COL331 : Operating Systems

Assignment 3 – Easy

Ishita Chaudhary
2019CS10360
April 30, 2023

Changes in Makefile

- The code is compiled without any optimization flags, i.e., -O2 flag is replaced by -O0 flag
- The executable is compiled as PIE (Position Independent Executable), for this, -fno-pie -no-pie flags are replaced by -fPIE -pie flags
- The new files created (like payload and aslr_flag) are added in fs.img rule of Makefile
- Since buffer_overflow.c is a user test, it is added to UPROGS and EXTRA rules

1. Buffer Overflow Attack in XV6

In xv6, the operating system uses a stack frame to allocate local variables and store the pointer to the parent stack frame. The ebp register holds this pointer, and the esp register holds the address of the top of the stack. Buffer overflow attacks exploit unsafe functions like *gets()* and *strcpy()* that don't check input bounds. The attacker inputs more data than the buffer can hold, overflowing into adjacent memory regions and overwriting saved registers and the return address. When the function returns, the modified return address can jump to a location controlled by the attacker, giving them control over the program's execution.

We write a python script *gen_exploit.py* that writes to a file (*payload*) an exploit code which, when passed to the buffer overflow binary (*buffer overflow.c*), executes the *foo()* function that prints a secret string on the console.

The struct.pack() function returns bytes object of the input given according to the format given, which in our case is <I which denotes unsigned integer.

```
# fill the buffer until the return_address
padding = b"0" * (buffer_size+12)

# The address of the foo function in memory deduced using printf(%p, &foo) in buffer_overflow.c
foo_addr = 0X0
return_address = struct.pack("<I", foo_addr)

# payload
to_write = padding + return_address
```

The padding has been decided such that when the *strcpy()* function tries to copy the *payload* into the buffer, the buffer overflows on the stack and so the vulnerable function returns into the function *foo()* which was never called and ends up printing the secret string. We know that when a function returns, the stored return address is popped into the instruction pointer (eip), and the control jumps to that address. But due to buffer

overflow, *strcpy()* ends up overwriting the stored return address (located at *buffer_size+12*) with that of function *foo()*. This offset is due to the presence of three registers namely *eax*, *ebp* and *ebx* along with the allocated buffer.

```
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
SECRET_STRING
```

The above snippet shows a successful buffer overflow attack as we can see the secret string being printed without actually calling the function *foo().*

2. Address Space Layout Randomization

The aslr_flag file contains a single bit which denotes the status of ASLR used or not (1 represents ASLR being used, 0 otherwise). Any new function defined in any of the kernel files is added to defs.h along with the signature of the function.
Random number generator in xv6 (exec.c), we made a function to set the seed and another one to obtain the random number based on the seed. We can generate random numbers using various seed values in the "srand" function. This random number generator provides a variable offset which is currently set to 0.

```
static uint rand_seed = 1;

void srand(uint seed) {
  rand_seed = seed;
}

uint rand() {
  const uint a = 1078;
  const uint c = 821;
  const uint m = 1 << 31;
  rand_seed = (a * rand_seed + c) % m;
  return rand_seed;
}
```

We implemented the ASLR functionality by introducing offsets in the functions responsible for memory mappings and page tables (in vm.c and exec.c). Their function signatures are changed accordingly in defs.h. This is done in order to update

the base addresses according to the random offsets. The targeted functions are loaduvm(), copyout(), exec(), allocuvm(), etc.

Challenges Faced and their resolutions

- A key challenge we faced was understanding how to use GDB to effectively debug xv6. This was resolved by seeking help from online resources and reviewing the xv6 documentation, which provided us with a better understanding of GDB's features and how to use them to isolate and fix bugs in the system.
- Another key challenge faced while implementing ASLR in xv6 was understanding the concept of position independent executable (PIE). PIE allows the program to be loaded at any address in memory, making it difficult to predict the address of the program's functions and variables. This posed a challenge in implementing ASLR as we needed to randomise the address of the program's code and data segments. To resolve this, we had to modify the linker script to create a PIE binary and update the kernel to support randomization of the program's address space.
- Random number generator must use a random seed so that the random addresses are not deterministic in case the seed is known. Another issue with respect to this was that xv6 didn't allow variables to be determined at runtime which posed a challenge while implementing a random number generator with a random seed. We explored passing a random seed everytime the program is run.
- Ensuring proper handling for memory allocation and deallocation, allocuvm() and deallocuvm() are used to allocate and deallocate space in the process's virtual address space in xv6. By default they are used to operate on adding and removing space based on address spaces starting at 0. We have to incorporate offset changes to accurately allocate & deallocate space due to randomising effects.
- ELF relocations are required after randomising the base address of the code segment because we need to adjust the absolute addresses and references within the ELF file to match the new randomised memory layout.

References

[1] https://github.com/mit-pdos/xv6-public
[2] https://github.com/johnmwu/xv6-aslr
[2] https://github.com/TypingKoala/xv6-riscv-aslr
[4] https://iq.opengenus.org/fpic-in-gcc/
[5] https://docs.oracle.com/cd/E26505_01/html/E26506/glmqp.html
[6] https://pax.grsecurity.net/docs/aslr.txt