COL331 A2

Roshan Prashant Bara (2020CS10377)
Kushagra Rode (2020CS10354)

April 2023

Flags which have been added or removed in the MakeFile

1. -fno-pic flag is removed
2. -O2 flag is removed and replaced by -O0 flag
3. -pie -fPIE flags have been added instead of -fno-pie -nopie so as to generate Position Independent Executable

Buffer Overflow Attack in XV6

Description

In this, we first analyzed the user stacks to get an idea of what all values get pushed and their positions on the stack. We tried to retrieve the values at which the buffer is allocated on the stack and its position with respect to the return address of the vulnerable function saved on the stack. To do this, we first used Inline Assembly instructions in C and then GDB. We observed that the stack positions of the buffer and return address on the stack relative to the buffer changed when we declared new variables. So we used GDB to analyze the relative offsets. Finally, we found that in the given test code, the return address of the `vulnerable` function is stored at `12+buffer_size` i.e. `ebp + 4`. So we write the payload file such that the address of foo is written after `12+buffer_size` bytes so that the return address saved on stack gets

overwritten by the address of foo `<0x0000>`. So, when the function returns, the `eip` register gets the address of `foo` instead of the original return address resulting in `foo` getting executed even when the program doesn't call the function. This is how we performed the buffer overflow attack. Our file `gen_exploit.py`, writes random values on first `12+buffer_size` bytes and appends to it the foo address `<0x0000>`.

Address Space Layout Randomization

Creating the Pseudo Random Number Generator

For generating the random number, we have made use of the Linear congruential generator algorithm which yields a sequence of pseudo random numbers based on the seed value given to it. We have defined this generator in **'random.c'** file. This file has two functions `srand()` and `rand()`. The `srand()` takes an integer as input, this input is stored as a seed for random number generation. The `rand()` function then uses the LCG parameters of the glibc standard, i.e. $a = 1103515245$ and $c = 12345$. We then generate a random number by applying the formula,

$$(a * seed + c)\, mod\ \ RAND\_MAX$$

. This generates a random number between 0 and RAND_MAX - 1 (both inclusive). In our code, RAND_MAX is set to 1000.

Implementing Address Space Layout Randomization(ASLR)

We tried many different things and faced lot of challenges for implementing ASLR. We tried randomizing the location of the user stack and heap but it didn't work out. We got to know about the ELF executable's program headers and tried randomizing their offsets. We then tried changing the sz parameter but that caused page alignment errors in the `loaduvm` function. Hence, we got to know that something needs to be modified in the `loaduvm` function, so we took the help of the resource given by sir on Piazza. We took ideas from the riscv implementation of ASLR and implemented it for our x86 version of xv6.
Now I'll describe the changes we had to make in the loaduvm function and how ASLR is implemented.

Firstly, the aslr_flag file's ELF file is read through the readi function, if it contains a 1 then the aslr flag is turned on otherwise it is turned off. The next step is to generate a random number for randomizing the loading of the program instructions into our memory. For the seed, we have used the `date.h` library and `cmostime` function which is present in the `lapic.c` file. This