CS 161 Spring 2024 — Introduction to 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.

> **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' or '%hn' 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 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 ***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:

```
1  int hulk(FILE *f, char *eyes) {
2      void (* green_ptr)(void) = &green; //function pointer
3      char buf[32];
4      char str[28];
5      fread(buf, 1, 32, f);
6      printf("%s", buf);
7      fread(buf, 4, 32, stdin);
8      if (strlen(eyes) > 28) {
9          return 0;
10     }
11     strncpy(str, eyes, sizeof(buf));
12     return 1;
13 }
```

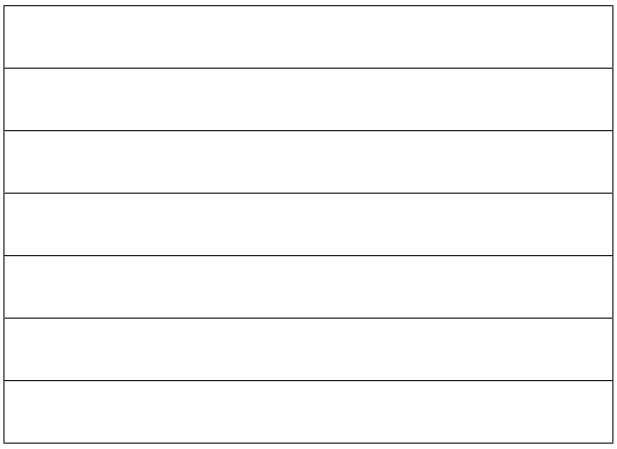
The following is the x86 code of `void green(void)`:

```
1 nop
2 nop
3 nop
4 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



Solution: Stack diagram:

```
[4] char *eyes
[4] File* f
[4] RIP hulk
[4] SFP hulk
[4] (void)* green_ptr
[32] char buf
[28] char str
```

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`):

Solution: SHELLCODE

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

Solution: 'A' * 32

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

Solution: 'A' * 40 + (OUTPUT[32:36] * 2)

Solution: Since ASLR is enabled, we need to leak an address on stack, and we notice that we have a function pointer with a ret instruction, so we can try to leak that address with the printf function. Recall that `printf` stops printing when it sees a null byte, so we have to get rid of all the null bytes between the beginning of the buffer and the function pointer. To slice the output, since we are printing from the start of `buf`, we can slice it from indices 32 to 36 as we notice that the green function is simply filled with NOPs and a ret instruction. In our second `fread`, we then start writing from the bottom of buf till we reach the RIP, and overwrite the RIP and (RIP+4) with ret instructions. We need two ret instructions here since we notice that we want some pointer that points to our SHELLCODE, which we placed in eyes.