Indian Institute of Technology Delhi

Operating System

ELL 783

Assignment 3

Name: Ranjan Kumar Singha
Entry No: 2023EET2192

Name: Atul Kumar Rana
Entry No: 2023EET2195

Course Instructor: Dr. Smruti R. Sarangi

Contents

1 Buffer Overflow Attack in xv6

For the purpose of crafting a payload file intended for a buffer overflow attack, we initiated the process by composing the following Python script within the file named `gen_exploit.py`:

```python
import sys

# Ask the user to input the payload length
payload_length = int(input("Enter the payload length: "))

# Open the file for writing
with open('payload', 'w') as f:
    # Write '0' characters
    for i in range(payload_length + 12):
        f.write(chr(48))  # ASCII value for '0'

        # Write null character
        if i == payload_length + 11:
            f.write(chr(0))  # Null character

print("Payload file 'payload' generated successfully.")
```

This script solicits user input to determine the desired payload length, a crucial parameter influencing the subsequent file generation. Once the user provides the payload length, the script proceeds to create a file named ”payload” in write mode. Within a loop iterating over the specified payload length plus an additional 12 characters, the script systematically writes ASCII-encoded '0' characters to the file. These characters are instrumental in manipulating the program's memory layout to exploit a buffer overflow vulnerability. Upon reaching the iteration corresponding to the payload length plus 11, the script injects a null character into the file. This null character serves a pivotal role, overwriting the return address of the vulnerable `strcpy` function on the stack. By strategically altering this address, the script can redirect the program's execution flow to a designated target function, such as the 'foo' function in the xv6 system, which outputs a predetermined message. Finally, upon completing the payload generation process, the script notifies the user of its successful execution, ensuring the creation of the payload file.

2 ASLR (Address Space Layout Randomization)

This part of the code reads the ASLR flag from a file named ”aslr_flag” and stores its value in the variable `c`. The `aslr_flag_checker` function is then called to determine the ASLR status based on the value of `c`. The result is stored in the `aslr_flag` variable. Subsequently, the `random_number_generator` function is invoked with `aslr_flag` as an argument to generate a random number.

2.1 ASLR Flag Checker Function

```c
int aslr_flag_checker(char c){
  int aslr_flag;
  if(c=='1'){
    aslr_flag=1;
  }
  else{
    aslr_flag=0;
  }
  return aslr_flag;
}
```

This function takes the value read from the file (`c`) as input and determines the ASLR flag status. If `c` is '1', it sets `aslr_flag` to 1 (indicating ASLR is enabled). Otherwise, it sets `aslr_flag` to 0 (indicating ASLR is disabled).

2.2 Random Number Generator Function

```c
int random_number_generator(int aslr_flag){
  int random_number;
  if(aslr_flag==1 && myproc()->pid>2){
    random_number = (ticks%5)+1;
  }
  else{
    random_number=0;
  }
  return random_number;
}
```

This function generates a random number based on the ASLR flag status (`aslr_flag`) and the process ID. If ASLR is enabled (`aslr_flag == 1`) and the process ID (`pid`) is greater than 2, it calculates a random number based on the system's tick count (`ticks`). Otherwise, it sets the random number to 0.

3 Memory Allocation in xv6

The following code snippet is part of the memory allocation process in the xv6 operating system. It is responsible for allocating memory for program segments during the execution of an executable file.

```c
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
```

```
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
        goto bad;
    if(ph.vaddr % PGSIZE != 0)
        goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz+(rand
       *4096))) == 0)
        goto bad;
    if(loaduvm(pgdir, (char*)(ph.vaddr+(rand*4096)), ip, ph.off
       , ph.filesz) < 0)
        goto bad;
}
```

This code snippet iterates through the program headers (‘elf.phnum‘) of an executable file and performs the following steps for each program header:

- Reads the program header from the file and validates its type.
- Checks if the memory size (‘ph.memsz‘) is sufficient to hold the contents of the segment.
- Validates the virtual address range to prevent invalid memory mappings.
- Ensures that the virtual address is page-aligned.
- Allocates memory for the program segment using the ‘allocuvm‘ function, incorporating a random offset generated by ASLR.
- Loads the program segment into memory using the ‘loaduvm‘ function, again incorporating the random offset.

If any of the checks fail, the code jumps to the ‘bad‘ label, indicating an error condition.

4 Program Segment Loading

The following code snippet demonstrates the loading of a program segment into memory in the xv6 operating system:

```
if(loaduvm(pgdir, (char*)(ph.vaddr+(rand*4096)), ip, ph.off, ph
   .filesz) < 0)
    goto bad;
```

This code snippet calls the `loaduvm` function to load a program segment into memory, incorporating a random offset introduced by ASLR. If the loading process encounters an error, it jumps to the `bad` label for error handling.

5 Stack Pointer Adjustment

The following code snippet adjusts the stack pointer (`esp`) in the trap frame (`tf`) of the current process in the xv6 operating system:

```
curproc->tf->esp = sp - (rand * 4096);
```

This line of code subtracts a random offset (generated by ASLR and multiplied by 4096) from the current stack pointer (`sp`), and assigns the result to the stack pointer field (`esp`) of the trap frame (`tf`) associated with the current process (`curproc`).

This adjustment ensures that the stack memory is allocated at a location that incorporates the random offset introduced by ASLR. This randomization enhances system security by making it more difficult for attackers to predict memory addresses and exploit vulnerabilities.

6 ASLR Implementation

The following code snippet demonstrates the implementation of Address Space Layout Randomization (ASLR) in the xv6 operating system:

```
if(rand > 0){
    pushcli();
    mycpu()->gdt[SEG_UCODE] = SEG(STA_X|STA_R, (rand*4096), 0
       xffffffff, DPL_USER);
    mycpu()->gdt[SEG_UDATA] = SEG(STA_W, (rand*4096), 0
       xffffffff, DPL_USER);
    popcli();
}
```

This code snippet checks if ASLR is enabled by verifying if the random offset (`rand`) is greater than 0. If ASLR is enabled, it adjusts the base addresses of the code and data segments in the Global Descriptor Table (GDT) to incorporate the random offset, ensuring that the addresses are offset by the random value. This randomization enhances system security by making it more difficult for attackers to predict the locations of code and data segments in memory.

7 Segment Register Restoration

In the `exit` function of `proc.c` in the xv6 operating system, the following lines of code are added to ensure that segment registers are restored to their original base address of 0 when a process exits:

```
pushcli();
mycpu()->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff,
   DPL_USER);
mycpu()->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
```

```
popcli();
```

These lines of code disable interrupts, restore the base addresses of the user code and data segment descriptors in the Global Descriptor Table (GDT) to 0, and then re-enable interrupts. This ensures that segment registers are reset to their original values after a process exits, preventing unintended changes in the address space that could affect other processes such as `init` and `sh`.

8 Challenges Faced and Resolutions

8.1 Trap 14 Error

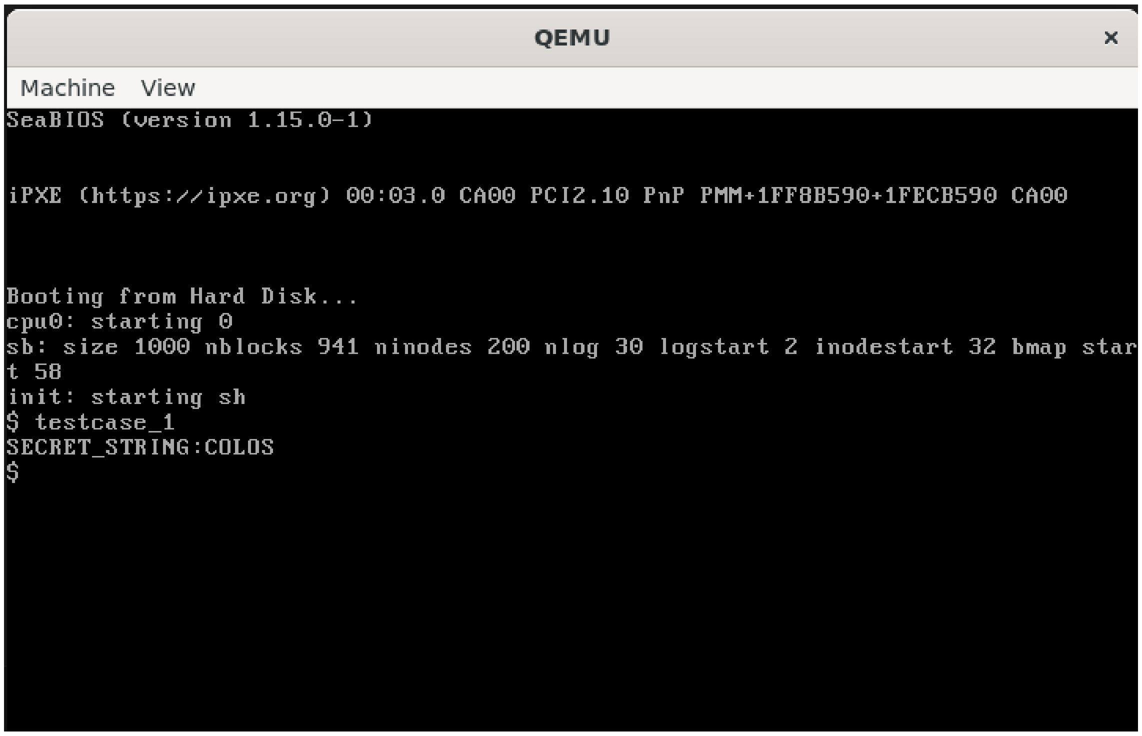
One of the challenges we faced initially was encountering a trap 14 error when segment registers were not changed. This error occurred because the stack and data addresses were incorrect due to the randomized address space. To resolve this issue, we changed the base addresses of the segments to provide the correct offset, ensuring that the program could run correctly.

8.2 Process Exit Issue

Another challenge we encountered was the unexpected termination of the init and sh processes. Process 3 was getting killed due to a trap 14 error, causing sh, which was waiting on process 3, to also receive a trap 14 error and get killed. Subsequently, init exited as well. This issue occurred because the randomization of the address space of process 3 did not restore the segments before exiting, resulting in sh using incorrect memory addresses and triggering a page fault. To resolve this issue, we ensured that the segment registers were restored to their original values before the processes exited, thus preventing incorrect memory addresses and subsequent termination of init and sh.

9 Result

9.1 With ASLR OFF



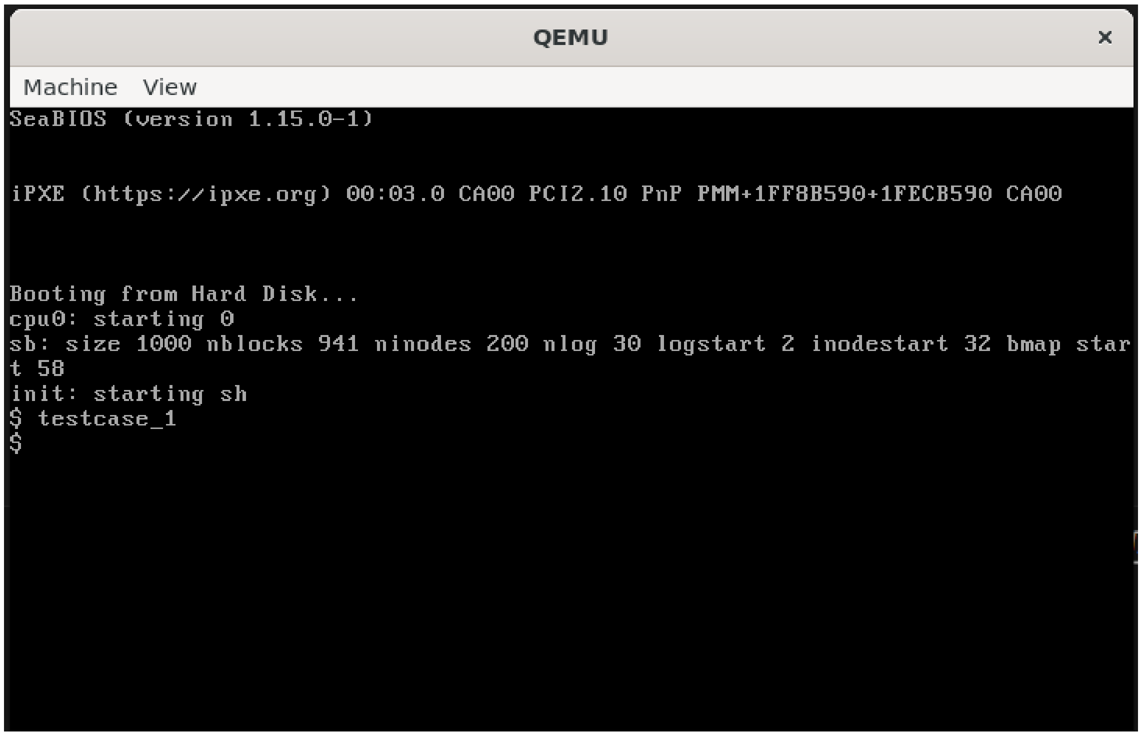
```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ testcase_1
SECRET_STRING:COLOS
$
```

9.2 With ASLR ON



```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ testcase_1
$
```