SQL Injection and XSS

CS 166: Introduction to Computer Systems Security
Database Driven Websites
Database Driven Websites

- Client-supplied 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>First_Name</th>
<th>Last_Name</th>
<th>Student_ID</th>
<th>Course_Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hillary</td>
<td>Clinton</td>
<td>345</td>
<td>B</td>
</tr>
<tr>
<td>Donald</td>
<td>Trump</td>
<td>122</td>
<td>C</td>
</tr>
<tr>
<td>Bernie</td>
<td>Sanders</td>
<td>543</td>
<td>A</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
<td>.....</td>
<td></td>
</tr>
</tbody>
</table>
SELECT Query

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

```sql
SELECT name FROM students 
WHERE grade = 'A'; -- list A students
```
Data Flow

• Typical data flow in web applications
  – Client input sent as a variable (GET or POST)
  – Variable used to form an SQL query
  – Data flows from the client to the database, i.e., the client supplies data to the database

• Example:
  
  ```sql
  SELECT role FROM users
  WHERE user = '$username';
  ```
Login Authentication Query

• Standard query for login authentication

```sql
SELECT * FROM users
   WHERE user = '$username' AND pwh = hash('sha256', '$passwd')
```

– Access granted if query returns nonempty table

• Client input

– Server side code sets variables `$username` and `$passwd`
  from user input into a login HTML form

– The `users` table includes attributes `user` (user name) and `pwh`
  (password hash)
SQL Injection
SQL Injection

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

• Standard query to authenticate users
  
  ```sql
  SELECT * FROM users
  WHERE user = '$username' AND pwh = hash('sha256', '$passwd');
  ```

• Attacker enters
  
  User name: A' OR 1 = 1; -- Password: anything

• Resulting query execution
  
  ```sql
  SELECT * FROM users WHERE user = 'A' OR 1 = 1; --' AND pwh = ...
  ```
SQL Injection for Privilege Escalation

• Standard query to update username
  
  ```sql
  UPDATE users SET user = '$newusername'
  WHERE user = $current
  ```

• Suppose table users has 0/1 attribute admin to denote user has administrator privileges

• Attacker is a regular user who wants to become administrator
  
  New username: evil', admin = '1

• Resulting query execution
  
  ```sql
  UPDATE users SET user = 'evil', admin = '1' WHERE user = $current
  ```
SQL Injection for Data Corruption

• Standard query to authenticate users:

```
SELECT * FROM users
WHERE user = '$username' AND pwh = hash('sha256', '$passwd');
```

• Attacker enters

Username: A'; DROP TABLE users; -- Password: anything

• Resulting query execution

```
SELECT * FROM users WHERE user = 'A'; DROP TABLE users; -- 'AND pwd = ...
```
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/
Input Sanitization

• Escape potentially malicious characters

• Result of escaping quotes in input M’ ; DROP table user; --

    SELECT * FROM users WHERE user = 'M\' DROP table user; --\'

• More generally, characters to escape include

    ' " \ <newline> <return> <null>

• Sanitizing input is tricky

  – Alternate character encodings may bypass default escape functions
  – PHP legacy escape function mysql_escape_string ignored encoding
  – PHP later developed mysql_real_escape_string
Second-Order SQL Injection

• Sanitized input may be reused in other queries
• Regular user selects username `admin'--`
• Application
  – Escapes quote to prevent possible injection attack
  – Stores value `admin'--` into user attribute of database
• Later, application retrieves username with clause
  \[ \text{WHERE username = 'admin'--'} \]
• Could be used to change administrator password to one chosen by attacker
Prepared Statements

- Two-phase SQL command
  - Write SQL statement with ? placeholders
  - Subsequently provide values that replace placeholders
- Best practice for
  - writing new applications
  - modifying existing ones
- May be unsuitable for legacy code

- Generally safe from SQL injection
  - Separation of code and data
  - Values replacing placeholders always treated as data

- Potential vulnerability when
  - Prepared statement is itself built from user input
  - Prepared statement calls queries from library
Structural 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

- SELECT user, pwd FROM users WHERE user = 'username';
- 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
Cross-Site Scripting
Cross-Site Scripting (XSS)

• Problem: users can submit text that will be displayed on web pages
  – Facebook posts
  – Blog comments
  – Webmail

• Browsers interpret everything in HTML pages as HTML

• What could go wrong?
Cross Site Scripting (XSS)

- Attacker injects scripting code into pages generated by a web application
  - Script could be malicious code
  - JavaScript (AJAX!), VBScript, ActiveX, HTML, or Flash
- Threats:
  - Phishing, hijacking, changing of user settings, cookie theft/poisoning, false advertising, execution of code on the client, ...
XSS Example

- Website allows posting of comments in a guestbook
- Server puts comments into page returned
  
  Thanks for signing my guestbook!<br />
  Here's what everyone else had to say:<br />
  Joe: Hi! <br />
  John: Hello, how are you? <br />
  Jane: How does this guestbook work? <br />
- Attacker can post comment that includes malicious JavaScript
  Evilguy: <script>alert("XSS Injection!");</script> <br />

```html
<html>
<title>Sign My Guestbook!</title>
<body>
Sign my guestbook!
<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 XSS Attacks

Attack 1

```
<script>
</script>
```

Attack 2

```
<script>
img = new Image();
</script>
```
Another XSS Attack

• Mallory finds that Bob’s site is XSS vulnerable
• Mallory makes a tampered URL to use this vulnerability and sends to Alice an email pretending to be from Bob with the tampered URL
• Alice uses the tampered URL at the same time while she is logged on Bob’s site
• The malicious script is executed in Alice browser
• Unbeknown to Alice, the script steals Alice’s confidential information and sends it to Mallory’s site
Client-side XSS defenses

- **Proxy-based:**
  - Analyze HTTP traffic between browser and web server
  - Look for special HTML characters
  - Encode them before rendering the page browser (e.g., NoScript)
- **Application-level firewall:**
  - Analyze HTML pages for hyperlinks that might leak information
  - Stop bad requests using a set of connection rules
- **Auditing system:**
  - Monitor execution of JavaScript code and compare the operations against high-level policies to detect malicious behavior
What We Have Learned

• Web applications with a server-side database
  – Architecture and data flow
  – Simple SQL queries
• A class of attacks on web servers
  – SQL injection
  – Example attacks and mitigation techniques
• A class of attacks on web clients
  – XSS
  – Example attacks and mitigation techniques