# ECE 568 – Computer Security

**The Edward S. Rogers Sr. Department of Electrical and Computer Engineering**

**Mid-term Examination, Part 1, October 2019**

| | |
|---|---|
| **Name** | **Solutions** |
| **Student #** | |

Answer all questions. Write your answers on the exam paper. Show your work.
Each question has a different assigned value, as indicated.

Permitted: one 8.5 x 11", two-sided page of notes.
No other printed or written material. No calculator.
NO PHOTOCOPIED MATERIAL
Total time: 50 minutes
Total marks available: 50 (roughly one mark per minute, with some extra time)
Verify that your exam has all the pages.
**Only exams written in ink will be eligible for re-marking**.

| 1 /25 | 2 /25 | Total |
|---|---|---|
| | | |

# Question 1: Buffer overflows [25 marks]

**Program:**

```
1:   int foo(char *arg)
2:   {
3:     char buf[16];
4:     int i, len;
5:
6:     len = strlen(arg);
7:
8:     if (len > 24)
9:       len = 24;
10:    for (i = 0; i <= len; i++)
11:      buf[i] = arg[i];
12:    return 0;
13:  }
14:
15:  int main(int argc, char *argv[])
16:  {
17:    char string[32];
18:
19:    strncpy(string, argv[1], 32);
20:    foo(argv[2]);
21:    return 0;
22:  }
```

**Registers:**

```
rbp            0x7fffffffe4a0   0x7fffffffe4a0
rsp            0x7fffffffe460   0x7fffffffe460
```

**Stack:**

```
0x7fffffffe460: 0x00000000      0x00000000      0xffffe83b      0x00007fff
0x7fffffffe470: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe480: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe490: 0x00000000      0x00000000      0x5327f500      0xae7022b9
0x7fffffffe4a0: 0xffffe4f0      0x00007fff      0x004006a6      0x00000000
0x7fffffffe4b0: 0xffffe5d8      0x00007fff      0x0040071d      0x00000003
0x7fffffffe4c0: 0x006f6f66      0x00000000      0x00000000      0x00000000
0x7fffffffe4d0: 0x00000000      0x00000000      0x00000000      0x00000000
```

**Other info:**

```
(gdb) p &buf
$1 = (char (*)[16]) 0x7fffffffe480
(gdb) p&i
$2 = (int *) 0x7fffffffe478
(gdb) p&len
$3 = (int *) 0x7fffffffe47c
(gdb) p &string
$4 = (char (*)[32]) 0x7fffffffe4c0
```

A program with a buffer overflow vulnerability is given above. The program is executed with an input passed in at the command line from the attacker. The state of the registers and stack when the program reaches line 6 is given. Answer the following questions (next page):

a) Are either the buffers `buf` or `string` vulnerable to a memory corruption attack? Please state your assumptions. [6 marks]

**buf- Vulnerable:** Because buf is 16 bytes but the loop allows writing 25 bytes, an attacker can corrupt data after the buffer

**string- Not vulnerable:** strncpy will only copy the first 32 bytes so while string might not be null terminated, strncopy will not write beyond the end of the buffer

b) At what addresses on the stack are the return address of `main` and `foo` located? Explain your answer [10 marks]

Foo's return address is at `0x7fffffffe4a8`
Main's return address is at `0x7fffffffe4f8`
The return address is always at the function's framepointer +8 (64 bits)

c) Can an attacker exploit any vulnerability in this program to execute arbitrary code of the attacker's choice? Explain your answer [5 marks]

No, while buf is vulnerable to memory corruption i and len are above the buffer so they can't be corrupted and the return address is more than 24 bytes from the start of the buffer

d) The attacker wants to get shellcode into the program but the attacker's shellcode is exactly 42 bytes long and can't fit into either buffer. Describe how the attacker can modify their shellcode to successfully solve this limitation. Use array notation to describe chunks of existing shellcode (i.e. shellcode[0-9] is the first 10 bytes of the old shellcode), and use pseudo-assembler to describe any new instruction you would insert into the shellcode [4 marks]:

The attacker can split the shellcode between buf and string. For example, she can put part of the shellcode in buf and then jump to the rest of the shellcode in string.

## Question 2: Fixing vulnerabilities [25 marks]

In this question you will be referring to the program in Question 1. For each proposed code change below, indicate with **Yes** or **No** whether the change fixes a vulnerability and/or introduces (i.e. adds) a new vulnerability. For clarity, the code changes have been bolded. For each answer, include a brief explanation [4 marks each]

i. Change line 19 to `strncpy(string, argv[1], 31);`

| Fixes? | Adds? |
|---|---|
| No | No |

Copies one less than the size of the buffer, but was not originally vulnerable so No to both

ii. Change line 8 to `if (len >= 24)`

| Fixes? | Adds? |
|---|---|
| No | No |

No difference since if Len is 24, it still gets set to 24, so this is identical to the original program

iii. Change line 19 to `snprintf(string, 32, argv[1],);`

| Fixes? | Adds? |
|---|---|
| No | Yes |

Functionally equivalent to original except now it is guaranteed to null terminate. However, introduces a format string vulnerability.

iv. Change line 3 to char buf[**24**];

| Fixes? | Adds? |
|---|---|
| No | No |

This doesn't help because the loop can copy 25 bytes so attacker can still overflow the buffer, just by less bytes. The attacker can still corrupt memory (note that the vulnerability cannot be exploited to corrupt a return address, but can potentially corrupt memory)

e) Please explain if and how the following counter measures address the types of vulnerabilities in the program on page 2. For each vulnerability, please state if it **Completely** prevents the vulnerabilities from being exploited, **Mitigates** the vulnerability by making it harder to exploit or does **Nothing** to the vulnerability [3 marks each]:

i. Stackguard/Stack Canaries:

The question asks about the type of vulnerability, which are stack-based overflows (buffers are all local variables). Thus we answer with respect to stack-based overflows.

**Completely:** it protects return addresses from being corrupted. Stack-based overflows cannot go around canary.

ii. Non-executable pages/DEP:

**Mitigates:** Does not prevent ROP attacks

iii. Control-flow integrity (CFI):

**Completely:** prevents the attacker from redirecting execution after corrupting return addres