CS161 Summer 2026 — Introduction to Computer Security — Discussion 3

Q1 *C Memory Defenses* **(5 points)**

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

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

○ True ○ False

Q1.2 (1 point) A format-string vulnerability can allow an attacker to overwrite values below the stack pointer.

○ True ○ False

Q1.3 (1 point) ASLR, stack canaries, and NX bits all combined are insufficient to prevent exploitation of all buffer overflow attacks.

○ True ○ False

Short answer!

Q1.4 (1 point) What vulnerability would arise if the stack canary was between the return address and the saved frame pointer?

Q1.5 (1 point) Assume ASLR is enabled. What vulnerability would arise if the instruction **jmp ESP** exists in memory?

Q2 *Second-order linear... err I mean SQL injection* **(3 points)**

Alice likes to use a startup, `NotAmazon`, to do her online shopping. Whenever she adds an item to her cart, a POST request containing the field `item` is made. On receiving such a request, `NotAmazon` executes the following statement:

```
cart_add := fmt.Sprintf("INSERT INTO cart (session, item) "
                                + "VALUES ('%s', '%s')", sessionToken, item)
db.Exec(cart_add)
```

Each item in the cart is stored as a separate row in the `cart` table.

Q2.1 (1 point) Alice is in desperate need of 2 pancake mixes, but the website only allows Alice to add 1 bag of pancake mix. Describe a POST request she can make to cause the `cart_add` statement to add 2 bags of pancake mix to her cart.

Q2.2 (1 point) What if Alice is in desperate need of 100 pancake mixes, how can you extrapolate the above input to add 100 pancake mixes to Alice's cart?

After part (a), Alice recognizes a great business opportunity and begins reselling all of NotAmazon's pancake mix at inflated prices. In a panic, NotAmazon fixes the vulnerability by parameterizing the `cart_add` statement.

```
cart_add := "INSERT INTO cart (session, item) VALUES (?, ?)"
db.Exec(cart_add, sessionToken, item)
```

But Alice realized that `cart_add` isn't the only place `item` is used. When a user visits their cart, NotAmazon populates the webpage with links to the items. If a user only has one item in their cart, NotAmazon optimizes the query (avoiding joins) by using the following:

```
cart_query := fmt.Sprintf("SELECT item FROM cart " +
                                "WHERE session='%s' LIMIT 1", sessionToken)
item := db.Query(cart_query)
link_query = fmt.Sprintf("SELECT link FROM items WHERE item='%s'", item)
db.Query(link_query)
```

Q2.3 (1 point) Alice claims that parameterizing the `cart_add` statement won't stop her pancake mix trafficking empire. Describe how she can still add 100 bags of pancake mix to her cart. Assume that `NotAmazon` checks that `sessionToken` is valid before executing any queries involving it.

Q3 *Cross-site not scripting* **(2 points)**

Consider a simple web messaging service. You receive messages from other users. The page shows all messages sent to you. Its HTML looks like this:

```
Mallory: Do you have time for a conference call?
Steam: Your account verification code is 86423
Mallory: Where are you? This is <b>important!!!</b>
Steam: Thank you for your purchase
       <img src="https://store.steampowered.com/assets/thankyou.png">
```

The user is off buying video games from Steam, while Mallory is trying to get ahold of them.

Users can include **arbitrary HTML code** messages and it will be concatenated into the page, **unsanitized**. Sounds crazy, doesn't it? However, they have a magical technique that prevents *any* JavaScript code from running. Period.

Q3.1 (1 point) Discuss what an attacker could do to snoop on another user's messages. What specially crafted messages could Mallory have sent to steal this user's account verification code?

Q3.2 (1 point) Keeping in mind the attack you constructed in the previous part, what is a defense that can prevent against it?

Q4 *Hulk Smash! (Extra Practice)* **(13 points)**

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.

Q4.1 (3 points) 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

Q4.2 (10 points) 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`:
