Fuzail and Peyrin
Summer 2022

CS 161
Computer Security

Discussion

Question 1 ***Hacked EvanBot*** **()**

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) ——

Question 2 *Indirection* ()

Consider the following vulnerable C code:

```c
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.

Q2.1 (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) ——

Q2.2 (3 points) Seeing an opportunity to exploit this program, you fire up GDB and step into the `log_event` function. Give a GDB command that will show you the address of the rip of the `log_event` function. (Abbreviations are fine.)

Q2.3 (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

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

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

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

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

○ (E) ——

○ (F) ——

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).

Q2.4 (6 points) Give the input for the `title` argument.

Q2.5 (6 points) Give the input for the `msg` argument.

○ (A) —— ○ (B) —— ○ (C) —— ○ (D) —— ○ (E) —— ○ (F) ——

Question 3 *161 Meets 61A* ()

Consider the following buggy C code:

```c
void add_letter(int i, char *buf) {
    char word[4];
    printf("Enter Word %d:\n", i);
    fgets(word, 4, stdin);
    buf[i] = word[0];
    if (i > 0) {
        add_letter(i - 1, buf);
    }
}

void make_acronym(void) {
    char result[4];
    add_letter(4, result);
    printf("%s\n", result);
}

void word_games(void) {
    make_acronym();
}

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

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or additional saved registers in all subparts. Assume all memory-safety defenses (ASLR, stack canaries, pointer authentication codes, and non-executable pages) are disabled, unless otherwise specified.

Q3.1 (3 min) How many times will the `add_letter` function be run each time the `make_acronym` function is called?

○ (A) 0 ○ (B) 1 ○ (C) 2 ○ (D) 3 ○ (E) 4 ○ (F) 5

Q3.2 (4 min) Which value(s) will be overwritten (partially or completely) when you provide an input for the prompt to “Enter Word 4:”? Select all that apply.

☐ (G) RIP of `word_games`

☐ (H) SFP of `word_games`

☐ (I) RIP of `make_acronym`

☐ (J) SFP of `make_acronym`

☐ (K) None of the above

☐ (L) ——

Assume that malicious shellcode is stored at `0x44332211` and the address of `result` is `0xAABBCCB8`. In the next five subparts, provide a series of inputs to `fgets` that would cause the program to execute shellcode.

Q3.3 (1 min) First input:

○ (A) `\xA9` ○ (B) `\xAC` ○ (C) `\xB0` ○ (D) `\xB4` ○ (E) `\xB8` ○ (F) `\xBC`

Q3.4 (1 min) Second input:

○ (G) `\x00` ○ (H) `\x11` ○ (I) `\x22` ○ (J) `\x33` ○ (K) `\x44` ○ (L) `\x48`

Q3.5 (1 min) Third input:

○ (A) `\x00` ○ (B) `\x11` ○ (C) `\x22` ○ (D) `\x33` ○ (E) `\x44` ○ (F) `\x48`

Q3.6 (1 min) Fourth input:

○ (G) `\x00` ○ (H) `\x11` ○ (I) `\x22` ○ (J) `\x33` ○ (K) `\x44` ○ (L) `\x48`

Q3.7 (1 min) Fifth input:

○ (A) `\x00` ○ (B) `\x11` ○ (C) `\x22` ○ (D) `\x33` ○ (E) `\x44` ○ (F) `\x48`

Q3.8 (3 min) Assume that you've successfully executed the exploit above. At what point will the function jump to your shellcode?

○ (G) When `main` returns

○ (H) When `word_games` returns

○ (I) When `make_acronym` returns

○ (J) When `add_letter` (called with `i == 4`) returns

○ (K) When `add_letter` (called with `i == 3`) returns

○ (L) None of the above