Memory Safety

Question 1 ***Hacked EvanBot*** **(16 min)**

Hacked EvanBot is running code to violate students' privacy, and it's up to you to disable it before it's too late!

```
1  #include <stdio.h>
2
3  void spy_on_students(void) {
4      char buffer[16];
5      fread(buffer, 1, 24, stdin);
6  }
7
8  int main() {
9      spy_on_students();
10     return 0;
11 }
```

The shutdown code for Hacked EvanBot is located at address `0xdeadbeef`, but there's just one problem—Bot has learned a new memory safety defense. Before returning from a function, it will check that its saved return address (rip) is not `0xdeadbeef`, and throw an error if the rip is `0xdeadbeef`.

Clarification during exam: Assume little-endian x86 for all questions.

Assume all x86 instructions are 8 bytes long. Assume all compiler optimizations and buffer overflow defenses are disabled.

The address of `buffer` is 0xbffff110.

Q1.1 (3 points) In the next 3 subparts, you'll supply a malicious input to the `fread` call at line 5 that causes the program to execute instructions at `0xdeadbeef`, *without* overwriting the rip with the value `0xdeadbeef`.

The first part of your input should be a single assembly instruction. What is the instruction? x86 pseudocode or a brief description of what the instruction should do (5 words max) is fine.

Q1.2 (3 points) The second part of your input should be some garbage bytes. How many garbage bytes do you need to write?

◯ (G) 0 ◯ (H) 4 ◯ (I) 8 ◯ (J) 12 ◯ (K) 16 ◯ (L) —

Q1.3 (3 points) What are the last 4 bytes of your input? Write your answer in Project 1 Python syntax, e.g. `\x12\x34\x56\x78`.

Q1.4 (3 points) When does your exploit start executing instructions at `0xdeadbeef`?

◯ (G) Immediately when the program starts

◯ (H) When the `main` function returns

◯ (I) When the `spy_on_students` function returns

◯ (J) When the `fread` function returns

◯ (K) —

◯ (L) —

Q1.5 (4 points) Which of the following defenses would stop your exploit from the previous parts?

☐ (A) Non-executable pages (also called DEP, W^X, and the NX bit)

☐ (B) Stack canaries

☐ (C) ASLR

☐ (D) Rewrite the code in a memory-safe language

☐ (E) None of the above

☐ (F) —

Question 2 ***Stack Exchange*** **(19 min)**

Consider the following vulnerable C code:

```
1  #include <byteswap.h>
2  #include <inttypes.h>
3  #include <stdio.h>
4
5  void prepare_input(void) {
6      char buffer[64];
7      int64_t *ptr;
8
9      printf("What is the buffer?\n");
10     fread(buffer, 1, 68, stdin);
11
12     printf("What is the pointer?\n");
13     fread(&ptr, 1, sizeof(uint64_t *), stdin);
14
15     if (ptr < buffer || ptr >= buffer + 68) {
16         printf("Pointer is outside buffer!");
17         return;
18     }
19
20     /* Reverse 8 bytes of memory at the address ptr */
21     *ptr = bswap_64(*ptr);
22 }
23
24 int main(void) {
25     prepare_input();
26     return 0;
27 }
```

The `bswap_64` function takes in 8 bytes and returns the 8 bytes in reverse order.

Assume that the code is run on a 32-bit system, no memory safety defenses are enabled, and there are no exception handlers, saved registers, or compiler padding.

Q2.1 (3 points) Fill in the numbered blanks on the following stack diagram for `prepare_input`.

1	(0xbffff494)
2	(0xbffff490)
3	(0xbffff450)
4	(0xbffff44c)

◯ (A) 1 = `sfp`, 2 = `rip`, 3 = `buffer`, 4 = `ptr`

◯ (B) 1 = `sfp`, 2 = `rip`, 3 = `ptr`, 4 = `buffer`

◯ (C) 1 = `rip`, 2 = `sfp`, 3 = `buffer`, 4 = `ptr`

◯ (D) 1 = `rip`, 2 = `sfp`, 3 = `ptr`, 4 = `buffer`

◯ (E) —

◯ (F) —

Q2.2 (4 points) Which of these values on the stack can the attacker write to at lines 10 and 13? Select all that apply.

☐ (G) `buffer`
☐ (H) `ptr`
☐ (I) `sfp`
☐ (J) `rip`
☐ (K) None of the above
☐ (L) —

Q2.3 (3 points) Give an input that would cause this program to execute shellcode. At line 10, first input these bytes:

◯ (A) 64-byte shellcode
◯ (B) `\xbf\xff\xf4\x4c`
◯ (C) `\x4c\xf4\xff\xbf`
◯ (D) `\xbf\xff\xf4\x50`
◯ (E) `\x50\xf4\xff\xbf`
◯ (F) —

Q2.4 (3 points) Then input these bytes:

◯ (G) 64-byte shellcode
◯ (H) `\xbf\xff\xf4\x4c`
◯ (I) `\x4c\xf4\xff\xbf`
◯ (J) `\xbf\xff\xf4\x50`
◯ (K) `\x50\xf4\xff\xbf`
◯ (L) —

Q2.5 (3 points) At line 13, input these bytes:

◯ (A) `\xbf\xff\xf4\x50`
◯ (B) `\x50\xf4\xff\xbf`
◯ (C) `\xbf\xff\xf4\x90`
◯ (D) `\x90\xf4\xff\xbf`
◯ (E) `\xbf\xff\xf4\x94`
◯ (F) `\x94\xf4\xff\xbf`

Q2.6 (3 points) Suppose you replace `68` with `64` at line 10 and line 15. Is this modified code memory-safe?

◯ (G) Yes ◯ (H) No ◯ (I) — ◯ (J) — ◯ (K) — ◯ (L) —