Popa & Kao
Spring 2023

CS 161 Computer Security

Exam Prep 2

Q1 *Indirection* (0 points)

Consider the following vulnerable C code:

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  struct log_entry {
5      char title[8];
6      char *msg;
7  };
8
9  void log_event(char *title, char *msg) {
10     size_t len = strnlen(msg, 256);
11     if (len == 256) return; /* Message too long. */
12     struct log_entry *entry = malloc(sizeof(struct log_entry));
13     entry->msg = malloc(256);
14     strcpy(entry->title, title);
15     strncpy(entry->msg, msg, len + 1);
16     add_to_log(entry); /* Implementation not shown. */
17 }
```


Assume you are on a little-endian 32-bit x86 system and no memory safety defenses are enabled.

Q1.1 (3 points) Which of the following lines contains a memory safety vulnerability?

○ (A) Line 10

○ (B) Line 13

● (C) Line 14

○ (D) Line 15

○ (E) ——

○ (F) ——

Solution: Line 14 uses a `strcpy`, which is not a memory-safe function because it terminates only when it sees a NULL byte, which is under the control of the attacker. Note that line 15 uses a `strncpy` whose length parameter comes from `strnlen`, so it is safe.

Q1.2 (3 points) Fill in the numbered blanks on the following stack and heap diagram for `log_event`. Assume that lower-numbered addresses start at the bottom of both diagrams.

Stack
`msg`
1
rip
sfp
`len`
`entry`

Heap
3
2

○ (G) 1 = `entry->title` 2 = `entry->title` 3 = `msg`

○ (H) 1 = `entry->title` 2 = `msg` 3 = `entry->title`

● (I) 1 = `title` 2 = `entry->title` 3 = `entry->msg`

○ (J) 1 = `title` 2 = `entry->msg` 3 = `entry->title`

○ (K) ——

○ (L) ——

Solution: The two arguments, `title` and `msg`, must be on the stack, so 1 = `msg`.

Structs are filled from lower addresses to higher addresses, so 2 = `entry->title` and 3 = `entry->msg`.

Using GDB, you find that the address of the rip of `log_event` is `0xbfffe0f0`.

Let `SHELLCODE` be a 40-byte shellcode. Construct an input that would cause this program to execute shellcode. Write all your answers in Python 2 syntax (just like Project 1).

Q1.4 (6 points) Give the input for the `title` argument.

Solution: The `title` will be used to overflow the `title` buffer in the struct to point the `msg` pointer to the RIP. The input should thus be

```
'A' * 8 + '\xf0\xe0\xff\xbf'
```

Q1.5 (6 points) Give the input for the `msg` argument.

Solution: The first 4 bytes will be written in the location of the RIP, which should point to the shellcode. Thus, our input should be

```
'\xf4\xe0\xff\xbf' + SHELLCODE
```

Q2 *Stack Exchange* (19 points)

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 [1] 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.

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

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) —

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.

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`
- ○ (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`

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 an arbitrary value into ptr, which is then dereferenced in a call to `bswap_64`. This lets us reverse any 8 bytes in memory that we want.

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\xbf`. Notice that the rip is now pointing to the address of shellcode in the correct little-endian order.

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.

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: No. If you make ptr point at one of the last 4 bytes of buffer (which passes the check at line 15), 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).

This is the end of Q2. Leave the remaining subparts of Q2 blank on Gradescope, if there are any. You have reached the end of the exam.