Slides for a 25-minute conference presentation on this paper:


Created by Philip J. Guo
pg@cs.stanford.edu
Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

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

International Conference on Software Engineering
May 20, 2009
Overview

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

Approach:
1. Automatically generate inputs
2. Dynamically track taint (through program and DB)
3. Mutate inputs to produce exploits

Results:
60 unique new vulnerabilities in 5 PHP applications, no false positives
PHP Web applications

Example: Message board (add mode)

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

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

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";
}

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

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
SQL injection attack

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

$_GET[]:
mode = "display"
topicID = 1' OR '1'='1

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'];
    }
}

SELECT msg FROM messages WHERE topicID='1' OR '1'='1"
First-order XSS attack

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

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

Example **MALICIOUS** input:
```
"uh oh<script>alert('XSS')</script>"
```

Thanks for posting, uh oh
Second-order XSS attack

Example **MALICIOUS** input:
“uh oh&lt;script&gt;alert(‘XSS’)&lt;/script&gt;”

Attacker’s input

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

addMessageForTopic()

PHP application

Database
Second-order XSS attack

Attacker’s input

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

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

PHP application

addMessageForTopic()
displayAllMessagesForTopic()
echo()

Message: uh oh

Victim’s input

$_GET[]:
mode = “display”
topicID = 42
Input generation

**Goal:** Create a set of concrete inputs ($_GET[] & $_POST[])

We use Apollo generator (Artzi et al. ’08), based on **concolic execution**
Input generation: concolic execution

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

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

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

- **$_GET[]:**
  - mode = "add"
  - msg = "1"
  - topicID = 1
  - poster = "1"
if ($_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
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()`
Taint propagation: data-flow

Each value has a **taint set**, which contains input variables whose values flow into it.
Taint propagation: data-flow

Each value has a **taint set**, which contains input **variables** whose values flow into it

```php
function displayAllMessagesForTopic() {
    $my_topicID = $_GET['topicID'];
    $sqlstmt = "SELECT msg FROM messages WHERE topicID='$my_topicID';"
    $result = mysql_query($sqlstmt); /* {‘topicID’} */
```
Taint propagation: data-flow

Each value has a **taint set**, which contains input **variables** whose values flow into it.

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

**Taint propagation**

- **Assignments**: `$my_poster = $_GET['"poster"']`
- **String concatenation**: `$full_n = $first_n . $last_n`
- **PHP built-in functions**: `$z = foo($x, $y)`
- **Database operations** (for 2nd-order XSS)
Attack generation

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

**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>"

**Alternative:** String constraint solver (Kiezun et al. '09)
Attack generation

Given a program and an input i
**Attack generation**

*Given a program and an input i*

for each var that reaches any sensitive sink:

```
res = exec(program, i)
```
Given a program and an input \( i \)

for each var that reaches any sensitive sink:

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

for shady in shady\_strings:

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

\[
\text{mutated\_res} = \text{exec}(\text{program}, \text{mutated\_input})
\]
Attack generation

*Given a program and an input* $i$

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
```
Atack 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 generation: differ 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>)

Avoids false positives from input sanitizing and filtering
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' }`

3. **Generate** attack candidate by picking a shady string

4. **Check** by mutating input and comparing SQL parse trees:
   - *innocuous*: `SELECT msg FROM messages WHERE topicID='1'
   - *mutated*: `SELECT msg FROM messages WHERE topicID='1' OR '1'='1'`

5. **Report** an attack since SQL parse tree structure differs
## 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>Kind</th>
<th>Sensitive sinks</th>
<th>Reached sensitive sinks</th>
<th>Tainted sensitive sinks</th>
<th>Unique attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLI</td>
<td>366</td>
<td>91</td>
<td>76</td>
<td>23</td>
</tr>
<tr>
<td>1\textsuperscript{st}-order XSS</td>
<td>274</td>
<td>97</td>
<td>78</td>
<td>29</td>
</tr>
<tr>
<td>2\textsuperscript{nd}-order XSS</td>
<td>274</td>
<td>66</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

Total: 60
Comparison with previous work

Defensive coding:
+ : can completely solve problem if done properly
- : must re-write existing code

Static analysis:
+ : can potentially prove absence of errors
- : false positives, does not produce concrete attacks

Dynamic monitoring:
+ : can prevent all attacks
- : runtime overhead, false positives affect app. behavior

Random fuzzing:
+ : easy to use, produces concrete attacks
- : creates mostly invalid inputs
Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

- Contributions
  - Automatically create SQLI and XSS attacks
  - First known 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