Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

Presenter: Ruichao Qiu

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

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

URL
- GET
- POST

Database

Data

SQL

PHP application

HTML
- ...<HTML>
- ...
- <script>
- ...

Web browser

Example: Message board (add mode)

```php
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";
}
```

$_GET[]:  
mode = “add”  
msg = “hi there”  
topicID = 42  
poster = “Bob”

Thanks for posting, Bob
Example: Message board (display mode)

```php
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'];
    }
}
```

$_GET[]: mode = "display"  topicID = 42

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

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

```sql
SELECT msg FROM messages WHERE topicID='1' OR '1'='1'
```
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

```html

```

The page at file:///localhost says:

XSS
```

OK
Terminology Definition

• 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>”

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

addMessageForTopic()
Second-order XSS attack

**Example MALICIOUS input:**

```
“uh oh<script>alert(‘XSS’) </script>”
```

** Victim’s input: **

$_GET[]:
mode = “display”
topicID = 42
poster = “Villain”

displayAllMessagesForTopic()

** PHP application **

** Attacker’s input: **

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

addMessageForTopic()

echo()

**Message:** uh oh
Architecture

[Diagram showing the architecture of Ardilla]

- **Input Generator**
  - Inputs
  - Symbolic Database

- **Taint Propagator**
  - Taint sets

- **Attack Generator/Checker**

- **PHP Source Code**

- **Concrete + Symbolic Database**

- **Malicious inputs**
Goal: Create a set of concrete inputs (_$GET[] & _$POST[])

Use Apollo generator (Artzi et al. ’08)
Input generation:

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

inputs:

- **$_GET[]:**
  - mode = "1"
  - msg = "1"
  - topicID = 1
  - poster = "1"

- **$_GET[]:**
  - mode = "add"
  - msg = "1"
  - topicID = 1
  - poster = "1"

- **$_GET[]:**
  - mode = "display"
  - msg = "1"
  - topicID = 1
  - poster = "1"
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:
   ```
   {'topicID'}
   ```

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

Sensitive sink

Taint set
**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:

\[
\text{res} = \text{exec}(\text{program}, i)
\]

for shady in shady_strings:

\[
\text{mutated_input} = i.\text{replace}(\text{var}, \text{shady})
\]

\[
\text{mutated_res} = \text{exec}(\text{program}, \text{mutated_input})
\]

if mutated_res **DIFFERS FROM** \( \text{res} \):

report \( \text{mutated_input} \) as attack

---

**PHP Source Code**

**Attack Generator/Checker**

**Malicious inputs**

**Attack generation**

**Attack checking**
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

```python
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>∅</td>
<td>∅</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

### Vulnerability kinds

<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(^{st})-order XSS</td>
<td>274</td>
<td>97</td>
<td>29</td>
</tr>
<tr>
<td>2(^{nd})-order XSS</td>
<td>274</td>
<td>66</td>
<td>8</td>
</tr>
<tr>
<td>Total:</td>
<td></td>
<td></td>
<td>60</td>
</tr>
</tbody>
</table>

### Software details

<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>
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