INDIAN INSTITUTE OF TECHNOLOGY DELHI

Course: ELL 783

Operating Systems

Assignment-3 (Easy) | 30 April 2023

BUFFER OVER FLOW ATTACK IN XV6

AND

ADDRESS SPACE LAYOUT RANDOMIZATION

Submitted by

Maj Damandeep Singh - 2022EET2805

Maj Gaurav Harjule - 2022EET2806

PART 1 : BUFFER OVER FLOW ATTACK IN XV6

Implementation

1. **Motivation** : A typical cyber attack known as a "buffer overflow attack" takes advantage of a vulnerability known as a "buffer overflow" where user-controlled data is written to memory. The attacker can overwrite data in other areas of memory by sending more data than can fit in the allotted memory block. Attackers may conduct buffer overflow assaults for a number of reasons, including overwriting crucial code or data to cause the program to crash, inserting malicious code to be executed by the program, or changing crucial values to alter the program's execution flow.

2. Buffer overflow attacks can be used to achieve various objectives, including Denial of Service (DoS) Attacks, Code execution, Access Control Bypasses etc. A buffer overflow attack can be performed in a few different ways, but some of the most common examples include stack based buffer overflow, heap based buffer overflow, format string attacks etc.

3. **Working** : When a function is invoked in a process, its stack frame is loaded onto the user stack. The local variables of the callee function make up the stack frame. The return address, a pointer to the base address of the caller function's stack frame, and the local variables of the caller function are pushed onto the stack. When writing data into the local variables of the callee function, the function strcpy() does not check bounds. Thus, it can be used to overwrite the return address to any desired address within the process' virtual address space. In the event that this attack is carried out while the process is running in the kernel mode, the entire system data can be easily accessed.

4. Apropos above for the assignment purpose, the steps followed to simulate the buffer overflow attack are as under:

   (a) Turn off the optimization and run the xv6.

   (b) Define function foo(). The address of user defined function foo()changes when we run xv6. It becomes 0x0000.

(c) We copied a null terminated string of size (12 plus buffer_size) to corrupt the return address via the vulnerable_function().

(d) The strcpy() function overwrote the return address's least significant byte with a 0.

(e) When the callee function returns & the address 0x0000 is entered into the esp register, the OS prints the SECRET_STRING before terminating the process with a panic.

(f) The Code snippets for the same are as under:

```
#include "types.h"
#include "user.h"
#include "fcntl.h"

void foo () {
      printf (1 , " SECRET_STRING " ) ;
}

void vulnerable_function ( char * input ) {
      char buffer [4];
      strcpy(buffer,input) ;
}

int main ( int argc , char ** argv ){
      int fd = open ("payload",O_RDWR) ;
      char payload [100];
      read (fd , payload , 100) ;
      vulnerable_function (payload) ;
      exit () ;
}
```

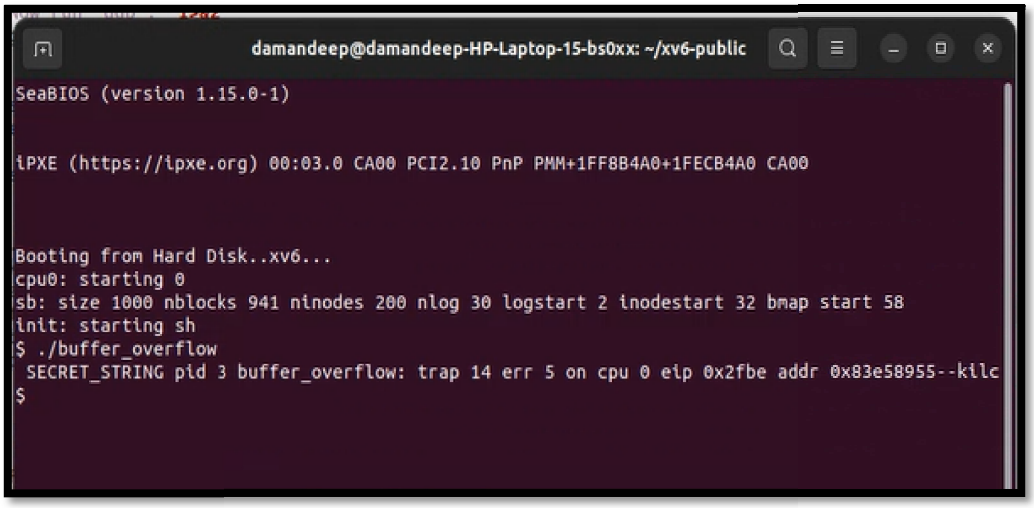
Snippet of buffer overflow.c

```
import sys

file1 = open("payload", "w")
length = int(sys.argv[1])
s = "x"
for i in range(0,length):
        file1.write(s)
file1.close()
```

Snippets of gen_exploit.py

Payload



Snippet of output

PART 2 : ADDRESS SPACE LAYOUT RANDOMIZATION

Implementation

5. **Motivation** : In order to make a buffer overflow attack that requires the attacker to know the location of an executable in memory, address space layout randomization (ASLR) is a technique. A buffer overflow vulnerability is a flaw in software written in a programming language that is not safe for memory, such as C. An application fails to verify the size of user input data that is written to memory. In order to fix this error, an application can check the length of the user input data and throw an exception or an error message if the actual length does not match the expected one. ASLR works by means of breaking assumptions that any developers could otherwise formulate about where programs and libraries would lie in memory at runtime. ASLR mixes the vulnerable process' address space (the main program, its dynamic libraries, the stack and heap, memory-mapped files, etc.) so that it exploit payloads must be specifically tailored to however the victim process' address space is laid out at the time.

6. **Working** : In the xv6 file *exec.c* is the default *loader*, which reads ELF header. Along with user stack and the heap, it then sets up the pages. In order to achieve the address randomization and prevent buffer overflow attack we must provide an offset. This confuses the attacker and would not be able to pinpoint at the address of return address in the stack. A random number generator would be needed to implement this. On the basis of this an offset will be created to ‘randomise’ the address space.

7. Apropos above the basic implementation steps are as under

   (a) We must ensure that the application, stack, and heap are loaded into truly random memory locations in order to minimize the possibility of an attacker “guessing” the correct memory address of critical memory sections.

   (b) In order to create a sequence of pseudo-random numbers, we employ the Linear Congruential Generator (LCG) algorithm. Apropos the random number generator function has been defined in exec.c.

(c) In proc.c changes were done in order to read the value of aslr_flag.txt file which is present in the xv6 folder. The value has been changed to 0.

(d) It was observed that the file has been able to successfully read, but the contents cannot be altered after the execution of make qemu as its image is stored during compilation. The value if require can be changed to 1 or vice versa only before execution of make.

(e) Modify the Make file for the flags.

(f) An Offset of pagesize x Random number(between 1 and 4) is added to the Virtual Address Space while loading the ELF file in virtual address space.

(g) The code snippets for the same are under.

```
/*Random Number Generator
Genertes Random number within a range*/

uint random(void)
{
  static unsigned int z1 = 12345, z2 = 12345, z3 = 12345, z4 = 12345;
  unsigned int b;
  b  = ((z1 << 6) ^ z1) >> 13;
  z1 = ((z1 & 4294967294U) << 18) ^ b;
  b  = ((z2 << 2) ^ z2) >> 27;
  z2 = ((z2 & 4294967288U) << 2) ^ b;
  b  = ((z3 << 13) ^ z3) >> 21;
  z3 = ((z3 & 4294967280U) << 7) ^ b;
  b  = ((z4 << 3) ^ z4) >> 12;
  z4 = ((z4 & 4294967168U) << 13) ^ b;

  return (z1 ^ z2 ^ z3 ^ z4) / 2;
}

// Return a random integer between a given range.
int
randomnumber(int lo, int hi)
{
  if (hi < lo) {
    int tmp = lo;
    lo = hi;
    hi = tmp;
  }
  int range = hi - lo + 1;
  return random() % (range) + lo;
}
```

Snippet of fn Random No Generator

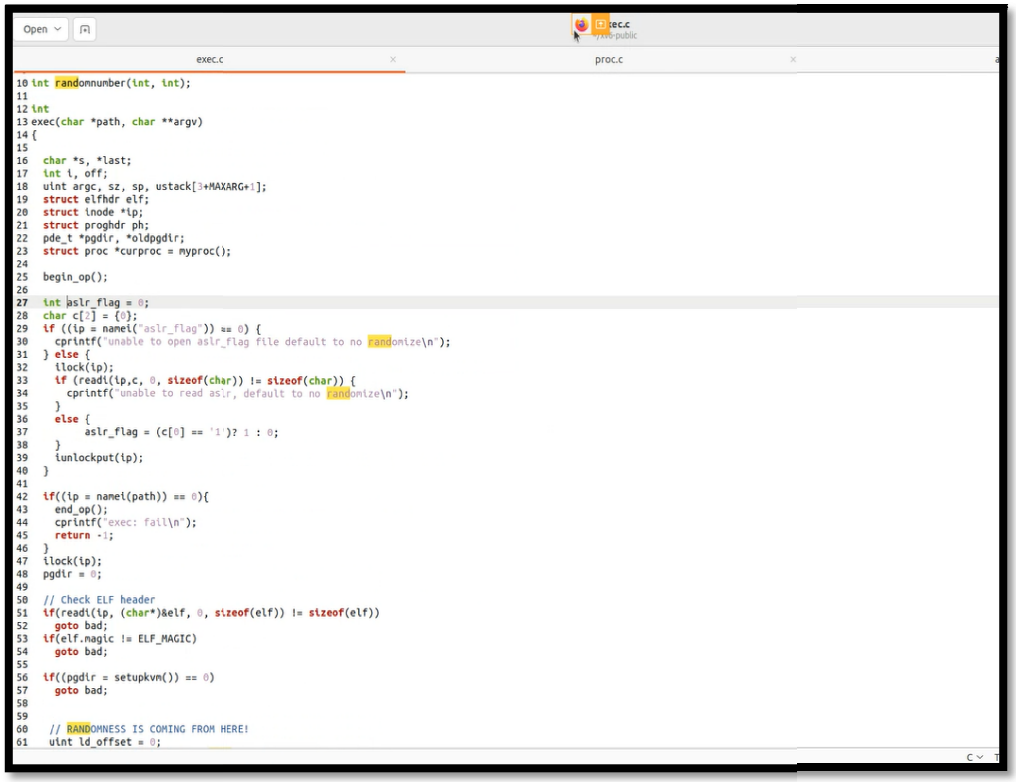
aslr_flag.txt

```
233 // Exit the current process.  Does not return.
234 // An exited process remains in the zombie state
235 // until its parent calls wait() to find out it exited.
236 void
237 exit(void)
238 {
239
240   pushcli();
241   mycpu()->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
242   mycpu()->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
243   popcli();
244   struct proc *curproc = myproc();
245   struct proc *p;
246   int fd;
247
248   if(curproc == initproc)
249     panic("init exiting");
250
251   // Close all open files.
252   for(fd = 0; fd < NOFILE; fd++){
253     if(curproc->ofile[fd]){
254       fileclose(curproc->ofile[fd]);
255       curproc->ofile[fd] = 0;
256     }
257   }
```

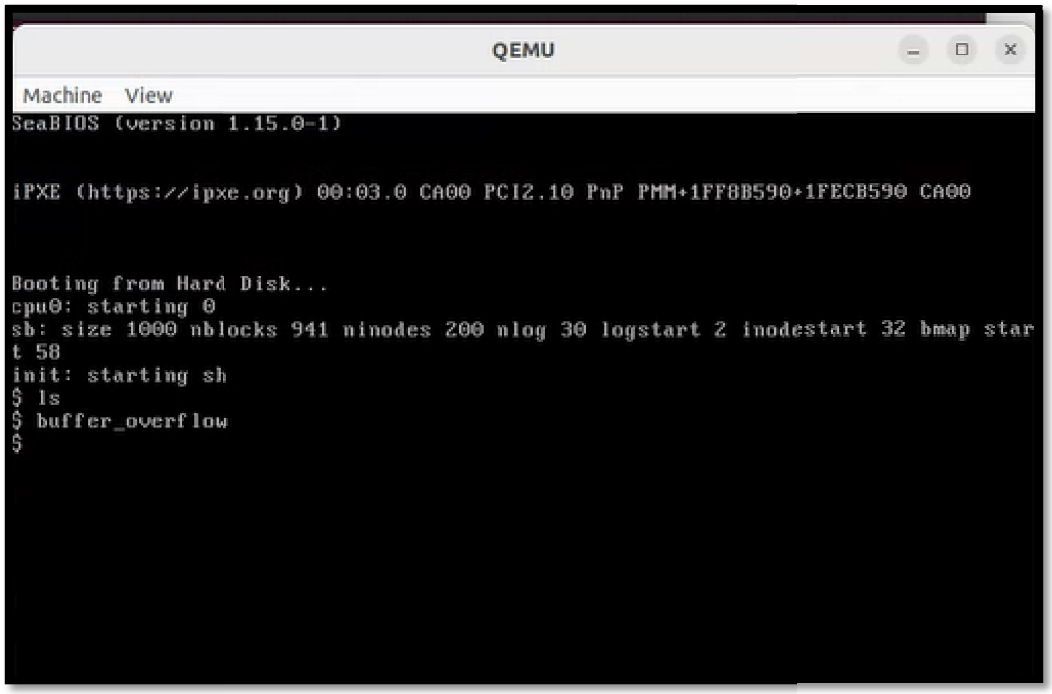
Snippet of changes in proc.c

```
59
60    // RANDOMNESS IS COMING FROM HERE!
61    uint ld_offset = 0;
62    //ld_offset = (aslr_flag)? randomnumber(1, 7) << 12 : 0;
63    if(aslr_flag){
64          ld_offset = randomnumber(1, 7) << 12;
65          if(curproc->pid == 1 || curproc->pid == 2) ld_offset = 0;
66    }
67
68    //cprintf("load offset: %d\n", ld_offset);
69
70   // Load program into memory.
71   sz = allocuvm(pgdir, 0, ld_offset);
72   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
73     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
74       goto bad;
75     if(ph.type != ELF_PROG_LOAD)
76       continue;
77     if(ph.memsz < ph.filesz)
78       goto bad;
79     if(ph.vaddr + ph.memsz < ph.vaddr)
80       goto bad;
81     if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz + ld_offset)) == 0)
82       goto bad;
83     if(ph.vaddr % PGSIZE != 0)
84       goto bad;
85     if(loaduvm(pgdir, (char*)(ph.vaddr + ld_offset), ip, ph.off, ph.filesz) < 0)
86       goto bad;
87   }
88   iunlockput(ip);
89   end_op();
90   ip = 0;
91
92   // Allocate two pages at the next page boundary.
93   // Make the first inaccessible.  Use the second as the user stack.
94   //int stack_offset = (aslr_flag)? randomrange(2, 1000) : 2;
95   sz = PGROUNDUP(sz);
96   if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
97     goto bad;
98   clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
99   sp = sz;
100
101  // Push argument strings, prepare rest of stack in ustack.
102  for(argc = 0; argv[argc]; argc++) {
103    if(argc >= MAXARG)
104      goto bad;
105    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
106    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
107      goto bad;
108    ustack[3+argc] = sp;
```

Snippet of changes in exec.c



Snippet of changes in exec.c



Snippet of the output