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.

   Solution:

   False, stack canaries can be defeated if they are revealed by information leakage, or if there is not sufficient entropy, in which case an attacker can guess the value. Also, format string vulnerabilities can simply skip past the canary.

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

   Solution:

   True, format string vulnerabilities can write to arbitrary addresses by using a '%n' together with a pointer.

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

   Solution:

   True, all of these protections can be overcome. The only way to prevent buffer overflow attacks is by using a memory-safe language.

Short answer!

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

   Solution:

   An attacker can overwrite the saved frame pointer so that the program uses the wrong address as the base pointer after it returns. This can be turned into an exploit, like an off-by-one attack that builds upon changing the LSB of SFP.

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

   Solution: An attacker can overwrite the RIP with the address of the **jmp *esp** instruction. An attacker could place the shellcode directly above the `RIP`. This will cause the function to execute the shellcode when it returns, since ESP will have just popped RIP off of the stack.

   There are a few more complications with this specific technique, "ret2esp", since the instruction **jmp *esp** is not usually part of a generated binary. You can find more details about it in section 8.3 of the "ASLR Smack & Laugh Reference" by Tilo Müller.

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:

Solution: N/A

Line 8:

Solution: N/A

Solution: There's not much we can do here as an attacker: there's no way to execute arbitrary shellcode to leak the canary, because we'd have to bypass the canary somehow; and there's no way of leaking the canary value directly as there are no read commands, only write commands.

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:

Solution: `'\xff\xff\xff\xff'`

Line 8:

Solution: Not needed

Solution: We can set `i` to -1 to read a value one byte below the buffer. We know that -1 is `0xffffffff` in two's complement, so we just enter that for the integer.

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

A. Possible

B. Not possible

Line 5:

Solution: Not needed

Line 8:

Solution: `'%c%c%c%c%c%x'`

Solution: This is just a simple format string attack: We just need to walk our way up the stack using `%c` specifiers until we reach `canary`, at which point we can dump the value of the canary using a `%x`.

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

A. Possible

B. Not possible

Line 5:

Solution: Approach 1: `'\xe8\xd8\xff\xff'`

Approach 2: `'\xd8\xd8\xff\xff'`

Line 8:

Solution: Approach 1: Not needed

Approach 2: `'%c%c%c%c%c%c%x'`

Solution: The first option is simple: Use the integer as a pointer directly to the stack canary, which causes it to be leaked since it's contents will be treated as the format string and directly printed out (since it's unlikely for it to contain a format specifier).

The second option is identical to the previous subpart, except for the fact that we're printing `i` instead of `buf` - as such, we need to set this up such that `i` is a pointer to the format string specifier, which resides at `buf`. We can do this by setting `i` to this address, so that when it's passed into `printf`, it's treated identically to passing in `buf` directly.

Question 3 *Echo, Echo, Echo* ()

Consider the following vulnerable C code:

```c
#include <stdio.h>
#include <stdlib.h>

char name[32];

void echo(void) {
    char echo_str[16];
    printf("What do you want me to echo back?\n");
    gets(echo_str);
    printf("%s\n", echo_str);
}

int main(void) {
    printf("What's your name?\n");
    fread(name, 1, 32, stdin);
    printf("Hi %s\n", name);

    while (1) {
        echo();
    }

    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 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.

Solution: When `echo` returns, the stack should look like below. We need to first garbage up to the rip (`'A' * 20`), replace the normal return address with the address of `system` (`'\xef\xbe\xad\xde'`), add four bytes of garbage where the `system` method expects the RIP of the caller to be, and place the address of `name` at the location where `system` expects an argument.

Stack
`command` (pointer to `"sh"`)
(Expected) RIP of `system`

As such, our exploit may look something like the following:

Approach 1: Place the "sh" above the RIP

```
'A' * 20 + '\xef\xbe\xad\xde' + 'B' * 4 + '\xd0\x1f\xf6\x9f'
+ 'sh' + '\x00'
```

Approach 2: Place the "sh" in the buffer

```
'sh' + '\x00' + 'A' * 17 + '\xef\xbe\xad\xde'
+ 'B' * 4 + '\xb0\x1f\xf6\x9f'
```

There may be a few other correct answers here (with the shellcode placed at slightly different offsets within the buffer or above the RIP), but these are the most common.

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) ——

Solution: If ASLR is enabled, the address of `system`, a line of code in the *code* section of memory, will be randomized each time the program is run. Because our exploit uses this address, ASLR will effectively prevent us from using our approach!