Peyrin Kao
Fall 2022

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.

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

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

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

Short answer!

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

2. 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);`.

A. Possible

B. 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]);`.

A. Possible

B. Not possible

Line 5:

Line 8:

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

A. Possible

B. Not possible

Line 5:

Line 8:

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

A. Possible

B. Not possible

Line 5:

Line 8:

Question 3 ***Echo, Echo, Echo*** ()

Consider the following vulnerable C code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  char name[32];
5
6  void echo(void) {
7      char echo_str[16];
8      printf("What do you want me to echo back?\n");
9      gets(echo_str);
10     printf("%s\n", echo_str);
11 }
12
13 int main(void) {
14     printf("What's your name?\n");
15     fread(name, 1, 32, stdin);
16     printf("Hi %s\n", name);
17
18     while (1) {
19         echo();
20     }
21
22     return 0;
23 }
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or additional saved registers in all questions.

Q3.1 (5 min) Assume that non-executable pages are enabled so we cannot execute SHELLCODE on stack. We would like to exploit the `system(char *command)` function to start a shell. This function executes the string pointed to by `command` as a shell command. For example, `system("ls")` will list files in the current directory.

Construct an input to `gets` that would cause the program to execute the function call `system("sh")`. Assume that the address of `system` is `0xdeadbeef` and that the address of the RIP of `echo` is `0x9ff61fc4`. Write your answer in Python 2 syntax (like in Project 1).

Hint: Recall that a return-to-libc attack relies on setting up the stack so that, when the program pops off and jumps to the RIP, the stack is set up in a way that looks like the function was called with a particular argument.

Q3.2 (6 min) Assume that, in addition to non-executable pages, ASLR is also enabled. However, addresses of global variables are not randomized.

Is it still possible to exploit this program and execute malicious shellcode?

◯ (G) Yes, because you can find the address of both `name` and `system`

◯ (H) Yes, because ASLR preserves the relative ordering of items on the stack

◯ (I) No, because non-executable pages means that you can't start a shell

◯ (J) No, because ASLR will randomize the code section of memory

◯ (K) —

◯ (L) —