Web Security II

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

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 (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 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 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.
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  Cross-Site Request Forgery (CSRF)
In a CSRF attack, a malicious user is able to take action on behalf of the victim. Consider the following example. Mallory posts the following in a comment on a chat forum:

```html
<img src="http://patsy-bank.com/transfer?amt=1000&to=mallory"/>
```

Of course, Patsy-Bank won’t let just anyone request a transaction on behalf of any given account name. Users first need to authenticate with a password. However, once a user has authenticated, Patsy-Bank associates their session ID with an authenticated session state.

(a) Explain what could happen when Alice visits the chat forum and views Mallory’s comment.

**Solution:** The `img` tag embedded in the form causes the browser to make a request to `http://patsy-bank.com/transfer?amt=1000&to=mallory` with Patsy-Bank’s cookie. If Alice was previously logged in (and didn’t log out), Patsy-Bank might assume Alice is authorizing a transfer of 1000 USD to Mallory.

(b) Patsy-Bank decides to check that the `Referer` header contains patsy-bank.com. Will the attack above work? Why or why not?

**Solution:** In most cases, it will solve the problem since the `Referer` header will contain the blog’s URL instead of patsy-bank.com. However, not all browsers send the `Referer` header, and even when they do, not all requests include it.

(c) Describe one way Mallory can modify her attack to always get around this check.

**Solution:** She can have the link go to a URL under Mallory’s control which contains patsy-bank.com such as `patsy-bank.com.attacker.com` or `attacker.com/attack?dummy=patsy-bank.com`. Then this page can redirect to the original malicious link. Now the `Referer` header will pass the check.

Another solution, is if the Patsy-Bank has a so-called “open redirect” `http://patsy-bank.com/redirect?to=url`, the referrer for the redirected request will be `http://patsy-bank.com/redirect?to=...`. An attacker can abuse this functionality by causing a victim’s browser to fetch a URL like `http://patsy-bank.com/redirect?to=http://patsy-bank.com/transfer...`, and from patsy-bank.com’s perspective, it will see a subsequent request for `http://patsy-bank.com/transfer...` that indeed has a `Referer` from patsy-bank.com.

(d) Recall that the `Referer` header provides the full URL. HTTP additionally offers an
**Origin** header which acts the same as the **Referer** but only includes the website domain, not the entire URL. Why might the **Origin** header be preferred?

**Solution:** Leaking the entire URL can be a violation of privacy against users. As an example, consider Alice transferred money by visiting http://patsy-bank.com/transfer?amt=1000&to=bob and subsequently went to a website under an attacker’s control - now the attacker has learned the exact amount of money Alice sent and to who. The **Origin** header would only leak that Alice was at the patsy-bank.com.

As a sidenote not directly related to the question, the **Origin** is a very useful way to solve the CSRF problem since it makes it much easier for multiple, trusted sites to make some action. For example, Patsy-Bank might trust http://www.trustedcreditcardcompany.com to directly transfer money from a user’s account. This is a use-case that the CSRF token-based solution doesn’t support cleanly.

(e) Almost all browsers support an additional cookie field **SameSite**. When **SameSite=strict**, the browser will only send the cookie if the requested domain and origin domain correspond to the cookie’s domain. Which CSRF attacks will this stop? Which ones won’t it stop? Give one big drawback of setting **SameSite=strict**.

**Solution:** It stops almost all CSRF attacks, except those involving open redirects from the website in question or if the website itself has an XSS vulnerability (discussed in the next problem).

However, setting **SameSite=strict** can greatly limit functionality since any external links that require a user to be logged in won’t work. For instance, consider a friend sends you a Facebook link via email, clicking on that link will require you to sign in again since your session cookie wasn’t sent with the request.
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:

```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:

```go
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.