Web Security IV:
SQL Injection, XSS,
Vulnerability Discovery & Disclosure

CS 1660: Introduction to Computer Systems Security
Code Injection

User input gets treated as part of the application code

=> user can do something they couldn’t otherwise
Code injection we have seen so far...

1) **SQL INJECTION**
   - User input affects DB query run by server

2) **Cross-Site Scripting**
   - User input runs JS in victim's browser
From last time

**SQL injection**: input becomes part of SQL query on server

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

**Cross-Site Scripting (XSS)**: input can run arbitrary Javascript in browser

```html
<ul>
  <li>hi, this is alice</li>
  <li><script>alert("xss");</script></li>
  ...
</ul>
```
How do we defend against this?

Once again, defense in depth...
First: limiting cookie sharing

More important attributes:

```
Set-Cookie: sessionid=12345; . . . HttpOnly=true
```

- **HttpOnly (true/false):** If true, cookie can’t be read by Javascript, eg. `document.cookie`

- **Fetch/XMLHTTPRequest** can still send them, even if JS code can’t read directly (“credentialed requests”)

**Tradeoff in how cookie can be used** => useful for cookies with credentials

More info: [Mozilla MDN]
Authentication Required

Enter your Brown credentials

Username
jcarberr

Password

Log In

You have asked to log in to:

CANVAS
Brown University – Canvas
What can we do about it?
Idea: clean up the text

How could we prevent input from acting like code?

This is called input sanitization => escape or filter certain characters to avoid them being parsed as code
**XSS: What to filter?**

Problem: really hard to build a good sanitization function => can’t hope to catch everything...

<script>alert("XSS");</script>

Can get devious...

<script>alert("XSS");</script>

<code>&lt;img src=# onerror="alert('XSS')"&gt;</code>

<code>&lt;img src=# onerror="alert(String.fromCharCode(88,83,83))"&gt;</code>

. . .

More info: Flag wiki, OWASP filter evasion cheat sheet

Note: *not all of these exact tricks may work in all modern browsers.*
Your experience may vary, see cheat sheet for more examples.
Sanitizing SQL

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

What to escape? Starting point:
  ' " \ <newline> <return> <null>

Quirks: Unicode, rich text, ...
Warning: building sanitizers is very tricky to get right. 

Never, ever write custom sanitizers on your own!

Instead

• Use library functions designed for this
• Reconsider your design to avoid needing a sanitizer in the first place
Input Sanitization: Examples

Examples
- PHP legacy escape function `mysql_escape_string` ignored similar character encodings in Unicode
- PHP later developed `mysql_real_escape_string`

Both of these functions are deprecated now...
How can we do better?
A better way for SQL: Prepared Statements

1. Backend parses this:

```
SELECT * from users WHERE user = ? AND password = ?
```

Output: query object

- Newer form of writing queries: variables with `?` filled in _after_ query text is parsed

- Generally safe from SQL injection, if used correctly

2. When user input is received, backend calls something like:

```python
query_obj.process(input)
```

This fills in the placeholders for the inputs and actually runs the query.
// Prepare query ahead of time
$stmt = $db->prepare(
    'SELECT * from users WHERE username = :user AND password = :pass');

For each input, execute query
$r = $stmt->execute([':user' => $user, ':pass' => $pass]);
Anomaly Detection

- Observe queries on legitimate inputs
- Determine properties of typical queries
  - Result size (e.g., list of values or probability distribution)
  - Structure (e.g., WHERE expression template)
- Reject inputs that yield atypical queries and outputs
Anomaly Detection (eg. for SQL)

- Typical queries
  - Result size: 0 or 1
  - Structure: variable = string
- On malicious input $A' OR 1 = 1$
  - Result size: table size
  - Structure: variable = string OR value = value

```sql
SELECT * FROM CS1660 WHERE Name=$username AND Password = hash( $passwd ) ;
```
XSS: Content-Security-Policy (CSP)

CSP header tells browser to load content only from certain origins

<!- Only allow content from this origin -->
<!- (also restricts inline scripts) -->
Content-Security-Policy: default-src 'self'
XSS: Content-Security-Policy (CSP)

CSP header tells browser to load content only from certain origins

<!- Only allow content from this origin -->
<!- (also restricts inline scripts) -->
Content-Security-Policy: default-src 'self'

<!- Allow certain media from different sources-->
Content-Security-Policy: default-src 'self'; img-src *;
media-src example.org example.net;
script-src userscripts.example.com

Opportunities for more precise control over what resources can be loaded
What happens when user inputs need rich formatting?
How To Create A Great Page For Your MySpace

2.3K views 11 years ago
In the Real World: MySpace Worm

• Users could post HTML on MySpace pages...
  • ...but MySpace blocks a lot of tags (except for <a>, <img>, and <div>)
    • No <script>, <body>, onClick attributes, <a href=javascript:/>,
      ...but some browsers allowed JavaScript within CSS tags:
      • <div style="background:url('javascript:eval(...')")">
  • ...but MySpace strips out the word “javascript”...
    • ...so use <div style="background:url('java\nscript:eval(...')")">
  • ...but MySpace strips out all escaped quotes...
    • ...so convert from decimal: String.fromCharCode(34) to get “
  • ...etc

Source: https://samy.pl/myspace/tech.html
In the Real World: MySpace Worm

- Everyone who visits an “infected” profile page becomes infected and adds samy as a friend
  - Within 5 hours, samy has 1,005,831 friends
- Moral of the story
  - Don’t homebrew your own filtering mechanisms
  - Use established libraries that you trust
  - Multiple valid representations make it difficult to account for every possible scenario

Source: https://samy.pl/myspace/tech.html
Rich text: What can we do instead?

- Does social media allow inline HTML anymore? Nope.
- An alternative: languages like markdown that are rendered to HTML

### Headings

To create a heading, add number signs (#) in front of a word or phrase. The number of number signs you use should correspond to the heading level. For example, to create a heading level three (<h3>), use three number signs (e.g., ### My Header).

<table>
<thead>
<tr>
<th>Markdown</th>
<th>HTML</th>
<th>Rendered Output</th>
</tr>
</thead>
<tbody>
<tr>
<td># Heading level 1</td>
<td>&lt;h1&gt;Heading level 1&lt;/h1&gt;</td>
<td>Heading level 1</td>
</tr>
<tr>
<td>## Heading level 2</td>
<td>&lt;h2&gt;Heading level 2&lt;/h2&gt;</td>
<td>Heading level 2</td>
</tr>
<tr>
<td>### Heading level 3</td>
<td>&lt;h3&gt;Heading level 3&lt;/h3&gt;</td>
<td>Heading level 3</td>
</tr>
</tbody>
</table>
Rich text: What can we do instead?

• Does social media allow inline HTML anymore? Nope.
• An alternative: languages like markdown that are rendered to HTML

---

Parse input and add features, rather than removing them!
One more injection example...
From last time: 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)
Another way: reflected XSS attack

**Class demo**: site with 404 ("not found") page:

**Goal**: get victim to click a link like this:

**Note**: URLs normally can’t contain characters like "<". When creating URLs with special characters, URLs use "URL encoding", which is a way to represent these characters.

Browsers normally convert characters to/from URL encoding transparently, but this might not happen in other tools (like Burp). If you’re doing an attack like this, be sure to make sure your payload is formatted correctly. For example, the raw form of the URL in this attack would be:
http://localhost:9080/04-xss-02/404.php?page=%3Cscript%3Ealert(%22yo%22);%3C/script%3E
More server-side code injection: what if we can run arbitrary code on the backend?

Class demo: a PHP page that uses eval() or require to load arbitrary PHP code

- Example 1: eval(x) interprets x as if it were code => almost all scripting languages have a function like this
- Example 2: in PHP, "require x" loads PHP code from a file or URL

=> Both of these are extremely dangerous => attacker can do anything the backend code on the server can do!

Examples:
- `echo "hello world!";` // Print something
- `shell_exec("ls -la");` // Run an arbitrary shell command :O

Running arbitrary shell commands means we can do anything we want on the server--it’s as if we had a terminal logged into the server as the same user that runs the webserver.

However, this is a really slow way to run commands--can we do better? We can already run arbitrary code, so the answer is yes, we can!

=> How? The answer is a reverse shell: a program we can run on the server that will connect back to us and start a shell (an interactive terminal where we can run commands), like /bin/bash

We don’t need to write the code for the shell--there are lots of examples for this online. Here’s how to think about setting it up (annotated based on the class example):

1. **Attacker listens for connections**
   => Attacker needs to have a place to receive new network connections. In class, we did this on the dev container by listening on a port.

2. **Send payload**: Attacker sends reverse shell code to server such that it will execute => Code needs to contain address and port it to use for connection (which must match listener in step (1)

3. **Server connects to listener and start a shell**: the reverse shell code causes the server to make a network connection back to (1) and starts a shell program (usually /bin/bash). This allows the attacker to send commands as if they were using a normal shell!

For more details, including how to use the simple webserver to host the reverse shell code, see the Flag setup guide!