. The vuln() function contains a character buffer of length 16 bytes (char buf[16]) which is allocated on the stack when the function is called. There is also an integer value (int secret = 0) allocated on the stack directly after the character buffer. Finally there is a call to a vulnerable function strcpy() (aka: string copy). Strcpy() function is responsible for copying our input into the 16 byte buffer on the stack. This function is vulnerable because it does not have any form of bounds checking. Meaning we could provide input larger than 16 bytes and the program will not raise an exception.

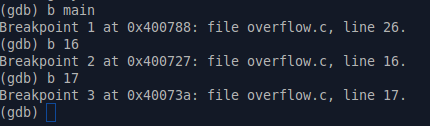
When we provide an input larger than 16 bytes everything beyond the 16 bytes will begin to overwrite memory values adjacent to the buffer on the stack. Since the integer variable ‘secret’ is declared directly after the ‘buf[16]’ variable these two variables are adjacent to each other on the stack. Therefore the first thing overwritten when our buffer overflows is the value contained by ‘secret’.

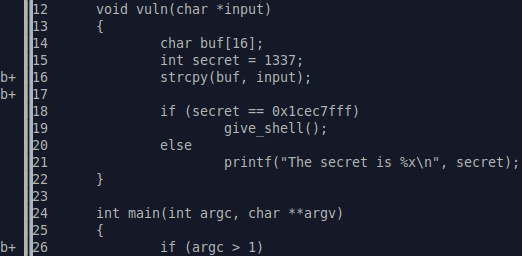
Let me show you using GDB. For illustrative purposes I’m also going to modify the initial value in the secret variable to the integer 1337. However, I will change the initial value back to 0 before we begin exploiting the program. Compile the program using the following command ‘gcc -m32 source.c -o outfilename execstack-fno-stack-protector -g -O0’

Once the program is compiled open the program inside of the GDB Debugger, I prefer to use the text UI myself ‘gdb -tui ./programToExecute’. Inside of GDB you can navigate the text UI pane by using either the arrow keys, or the page up / page down keys. During the program's execution it would be nice to halt the program at certain points to monitor our input on the stack.

This is achieved by setting a ‘breakpoint’ at a desired line of code ‘b LineofCode’ or an address of a specific assembly instruction ‘b \*0xADDRESS’ inside of GDB. Since we’re interesting in seeing our input on the stack as well as the secret value we want to set a breakpoint directly after the strcpy() instruction is executed. Personally, I also like to put a breakpoint on main() so the program halts immediately following execution ‘b main’

In total I have three breakpoints, one at main, line 16, and line 17. As shown below.





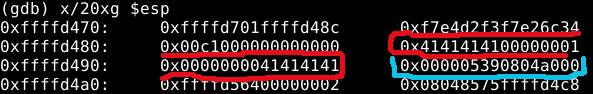
Once the breakpoints are set we can go ahead and execute the program inside of GDB. To do this we type the command ‘run’ combined with some input since the program does take our input afterall. The actual command should look something like: ‘run ThisIsMyInput’. However, I will be using python to pass my input to the program using python to pass input makes sending larger inputs to programs very easy, it’s also very easy to use.

Additionally, I don’t want to just send any random 8 character string into the program. It’s smart to use input that is easily identifiable on the stack. For example, the ASCII character ‘A’ when placed on the stack is represented by the value ‘41’ passing 8 ‘A’ characters will make identifying our input on the stack very easy.

Execute the program in gdb using the following command: ‘run $(python -c ‘print “A” \* 8’)’. When the program begins execution you will hit the breakpoint set at main immediately. You can continue execution of the program a single instruction at a time using the ‘step’ command. You can also continue execution until the next breakpoint is hit by using the ‘continue’ command. I’ll be using the ladder here.

Continue the program execution until you hit your last breakpoint at line 17.

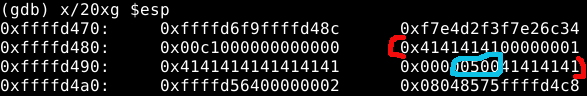
Once the program pauses execution at breakpoint 17 it’s time to examine the contents of the stack. This can be done by using the following command: ‘x/20xg $esp’. This command stands for “Examine 20 large words starting at the address pointed to by $esp”. $esp is a special register in the processor responsible for keeping track of the top of the current stack. The contents of the stack is below.



From the screenshot above we can see the stack begins at address ‘0xfffd470’. The 8 ‘A’s we passed to the program as input begin at address ‘0xffffd480’ and are outlined in red. Finally, the value of secret is outlined in blue. Since data and addresses are stored on the stack in hexidecimal value the decimal value 1337 becomes ‘0x539’ you can use a simple online calculator to confirm this.

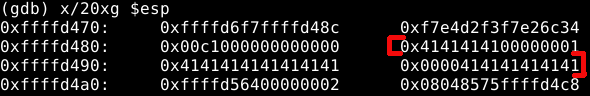
Now that we’re familiar with how our input is stored on the stack. Let’s see what happens when we pass an input that is bigger than 8 bytes. Let’s try an input of 16 bytes. Repeat the previous steps using an input that is 16 bytes long and examine the contents of the stack. There’s no need to restart GDB or anything like that, just type the run command again and GDB will ask if you wish to restart the program.

‘Run $(python -c ‘print “A” \* 16’)’

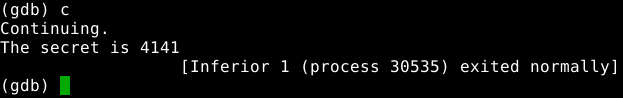


Interesting, It seems we’ve managed to modify the original ‘539’ value to ‘500’, which is good. This means we’re on the right track to exploiting this executable. Next, let’s try to pass two more additional ‘A’ characters with our input. This will bring our total input size up to 18 bytes, beyond what our buffer can hold.

‘Run $(python -c ‘print “A” \* 18’)’



After passing 18 bytes of input to the program, it seems we’ve completely overwritten the value stored in the address previously. Let’s see what happens if we continue to run the program like this.



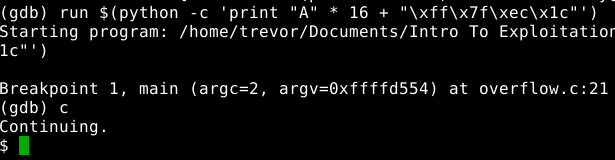
Perfect! We’ve successfully managed to overwrite the value of ‘secret’ with our input, exactly what we need. Now all we need to do is change the value of secret by passing the required value with our overflow. Looking at the programs code again the value of secret needs to be equal to ‘0x1cec7fff’ to execute the give\_shell() function.

The last piece of information you must know to complete this exploit is that hexadecimal values are placed on the stack in little endian order, in other words backwards.

Thus, we can overflow buf with 16 bytes and add 0x1cec7fff in little-endian to the end of our input. Thus, buf[16] will be set, and the remaining 4 bytes of 0x1cec7fff will be set to secret. In the end, this is our final payload:

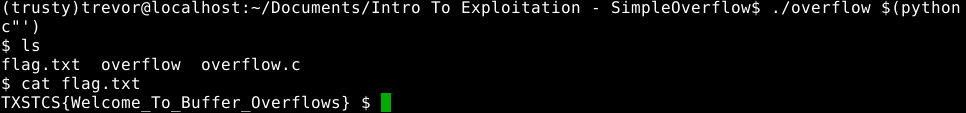
‘Run $(python -c ‘print “A” \* 16 + “\xff\x7f\xec\x1c”’)’

Run the program one last time in GDB and see what happens!



Prefect! If you successfully executed the exploit in GDB you should notice GDB seems to lock up and present us a ‘$’. This is because our original program executed a new program, a bash shell!.

Quit out of GDB and run the exploit against the program outside of GDB with the same payload as before. You may have to close your terminal session if you are unable to break out of GDB with the ‘CTRL + C’ command.



Congratulations, you now know how to execute simple buffer overflows.