Server-side Web Security: SQL Injection Attacks & XSS

CS 161: Computer Security
Prof. David Wagner
February 12, 2014
SQL Injection Scenario

• Suppose web server front end stores URL parameter “recipient” in variable $recipient and then builds up a string with the following SQL query:

```sql
$sql = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username='\$recipient' ";
```

• So for “?recipient=Bob” the SQL query is:

```
"SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username='Bob' ";
```
Parse Tree for SQL Example

```
SELECT AcctNum FROM Customer
WHERE Balance < 100 AND Username='Bob'
```
SQL Injection Scenario

• Suppose web server front end stores URL parameter “recipient” in variable $recipient and then builds up a string with the following SQL query:

```
$sql = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username='$recipient' ";
```

• How can $recipient cause trouble here?
  – How can we see anyone’s account?
    • Even if their balance is >= 100
SQL Injection Scenario, cont.

WHERE Balance < 100 AND
    Username='$recipient'

• Conceptual idea (doesn’t quite work): Set recipient to “foo' OR 1=1” ...

  WHERE Balance < 100 AND
      Username='foo' OR 1=1'

• Precedence makes this:

  WHERE (Balance < 100 AND
          Username='foo') OR 1=1

• Always true!
SELECT AcctNum FROM Customer
WHERE (Balance < 100 AND Username='foo') OR 1=1
Why “foo' OR 1=1” doesn’t quite work:
WHERE Balance < 100 AND Username='foo' OR 1=1'

**Syntax error**: quotes aren’t balanced

SQL server will reject command as ill-formed
• Why “foo' OR 1=1” doesn’t quite work:
  WHERE Balance < 100 AND
  Username='foo' OR 1=1'

• Sneaky fix: use “foo' OR 1=1️⃣”
  Begins SQL comment …
SQL Injection Scenario, cont.

• Why “foo' OR 1=1” doesn’t quite work:
  WHERE Balance < 100 AND
  Username='foo' OR 1=1'

• Sneaky fix: use “foo' OR 1=1 --”

• SQL server sees:
  WHERE Balance < 100 AND
  Username='foo' OR 1=1--'

When parsing SQL query, SQL server ignores all of this since it’s a comment …

So now it finds the quotes balanced; no syntax error; successful injection!
SQL Injection Scenario, con’t

WHERE Balance < 100 AND 
Username='$recipient'

• How about $recipient = foo'; DROP TABLE Customer -- ?

• Now there are two separate SQL commands, thanks to ‘;’ command-separator.

• Can change database however you wish
SQL Injection: Summary

- **Target:** web server that uses a back-end database

- **Attacker goal:** inject or modify database commands to either read or alter web-site information

- **Attacker tools:** ability to send requests to web server (e.g., via an ordinary browser)

- **Key trick:** web server allows characters in attacker’s input to be interpreted as SQL control elements rather than simply as data
Welcome to the Amazing World Of Squigler ...
# Some Squigler Database Tables

<table>
<thead>
<tr>
<th>username</th>
<th>body</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ethan</td>
<td><em>My first squig!</em></td>
<td>2013-02-27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21:51:52</td>
</tr>
<tr>
<td>cathy</td>
<td><em>@ethan: borrr-ing!</em></td>
<td>2013-02-27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21:52:06</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
def post_squig(user, squig):
    if not user or not squig: return
    conn = sqlite3.connect(DBFN)
    c = conn.cursor()
    c.executescript("INSERT INTO squigs VALUES ('%s', '%s', datetime('now'));" % (user, squig))
    conn.commit()
    c.close()

Server code for posting a “squig”

Syntax error
# Squigler Database Tables

## Accounts

<table>
<thead>
<tr>
<th>username</th>
<th>password</th>
<th>public</th>
</tr>
</thead>
<tbody>
<tr>
<td>dilbert</td>
<td>funny</td>
<td>‘t’</td>
</tr>
<tr>
<td>alice</td>
<td>kindacool</td>
<td>‘f’</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
INSERT INTO squigs VALUES
  (dilbert, ' ' || (select (username || ' _' || password) from accounts where username='bob') || ' ', date);
INSERT INTO squigs VALUES
  (dilbert, '|| (select (username || '_' || password) from accounts where username='bob') || '', date);

Empty string literals
INSERT INTO squigs VALUES
(dilbert, ' ' || (SELECT (username || '.' || password) FROM accounts WHERE username='bob') || ' ',
date);

A blank separator, just for tidiness
INSERT INTO squigs VALUES
(dilbert, (select (username || '_' || password) from accounts where username='bob') || ' ',
date);

Concatenation operator.

Concatenation of string $S$ with empty string is just $S$

INSERT INTO squigs VALUES
(dilbert, (select (username || '_' || password) from accounts where username='bob'),
date);

Value of the squig will be Bob’s username and password!
Defenses

Language support for constructing queries
Specify query structure independent of user input:
Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user)
{
    String query = "SELECT AcctNum FROM Customer WHERE
        Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

“Prepared Statement”
Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user) {
    String query = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

Untrusted user input
Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user) {
    String query = "SELECT AcctNum FROM Customer WHERE 
                     Balance < 100 AND Username ≠ ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

Input is confined to a single SQL atom
Note: *prepared* statement only allows '?'s for leaves, not internal nodes. So *structure* of tree is *fixed.*
Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user) {
    String query = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

Binds the value of `arg_user` to '?' atom
Defenses

Language support for constructing queries
Specify query structure independent of user input:

```java
ResultSet getProfile(Connection conn, String arg_user) {
    String query = "SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, arg_user);
    return p.executeQuery();
}
```

No matter what input user provides, Prepared Statement ensures it will be treated as a single SQL datum
SELECT / FROM / WHERE

AcctNum AND Customer AND

< Balance AND = 100 AND = Username AND foo' OR 1=1 --
This will never be true (assuming no bizarre Usernames!), so no database records will be returned.
XSS

- Cross-site scripting (XSS): tricking browsers into giving undue access to attacker’s Javascript
  - *Stored* XSS: attacker leaves Javascript lying around on benign web service for victim to stumble across
  - *Reflected* XSS: attacker gets user to click on specially-crafted URL with script in it, web service reflects it back
Dynamic Web Pages

• Rather than static HTML, web pages can be expressed as a program, say written in Javascript:

```html
<title>Javascript demo page</title>

<font size=30>
Hello, <b>
<script>
var a = 1;
var b = 2;
document.write("world: ", a+b, "</b>");
</script>
```
Javascript

• Powerful web page *programming language*
• Scripts are embedded in web pages returned by web server
• Scripts are executed by browser. Can:
  – Alter page contents
  – Track events (mouse clicks, motion, keystrokes)
  – Read/set cookies
  – Issue web requests, read replies
• *(Note: despite name, has nothing to do with Java!)*
Confining the Power of Javascript Scripts

• Given all that power, browsers need to make sure JS scripts don’t abuse it
• For example, don’t want a script sent from hackerz.com web server to read cookies belonging to bank.com ...
• … or alter layout of a bank.com web page
• … or read keystrokes typed by user while focus is on a bank.com page!
Same Origin Policy

- Browsers provide isolation for JS scripts via the **Same Origin Policy (SOP)**

- Simple version:
  - Browser associates web page elements (layout, cookies, events) with a given origin ≈ web server that provided the page/cookies in the first place
    - Identity of web server is in terms of its hostname, e.g., bank.com

- SOP = *only scripts received from a web page’s origin have access to page’s elements*
XSS: Subverting the Same Origin Policy

• It’d be **Bad** if an attacker from evil.com can fool your browser into executing script of their choice …
  – … with your browser believing the script’s origin to be some other site, like bank.com

• One nasty/general approach for doing so is **trick** the server of interest (e.g., bank.com) to actually send the attacker’s script to your browser!
  – Then no matter how carefully your browser checks, it’ll view script as from the same origin (because it is!) …
  – … and give it all that **powerful/nasty access**

• Such attacks are termed **Cross-Site Scripting** (XSS)
## Syndicate

- **R**: Domains already xss'ed.
- **S**: Famous and Government web sites.
- **F**: Status: Fixed/Unfixed.
- **PR**: Pagerank by Alexa®.

You can subscribe to our [mailing list](mailto:subscribe@xssed.com) to receive alerts by mail.

### Table

<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Domain</th>
<th>R</th>
<th>S</th>
<th>F</th>
<th>PR</th>
<th>Category</th>
<th>Mirror</th>
</tr>
</thead>
<tbody>
<tr>
<td>21/02/11</td>
<td>LostBrilliance</td>
<td>audience.cnn.com</td>
<td>R</td>
<td></td>
<td></td>
<td>53</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>21/02/11</td>
<td>db</td>
<td>freedns.afraid.org</td>
<td></td>
<td></td>
<td></td>
<td>8834</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>19/02/11</td>
<td>h3rcul3s</td>
<td>cwg2010.indianexpress.com</td>
<td></td>
<td></td>
<td></td>
<td>2942</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>18/02/11</td>
<td>Yeyah</td>
<td>app.email.skype.com</td>
<td></td>
<td></td>
<td></td>
<td>189</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>17/02/11</td>
<td>warvector</td>
<td><a href="http://www.level3.com">www.level3.com</a></td>
<td></td>
<td></td>
<td></td>
<td>53575</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>17/02/11</td>
<td>SeeMe</td>
<td>api.screename.aol.com</td>
<td></td>
<td></td>
<td></td>
<td>51</td>
<td>XSS</td>
<td>mirror</td>
</tr>
</tbody>
</table>
Two Types of XSS (Cross-Site Scripting)

- There are two main types of XSS attacks
- In a *stored* (or "persistent") XSS attack, the attacker leaves their script lying around on bank.com server
  - ... and the server later unwittingly sends it to your browser
  - Your browser is none the wiser, and executes it within the same origin as the bank.com server