**Intro To Exploitation - Smash Solution**

**Introduction:** Smash is a medium level exploitation challenge intended to improve students knowledge and understanding of buffer overflow exploits and 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.

**Task:** The provided program is vulnerable to a buffer overflow exploit. Execute a buffer overflow and jump to the flag() function to retrieve the flag. This task requires you to have basic GDB knowledge.

**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 256 characters of user input but the user is allowed to enter a 300 character string the program is vulnerable to a buffer overflow exploit.

Executing a basic buffer overflow exploit usually consists of roughly three or so steps.

1. Trigger the overflow
2. Control the Return Instruction Pointer (AKA: $rip)
3. Jump to the user controlled buffer
4. Execute arbitrary code (if applicable)

Today we won’t be executing any arbitrary code, but simply causing the program to execute a function that otherwise isn’t called within the program.

I placed both a copy of the program’s source code, and a pre-compiled binary for your use. However, you’re free to compile your own binary. The proper way to compile a binary so it’s vulnerable to basic buffer overflow exploits is: ‘gcc -m64 source.c -o outfilename -z execstack -fno-stack-protector -g’ this will compile the program without any basic protection against these exploits.

If you are testing this exploit locally you will also need to temporarily disable Address Space Layout Randomization protection. On a Ubuntu based machine This can be done with the following command: ‘echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space’. ASLR protection will automatically be re enabled on your system following a reboot.

Finally, please note that the addresses given to you by gdb on your machine may differ from the addresses in the screenshots. Understand though that the process is still the same. Let’s begin!

First thing’s first, let’s examine the provided source code and learn what exactly makes this program vulnerable to a buffer overflow exploit.

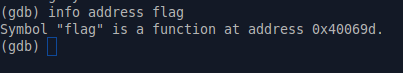


The program itself is very simple. Upon execution the program checks to see if two or more command line arguments are passed, and also checks to see if the second argument begins with the string “Smash\_The\_Stack\_For\_Fun\_And\_Profit”. If these conditions are met the program will call the bof() function, passing the user's input to the function. Otherwise the program will exit.

When the bof() function is called the input passed to the function is copied via the strcpy() function to a buffer that can contain up to 150 characters. If you notice there is no check to ensure that the user’s input will fit within the buffer. Due to this missing check, this program is vulnerable to buffer overflow exploits.

Finally, there’s a flag function that’s never called. Let’s see if we can’t change that.

Go ahead and open up the binary in the GDB Debugger. One of the first things we can do is find the address of the flag() function within the program. We’ll need this address to jump to it later. You can find the address of any variable or function within a program by using the command ‘info address functionName’ or ‘info address variableName’.

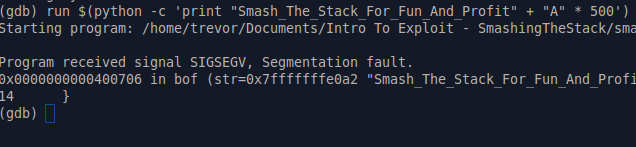


GDB happily reports that the flag function is located at address ‘0x40069d’ within the program. Make note of this address, you will need it later.

The next step in this process is to verify that our target program is in fact vulnerable to a buffer overflow by providing input to the program that incredibly large. The easiest way to do this is to send an incredibly large amount of characters to the program. Normally you would manually have to type large inputs to the program. However, with some Python magic we can simplify this process.

Remember that the input passed must begin with the string mentioned above. Afterwards you’re free to send the program whatever you like. Most people like to use large strings of characters with easily identifiable ASCII codes. For instance the character ‘A’ has the ASCII code of ‘0x41’.

So in summary we want to run this program in GDB and pass the input “Smash\_The\_Stack\_For\_Fun\_And\_Profit” and an arbitrarily long string of characters to the program. We can run the program in GDB and have Python print the input for us using the following command: ‘run $(python -c ‘print “Smash\_The\_Stack\_For\_Fun\_And\_Profit” + “A” \* 500’). Give it a try.



Perfect! The program took our input, then crashed because our input was too big for the buffer responsible for storing the input. This caused us to overwrite memory that didn’t belong to us. We’ve now confirmed this program is vulnerable to buffer overflows.

Let me take a moment here to explain in detail why the program crashed beyond “overwriting memory that doesn’t belong to us”. Inside of the CPU there’s a special register known as the ‘Return Instruction Pointer’ or $rip for short. This register is responsible for keeping track of the next instruction to be executed by the CPU via a memory address. When we pass an input too large for the buffer and cause an overflow we end up overwriting the address stored in this register causing the computer to try and execute an instruction that doesn’t exist.

What would happen though if we overwrote the address pointed to by $rip with our own valid address? Hypothetically it would be possible to control the execution flow of the program. The next step in this process is to learn how to craft our input such that we gain control over the Return Instruction Pointer.

This is done by first calculating the size of the buffer we have to write to before we begin overwriting the address pointed to by $rip. This is done by setting some breakpoints in the vulnerable function and monitoring the address pointed to by the $rsp register which always points to the top of the stack.

To make life easier for ourselves during this process open up GDB using the text ui layout this is done by opening gdb with the ‘-tui’ flag. Next we want to set GDB up to properly display the disassembly of our program. Execute the following commands within GDB: ‘set disassembly-flavor intel’ , ‘layout asm’. You can navigate the GDB text UI using the arrow keys on your keyboard or the page up / page down keys.

Next I want to set four breakpoints to monitor the execution flow of the program. One breakpoint at main, and three breakpoints within the bof() function shown below.

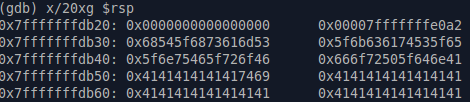
Screenshot - 08052016 - 01:01:10 AM.png

Run the program and pass the same input as we did previously. Continue execution until you land inside of the bof() function. Make note of the address pointed to by $rsp as you step through the instructions. We’re particularly concerned with the address pointed to by $rsp pre-leave instruction and the address pointed to by $rsp post-leave instruction.

We will use these two addresses to calculate the size of the buffer we need to overflow before we can control $rip. Examine the contents of the registers in GDB by using the command ‘info registers’

Screenshot - 08052016 - 01:09:47 AM.png

From the screenshot above pre-leave instruction the $rsp register points to the address “0x7fffffffdb20”. If you examine the contents of the stack after the strcpy call but before the leave instruction is executed you will see our input on the stack. Use the command ‘x/20xg $rsp’ to view the contents of the stack starting from the $rsp register.



Examining the contents of the stack we can clearly see our input stored on the stack. The top of the stack begins at address 0x7fffffffdb20 pointed to by $rsp. The string “Smash\_The Stack\_For\_Fun\_And\_Profit” begins with the ASCII values ‘7fffffffe0a2’, and ends at ‘0x666f72505f646e61’. Everything afterwards is all of the ‘A’ characters we passed to the program.

Post-leave instruction the $rsp register points to the address “0x7fffffffdbd8” as seen in the screenshot below.

Screenshot - 08052016 - 01:22:14 AM.png

The reason $rsp points to the new address ‘0x7fffffffdbd8’ after the leave instruction is executed is because the functionality of leave instruction. Which serves to prepare the program to return back to another function. In this case the main() function was responsible for calling the bof() function we’re currently inside. Once the leave instruction is executed, the computer prepares to return back to main() function by first collapsing the stack frame used by bof(), and then pointing $rsp to the top of another stack frame.

The first piece of data contained in this frame is the address of the next instruction to be executed upon returning to main().

We now have both addresses we need to calculate the total size of the buffer we need to overflow before we begin writing our own address into the $rip pointer. The formula for calculating buffer size (also known as offset) is:

This means we can pass 184 total bytes of data to the program before we begin overwriting memory that doesn’t belong to us. That should mean that our previous input of “Smash\_The\_Stack\_For\_Fun\_And\_Profit” + “A” \* 500 should have given us $rip control right? Unfortunately not, 0x414141414141 is not a valid memory address.

Besides that, on a 64 bit system the biggest address we can ever pass to the system is 6 bytes in length. Any address we provide beyond 6 bytes in length will raise an exception and cause the program to crash. This means if we pass 0x414141414141 as an address an exception will be raised, but the address 0x0000414141414141 is safe.

Lets test this by creating a new payload to send to the program. Remember our total payload must be 184 bytes in size before we begin to overwrite $rip. Each character is occupies one byte on the stack. Therefore the required string “Smash\_The\_Stack\_For\_Fun\_And\_Profit” is 34 bytes in length and ultimately we want to also provide a 6 byte address to jump to.

This means we need 144 bytes of filler characters before we begin to write to the desired location to control $rip.

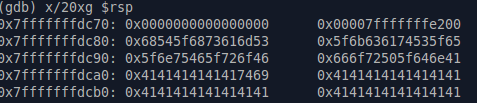
Something like: $(python -c ‘print “Smash\_The\_Stack\_For\_Fun\_And\_Profit” + “A” \* 144 + “B” \* 6’) Should work for us. Re-run the program with this new payload and follow the same steps as above in GDB.

Note: As you monitor the execution of the program with the new input you may notice the addresses pointed to by $rsp differ are different from the originals. Don’t worry, the offset is still the same!

Below are screenshots of the addresses contained by $rsp and the content of the stack pre-leave and post-leave instruction respectively.

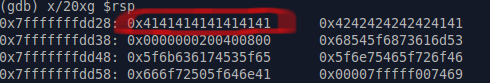
Pre-Leave:

Screenshot - 08062016 - 10:33:39 PM.png



Post-Leave:

Screenshot - 08062016 - 10:35:12 PM.png



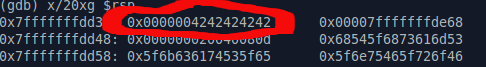
It seems our input is a little too large. If everything had gone according to plan the fake address, the 6 “B” characters we passed, represented by the ASCII value ‘0x42’ should be contained in the red box.

We must have calculated the number of filler characters required to overflow the buffer incorrectly, but how? If you’ve been attentive you may have noticed that pre-leave there are 11 bytes of filler before our input on the stack in the form of ‘0x0000000000000000’ and ‘0x0000’ before the input we passed to the program begins.

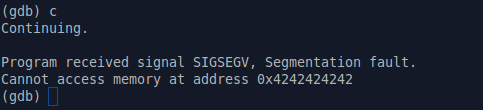
This means we need to subtract 11 filler characters from our payload to achieve the desired outcome. Our new payload is

$(python -c ‘print “Smash\_The\_Stack\_For\_Fun\_And\_Profit” + “A” \* 133 + “B” \* 6’).

Repeat the previous steps.

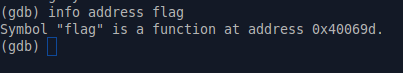


Perfect! Hypothetically we should now control $rip. Execute the return instruction in bof() and see what happens!



Success! When the return instruction was executed the computer tried to execute the next instruction pointed to by the address we provided: 0x424242424242 but couldn’t because there is no valid instruction contained at 0x424242424242.

But we do know the address of a valid instruction! The flag() function we’ve been wanting to jump to this entire time. What is the address of that function again?

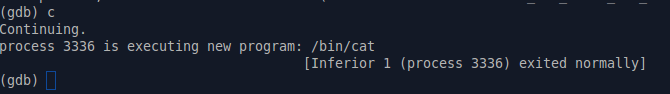


Oh right, 0x40069d. All that’s left to do now is update our payload with this address.

**One final note**: Addresses are placed into the stack backwards, this is known as ‘little-endian’ format. Additionally addresses are always in hexadecimal format meaning we will need to update our payload with the following string: \x9d\x06\x40. The ‘x’ indicates we’re passing a hex value to the program and the ‘\’s separate each byte.

Final Payload: $(python -c ‘print “Smash\_The\_Stack\_For\_Fun\_And\_Profit” + “A” \* 133 + “\x9d\x06\x40’)

Running the program with the above payload should result in the program jumping to the previously unexecuted flag() function.



Bingo! The program jumps to the provided address and executes /bin/cat exactly like we had hoped for. The last thing to do is quit GDB and execute the program normally passing the payload in as input.

Screenshot - 08062016 - 11:58:40 PM.png

**Note:** For some unknown reason I had an issue with the last byte of my address being dropped. To solve this problem I appended one extra byte of junk to my address knowing it would be cut off. The final address I passed was ‘\x99\x9d\x06\x04’. Try this method if you encounter a similar issue.