Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

Presenter: Ruichao Qiu

Adam Kiezun, Philip J. Guo, Karthick Jayaraman, Michael D. Ernst

PHP Source Code

SQLI attacks

1st-order XSS attacks

2nd-order XSS attacks

Slides adapted from https://homes.cs.washington.edu/~mernst/pubs/create-attacks-icse2009-slides.ppt
Overview

Problem: Finding security vulnerabilities (SQLI and XSS) in Web applications

Approach:
1. Automatically generate inputs
2. Dynamically track taint
3. Mutate inputs to produce attack input

Results:
60 unique new vulnerabilities in 5 PHP applications, first to create 2nd-order XSS, no false positives
PHP Web applications

if ($_GET['mode'] == "add")
    addMessageForTopic();
else if ($_GET['mode'] == "display")
    displayAllMessagesForTopic();
else
    die("Error: invalid mode");

function addMessageForTopic() {
    $my_msg = $_GET['msg'];
    $my_topicID = $_GET['topicID'];
    $my_poster = $_GET['poster'];

    $sqlstmt = "INSERT INTO messages VALUES('$my_msg', '$my_topicID') ";

    $result = mysql_query($sqlstmt);
    echo "Thanks for posting, $my_poster";
}
Example: Message board (display mode)

```php
if ($_GET['mode'] == "add")
    addMessageForTopic();
else if ($_GET['mode'] == "display")
    displayAllMessagesForTopic();
else
    die("Error: invalid mode");
```

```php
function displayAllMessagesForTopic() {
    $my_topicID = $_GET['topicID'];
    $sqlstmt = "SELECT msg FROM messages WHERE topicID='$my_topicID';";
    $result = mysql_query($sqlstmt);
    while($row = mysql_fetch_assoc($result)) {
        echo "Message: " . $row['msg'];
    }
}
```

Message: hi there
Terminology Definition

• SQL Injection
  – User input for database statement
  – Structure of the SQL query changed
  – Get unauthorized access to data
SQL injection attack

If ($_GET['mode'] == "add")
    addMessageForTopic();
else if ($_GET['mode'] == "display")
    displayAllMessagesForTopic();
else
die("Error: invalid mode");

function displayAllMessagesForTopic()
{
    $my_topicID = $_GET['topicID'];
    $sqlstmt = "SELECT msg FROM messages WHERE topicID='" . $my_topicID . "';"
    $result = mysql_query($sqlstmt);
    while($row = mysql_fetch_assoc($result)) {
        echo "Message: " . $row['msg'];
    }
}
Terminology Definition

- First-order XSS
  - Pass tainted data into function
  - Display HTML with attacker’s code
  - Steal browser cookies
First-order XSS attack

```php
if ($_GET['mode'] == "add")
    addMessageForTopic();

function addMessageForTopic() {
    $my_poster = $_GET['poster'];
    [...]
    echo "Thanks for posting, $my_poster";
}
```

Thanks for posting, A
• Second-order XSS
  – Store attacker’s input in database
  – Execute attacker’s code in HTML page
  – Affect multiple victim users
Second-order XSS attack

Example MALICIOUS input:
“uh oh<script>alert(‘XSS’)</script>”

 attackers input:

$_GET[]:
mode = “add”
msg = MALICIOUS
topicID = 42
poster = “Villain”

addMessageForTopic()
Second-order XSS attack

Example MALICIOUS input: “uh oh<script>alert(‘XSS’)</script>”

Message: uh oh
Architecture

- **Input Generator**
- **Taint Propagator**
- **Attack Generator/Checker**

Inputs flow into the Taint Propagator, which propagates taint sets. The Taint Propagator then provides inputs to the Attack Generator/Checker.

The output from the Attack Generator/Checker is fed into a **Concrete + Symbolic Database**, which is part of the Ardilla system.

Malicious inputs are also generated and fed into the system.
Input generation

**Goal:** Create a set of concrete inputs (_$GET[] & _$POST[])  
Use Apollo generator (Artzi et al. ’08)
if ($_GET['mode'] == "add")
    addMessageForTopic();
else if ($_GET['mode'] == "display")
    displayAllMessagesForTopic();
else
    die("Error: invalid mode");
Taint propagation

**Goal:** Determine which input variables affect each potentially dangerous value

**Technique:** Execute and track data-flow from input variables to *sensitive sinks*

**Sensitive sinks:** `mysql_query()`, `echo()`, `print()`
Example: SQL injection attack

1. **Generate** inputs until program reaches an SQL statement

   SELECT msg FROM messages WHERE topicID='$my_topicID'

2. **Collect taint sets** for values in sensitive sinks:

   ```
   function displayAllMessagesForTopic() {
     $my_topicID = $_GET['topicID'];
     $sqlstmt = "SELECT msg FROM messages WHERE topicID='$my_topicID'";
     $result = mysql_query($sqlstmt); /* {'topicID'} */
   }
   ```

   Sensitive sink

   Taint set
Attack generation and checking

**Goal:** Generate attacks for each sensitive sink

**Technique:** Mutate inputs into candidate attacks
- Replace tainted input variables with shady strings developed by security professionals:
  - e.g., “1’ or ‘1’ = ‘1”, “<script>code</script>”
Attack generation and checking

Given a program, an input i, and taint sets

for each var that reaches any sensitive sink:

res = exec(program, i)

for shady in shady_strings:

mutated_input = i.replace(var, shady)

mutated_res = exec(program, mutated_input)

if mutated_res DIFFERS FROM res:

report mutated_input as attack
Attack generation: mutating inputs

res = exec(program, i)
for shady in shady_strings:
    mutated_input = i.replace(var, shady)
    mutated_res = exec(program, mutated_input)
    if mutated_res DIFFERS FROM res:
        report mutated_input as attack
Attack checking: diffing outputs

res = exec(program, i)
for shady in shady_strings:
    mutated_input = i.replace(var, shady)
    mutated_res = exec(program, mutated_input)
    if mutated_res DIFFERS FROM res:
        report mutated_input as attack

What is a significant difference?
- For SQLI: compare SQL parse tree structure
- For XSS: compare HTML for additional script-inducing elements (<script></script>)
Concrete + Symbolic Database

- Database: shared state enables data exchange
- A duplicate of concrete database
- Additional columns for symbolic data (taint set)

<table>
<thead>
<tr>
<th>msg</th>
<th>topicid</th>
<th>msg_s</th>
<th>topicid_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test message</td>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>Hello</td>
<td>2</td>
<td>{msg}</td>
<td>{topicid}</td>
</tr>
</tbody>
</table>
Concrete + Symbolic Database

• Rewrite SQL statement

```
SELECT msg FROM messages WHERE topicid = '2'
```

```
SELECT msg, msg_s FROM messages WHERE topicid = '2'
```
Experimental results

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>LOC</th>
<th>SourceForge Downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchoolMate</td>
<td>School administration</td>
<td>8,181</td>
<td>6,765</td>
</tr>
<tr>
<td>WebChess</td>
<td>Online chess</td>
<td>4,722</td>
<td>38,457</td>
</tr>
<tr>
<td>FaqForge</td>
<td>Document creator</td>
<td>1,712</td>
<td>15,355</td>
</tr>
<tr>
<td>EVE activity tracker</td>
<td>Game player tracker</td>
<td>915</td>
<td>1,143</td>
</tr>
<tr>
<td>geccBBlite</td>
<td>Bulletin board</td>
<td>326</td>
<td>366</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vulnerability Kind</th>
<th>Sensitive sinks</th>
<th>Reached sensitive sinks</th>
<th>Unique attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLI</td>
<td>366</td>
<td>91</td>
<td>23</td>
</tr>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;-order XSS</td>
<td>274</td>
<td>97</td>
<td>29</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;-order XSS</td>
<td>274</td>
<td>66</td>
<td>8</td>
</tr>
</tbody>
</table>

Total: 60
Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

• Contributions
  – Automatically create SQLI and XSS attacks
  – First technique for 2\textsuperscript{nd}-order XSS

• Technique
  – Dynamically track taint through both program and database
  – Input mutation and output comparison

• Implementation and evaluation
  – Found 60 new vulnerabilities, no false positives