Madison & Ana
Summer 2023

CS 161 Computer Security

Discussion 3

Question 1 *C Memory Defenses*

Mark the following statements as True or False and justify your solution. Please feel free to discuss with students around you.

Q1.1 Stack canaries completely prevent a buffer overflow from overwriting the return instruction pointer.

Q1.2 A format-string vulnerability can allow an attacker to overwrite values below the stack pointer.

Q1.3 ASLR, stack canaries, and NX bits all combined are insufficient to prevent exploitation of all buffer overflow attacks.

Short answer!

Q1.4 What vulnerability would arise if the stack canary was between the return address and the saved frame pointer?

Q1.5 Assume ASLR is enabled. What vulnerability would arise if the instruction **jmp *esp** exists in memory?

Question 2 ***Robin***

Consider the following code snippet:

```
 1 void robin(void) {
 2     char buf[16];
 3     int i;
 4
 5     if (fread(&i, sizeof(int), 1, stdin) != 1)
 6         return;
 7
 8     if (fgets(buf, sizeof(buf), stdin) == NULL)
 9         return;
10
11     _________
12 }
```

Assume that:

- There is no compiler padding or additional saved registers.
- The provided line of code in each subpart compiles and runs.
- `buf` is located at memory address `0xffffd8d8`
- Stack canaries are enabled, and all other memory safety defenses are disabled.
- The stack canary is four completely random bytes (**no null byte**).

For each subpart, mark whether it is possible to leak the value of the stack canary. If you put possible, provide an input to Line 5 and an input to Line 8 that would leak the canary. If the line is not needed for the exploit, you must write "Not needed" in the box.

Write your answer in Python syntax.

Q2.1 (3 min) Line 11 contains `gets(buf);`.

○ Possible

○ Not possible

Line 5:

Line 8:

Q2.2 (5 min) **For this subpart only, enter an input that allows you to leak a single character from memory address `0xffffd8d7`. Mark “Not possible” if this is not possible.** Line 11 contains `printf("%c", buf[i]);`.

○ Possible

○ Not possible

Line 5:

Line 8:

Q2.3 (6 min) Line 11 contains `printf(buf);`.

○ Possible

○ Not possible

Line 5:

Line 8:

Q2.4 (6 min) Line 11 contains `printf(i);`.

○ Possible

○ Not possible

Line 5:

Line 8:

Question 3 ***Hulk Smash!***

Assume that:

- For your inputs, you may use SHELLCODE as a 16-byte shellcode.
- If needed, you may use standard output as OUTPUT, slicing it using Python syntax.
- All x86 instructions are **4 bytes** long.
- For each provided code snippet, you run GDB once, and discover that:
  - The address of the RIP of the `hulk` method is `0xffffcd84`.
  - The address of a `ret` instruction is `0x080722d8`.

Consider the following function:

```
int hulk(FILE *f, char *eyes) {
    void (* green_ptr)(void) = &green; //function pointer
    char buf[32];
    char str[28];
    fread(buf, 1, 32, f);
    printf("%s", buf);
    fread(buf, 4, 32, stdin);
    if (strlen(eyes) > 28) {
        return 0;
    }
    strncpy(str , eyes , sizeof(buf));
    return 1;
}
```

The following is the x86 code of `void green(void)`:

```
nop
nop
nop
ret
```

Assume that ASLR is enabled including the code section, but all other memory safety defenses are disabled.

Q3.1 (3 min) Fill in the following stack diagram, assuming that the program is paused after executing **Line 5**, including the arguments of `hulk` (the value in each row does not necessarily have to be four bytes long).

Stack

Q3.2 (10 min) Provide an input to each of the boxes below in order to execute `SHELLCODE`.

Provide a string value for `eyes` (argument to `hulk`):

Provide a string for the contents of the file that is passed in as the `f` argument of `hulk`:

Provide an input to the second `fread` in `hulk`: