

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.

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.

Q1.2 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.

Q1.3 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!

Q1.4 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.

Q1.5 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);`.

- Possible
- 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]);`.

- Possible
- 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);`.

- Possible
- 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);`.

- Possible
- 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%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 its 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.