Web Security III: CSRF Mitigation, SQL Injection

CS 1660: Introduction to Computer Systems Security
CSRF attacks

Browser performs unwanted action while user is authenticated
CSRF: via GET

bad-site.com:

<a href="http://bank.com/transfer.php&acct=1234?amt=1000.00?..."

• Bad practice: state change info encoded in GET request
• Can easily "replay" request
CSRF: via POST

bad-site.com:

<form action="https://bank.com/wiretransfer" method="POST"
    id="bank">
    <input type="hidden" name="recipient" value="Attacker">
    <input type="hidden" name="account" value="2567">
    <input type="hidden" name="amount" value="$1000.00">
    ...
</form>

document.getElementById("bank").submit();

Is user is logged in, this will work!
CSRF Trust Relationships

- Server trusts user (login)
- User trusts victim enough to visit attacker’s site/click link
- Attacker could be a hacked legitimate site

User

Legitimate Request

Malicious Request

Attacker

Server

Login

2/26/23

CSRF and SQL Injection
CSRF: How to defend?

How can we make sure a request comes from the intended origin?
One way: CSRF token

Server sends unguessable value to client, include as hidden variable in POST

```
<form action="/transfer.do" method="post">
<input type="hidden" name="csrf_token" value="aXg3423fjp. . .">
[...]
</form>
```

On POST, server compares against expected value, rejects if wrong or missing

What does this prove?
CSRF Token: Mechanics

Different web frameworks handle tokens differently

• Set token per-session or per-request?
• Can include token directly in generated HTML, or use JS to set via cookie

How to generate the tokens?

• "Synchronizer token": server picks random value, saves for checking
• "Encrypted token": server sends encrypt/MAC of some value that can be checked without saving extra state (e.g. user ID)
CSRF Token Types

Synchronizer Token
- Stateful
- Value randomly generated with large entropy
- Mapped to user's current session
- Server validates that token exists and is associated to user's session ID

Encrypted Token
- Stateless
- Token generated from user ID and timestamp
- Encrypted with server’s secret key
- Server validates token by verifying it and checking that it corresponds to current user and acceptable timestamp
- Ex. Encrypted Token = HMAC-SHA-1(‘secret key’ + user ID + timestamp)
Another way: checking headers

"Referer" [sic] header: URL from which request is sent

Request Headers

:authority: fonts.googleapis.com
:method: GET
:path: /css?family=Alegreya:ital,wght@0,400;0,700;1,400&family=Lost:ital,wght@0,300;0,400;0,500;0,1,500;1,600;1,700&display=swap
:scheme: https
:accept: text/css,*/*;q=0.1
:accept-encoding: gzip, deflate, br
:accept-language: en-US,en;q=0.9
:cache-control: no-cache
:pragma: no-cache
:referer: https://cs.brown.edu/
:sec-ch-ua: "Chromium";v="110", "Not A(Brand);v="24", "Google Chrome";v="110"
:sec-ch-ua-mobile: ?0
:sec-ch-ua-platform: "macOS"
:sec-fetch-dest: style
:sec-fetch-mode: no-cors
Another way: checking headers

• Could check Referer header (or a different header) on request, see if it matches expected origin
• Browser limits how Referer header can be changed

=> Useful if you trust browser; but ultimately can be controlled by client
Strict SameSite Cookie Attribute

Controls how a cookie is sent when making a cross-site request

```
Set-Cookie: sessionid=12345; Domain=b.com; SameSite=Strict
```

- **SameSite=None**: Always send cookie for any request to b.com
- **SameSite=Strict**: Only send cookie if request from same site (i.e., already on bob.com)
- **SameSite=Lax**: Only send if user is navigating to b.com (clicking a link), but not for in-page resource loads
  - As of 2020, default in most browsers not specified
Potential issues

• SameSite attribute set to Strict:
  – the browser will not include the cookie in any requests that originate from another site.

• A logged-in user follows a third-party link to a site:
  – they will appear not to be logged in, and will need to log in again before interacting with the site in the normal way

• Potential problems for usability and user tracking (e.g. Ads)

• Not all browsers have adopted default policy for websites that do not set SameSite
  – https://www.chromium.org/updates/same-site/
User Interaction

• Make a user reauthenticate, submit a one-time token, or do a CAPTCHA before performing any user-specific or privileged action on a website

• Scenario
  – Alice is logged into bob.com
  – Eve tricks Alice into visiting her page eve.com in another tab, which automatically redirects to send a malicious request to bob.com
  – Alice sees a login page for bob.com, but she thought she was visiting eve.com

• Potential issue: negatively impacts user experience
Example CSRF defenses: TryHackMe
Webapps + Databases
Most complex sites use a database

- Client-supplied data stored into database
- Access to database mediated by server
- Examples: Relational, Document oriented, ...
The Great CS1660(TM) Database

- Student data stored into database
- Access to database mediated by server
Standard Query Language (SQL)

- Relational database
  - Data organized into tables
  - Rows represent records and columns are associated with attributes

- SQL describes operations (queries) on a relational database

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Grade</th>
<th>Password</th>
<th>admin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernardo</td>
<td>345</td>
<td>-</td>
<td>H(password)</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>122</td>
<td>C</td>
<td>H(bob123)</td>
<td>0</td>
</tr>
<tr>
<td>Alice</td>
<td>543</td>
<td>A</td>
<td>H(a3dsr87)</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
One query type: SELECT

- Find records in table (FROM clause) that satisfy a certain condition (WHERE clause)
- Result returned as table (attributes given by SELECT)

SELECT attributes FROM table
WHERE condition; -- comments
SELECT: Data flow

Alice

Server

CS1660 Database

Insert your name to access your grade:

A

POST Alice's grade

SELECT name, grade from CS1660 WHERE name=Alice

200 OK: Alice, A

Student: Alice
Grade: A
Select: Data flow

Alice

Insert your name to access your grade:

Alice

Server

POST Alice's grade

CS1660 Database

SELECT name, grade from CS1660 WHERE name=Alice
### Example Query: Authentication

**SQL Query**

```sql
SELECT * FROM CS1660 WHERE Name='$username' AND Password = hash( $passwd );
```

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Grade</th>
<th>Password</th>
<th>admin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernardo</td>
<td>345</td>
<td>-</td>
<td>H(password)</td>
<td>1</td>
</tr>
<tr>
<td>Bob</td>
<td>122</td>
<td>C</td>
<td>H(bob123)</td>
<td>0</td>
</tr>
<tr>
<td>Alice</td>
<td>543</td>
<td>A</td>
<td>H(a3dsr87)</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example Query: Authentication

SELECT * FROM CS1660 WHERE
Name=$username AND Password = hash($passwd);
UPDATE Function

- Update records in table (UPDATE clause) that satisfy a certain condition (WHERE clause)
DELETE Function

• Delete records in table (DELETE clause) that satisfy a certain condition (WHERE clause)

DELETE FROM table
WHERE condition; -- comments
ALTER Function

- Alter the fields in table (ALTER clause) by adding a new column with a certain size (e.g. varchar(20))

```
ALTER TABLE table
ADD element varchar(20); -- comments
```
SQL Injection
Problem: How to handle user input?

Basic approach:

```php
$db->query("SELECT * from users where username=" . $user . 
" AND password = " . $hash ");
```
The problem

• User data could affect query string!

• What can we do??

• How to handle it??
SQL Injection

- Attacker bypasses protections on database
  - Causes execution of unauthorized queries by injecting SQL code into the database
SQL Injection to Bypass Authentication

$\textit{username} = 'A' OR 1 = 1 --'

$\textit{passwd} = \text{anything}

Resulting query:

\textbf{SELECT} * \textbf{FROM} CS1660 \textbf{WHERE} Name= 'A' OR 1 = 1 --' \textbf{AND} ...
SQL Injection for Data Corruption

- $username = A'; UPDATE CS1660 SET grade='A'
  WHERE name=Bob' --'
- $passwd = anything
- Resulting query execution

```
SELECT * FROM CS1660 WHERE Name = $username AND Password = hash($passwd);
```

```
SELECT * FROM CS1660 WHERE Name = 'A';
UPDATE CS1660 SET grade='A' WHERE Name='Bob' -- AND ...
```
SQL Injection for Privilege Escalation

SELECT * FROM CS1660 WHERE Name = '$username' AND Password = hash( $passwd ) ;

- $username = A'; UPDATE CS1660 SET admin=1 WHERE name=‘Bob' --'
- $passwd = anything
- Resulting query execution
  SELECT * FROM CS1660 WHERE Name = 'A';
  UPDATE CS1660 SET admin=1 WHERE name=‘Bob' -- AND ...
Source: http://xkcd.com/327/
What We Have Learned

• Cross-Site Request Forgery (CSRF) attack
• CSRF mitigation techniques
• Web applications with a server-side database
  – Architecture and data flow
  – Simple SQL queries
• SQL injection
  – Example attacks and mitigation techniques