Nicholas & Peyrin
Summer 2021

CS 161
Computer Security

Final Review

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. [1]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.

Solution: `jmp *0xdeadbeef`

[1]In practice, x86 instructions are variable-length.

You can't overwrite the rip with `0xdeadbeef`, but you can still overwrite the rip to point at arbitrary instructions located somewhere else. The idea here is to overwrite the rip to execute instructions in the buffer, and write a single jump instruction that starts executing code at `0xdeadbeef`.

Grading: most likely all or nothing, with some leniency as long as you mention something about jumping to address `0xdeadbeef`. We will consider alternate solutions, though.

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) ——

Solution: After the 8-byte instruction from the previous part, we need another 8 bytes to fill buffer, and then another 4 bytes to overwrite the sfp, for a total of 12 garbage bytes.

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`.

Solution: `\x10\xf1\xff\xbf`

This is the address of the jump instruction at the beginning of `buffer`. (The address may be slightly different on randomized versions of this exam.)

Partial credit for writing the address backwards.

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) ——

Solution: The exploit overwrites the rip of `spy_on_students`, so when the `spy_on_students` function returns, the program will jump to the overwritten rip and start executing arbitrary instructions.

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) ——

Solution: Non-executable pages prevents the exploit because the exploit requires executing the `jmp` instruction that was written on the stack.

Stack canaries prevent the exploit because the exploit will overwrite the canary between `buffer` and the rip.

ASLR prevents the exploit because the exploit requires overwriting the rip with a known address on the stack.

Many people asked in clarifications if ASLR would change the address of the shutdown code at `0xdeadbeef`. We didn't answer this clarification because it doesn't affect the correct answer choice here: even if you knew the absolute address of the shutdown code, you couldn't overwrite the rip with the address of the buffer on the stack, because ASLR would randomize addresses on the stack.

Using a memory-safe language always prevents buffer overflow attacks.

Question 2 *Stack Exchange* (19 min)

Consider the following vulnerable C code:

```c
#include <byteswap.h>
#include <inttypes.h>
#include <stdio.h>

void prepare_input(void) {
    char buffer[64];
    int64_t *ptr;

    printf("What is the buffer?\n");
    fread(buffer, 1, 68, stdin);

    printf("What is the pointer?\n");
    fread(&ptr, 1, sizeof(uint64_t *), stdin);

    if (ptr < buffer || ptr >= buffer + 68) {
        printf("Pointer is outside buffer!");
        return;
    }

    /* Reverse 8 bytes of memory at the address ptr */
    *ptr = bswap_64(*ptr);
}

int main(void) {
    prepare_input();
    return 0;
}
```

The `bswap_64` function [2]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`)

[2]Technically, this is a macro, not a function.

○ (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) —

Solution: The rip is pushed onto the stack first, followed by the sfp, followed by the first local variable buffer, followed by the second local variable ptr. (Remember that local variables are placed on the stack, highest-to-lowest address, in the order they are defined in the code.)

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) —

Solution: At line 10, the attacker can write 68 bytes starting at `buffer`. This overwrites all 64 bytes `buffer` and the 4 bytes directly above it, which is the `sfp`.

At line 13, the attacker can write exactly 1 `uint64_t *` into `ptr`. This overwrites `ptr`, and nothing else.

Notice that the rip cannot be directly overwritten.

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` [✓ (D) `\x90\xf4\xff\xbf`]

(B) `\x50\xf4\xff\xbf` (E) `\xbf\xff\xf4\x94`

(C) `\xbf\xff\xf4\x90` (F) `\x94\xf4\xff\xbf`

Solution: Line 10 writes 68 bytes into the 64-byte buffer, which lets us overwrite the sfp, but not the rip.

Line 13 lets us write a value into `ptr`, which is then dereferenced in a call to `bswap_64`. This lets us reverse any 8 bytes in memory we want, as long as they are between `buffer` and `buffer + 68`, i.e. in the buffer or sfp.

The overarching idea here is to write the address of shellcode in the sfp, and then use the call to `bswap_64` to swap the sfp and the rip.

First, we write the 64 bytes of shellcode into the buffer. Then, we overwrite the sfp with `\xbf\xff\xf4\x50`. These bytes are written backwards because `bswap_64` will reverse all 8 bytes of the sfp and the rip. Finally, we write the address of the sfp, `\x90\xf4\xff\xbf`, into ptr. These bytes are written normally because `bswap_64` never affects ptr.

Suppose the current rip is `0xdeadbeef`. Our input causes the 8 bytes starting at the sfp to be `\xbf\xff\xf4\x50\xef\xbe\xad\xde`. When we call `bswap_64` at the location of sfp, the 8 bytes starting at sfp are reversed, so they are now `\xde\xad\xbe\xef\x50\xf4\xff\`
Notice that the rip is now pointing to the address of shellcode in the correct little-endian order. Also note that the original rip has been swapped into the sfp and is now backwards, although we don't care because the rip has already been overwritten.

Note: Because you can overwrite the sfp, you might be tempted to use the off-by-one exploit from Q4 of Project 1. However, this does not work here because you need enough space to write the shellcode and the address of shellcode in the buffer, but the buffer only has space for the shellcode.

Partial credit for Q7.4 option (K) and Q7.5 option (C) (correct address, but backwards).

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) ——

Solution: This is still not memory-safe. If you make `ptr` point at one of the last 4 bytes of `buffer`, the check at line 15 will pass, but it will cause part of the sfp to be overwritten. For example, if `ptr` is located 4 bytes before the end of `buffer`, the last 4 bytes of `buffer` will be swapped into the sfp.

Because you can overwrite the sfp, you could still exploit this modified code using the technique from Project 1, Question 4 (although as mentioned above, you would need shorter shellcode). Regardless of what shellcode you use, since this code lets you write to the sfp (outside the bounds of `buffer`), it is not memory-safe.