COL331 A3

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

April 2023

Flags which have been added or removed in the Make-File

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` where ebp is extended base pointer register of x86. 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 `0x00000000`. 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 the first `12+buffer_size` bytes and appends to it the foo address `0x00000000`.

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,

$$((\text{a*seed + c}) \&\& \text{0x7fffffff}) \text{ mod RAND\_MAX}$$

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 a 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 function gives the current date and time from the system's real-time clock (RTC) and store it in a struct rtcdate. We then generate the random number using this seed.
This random number is set as the value of sz. We then allocate sz amount of space using allocuvm, this space is essentially the space that results in a change of the base address of the foo function. This sz is then used as an offset with `ph.vaddr` to load the different sections of the program (i.e text, data, bss etc.) into the memory.

Now the major function comes which we have modfied for randomization, this is the loaduvm function. The parameters to this function are same as the original loaduvm. We will now discuss what changes we have done for aligning the pages in the loaduvm function. Suppose, we chose any random value for sz. The loaduvm firstly uses the `PGROUNDDOWN` macro to get the virtual address which is just lower than sz and is page aligned, we next calculate the offset between sz and this virtual address obtained. We get the physical address corresponding to the starting virtual address. This is obtained through the virt_to_phy function which essentially calls walkpgdir to get the page table entry corresponding to our virtual address. We then fill the page starting from this physical address with zeros. This is basically the extra thing that is needed to align pages.

We then fill up the remainder of the page starting from the offset found earlier with our data. For subsequent page allocations we simply have a for loop which is similar to the one present in the original loaduvm function.

We further randomize the starting location of the stack. Here we generate another random number(this time having a maximum value of 63) and then allocate those many pages to the stack and then discard the first page which acts as a guard page.
Finally, in order for our program to start correctly we need to set the instruction pointer (eip register) to the `elf.entry` + offset, this offset is one which we generated randomly earlier.