Detecting Vulnerabilities in Web Code with concolic execution

Suman Jana

*slides are adapted from Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst
Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

Ardilla by Kiezun et al. [ICSE 2009]
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 exploits

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

$_GET[]

$_POST[]

Example: Message board (add mode)

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");
```

```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: " . $row['msg'];
    }
}
```
if ($_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 = 1' OR '1'=1

SELECT msg FROM messages WHERE topicID=1 OR 1=1
if ($_GET['mode'] == "add")
    addMessageForTopic();

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

Thanks for posting, uh oh
Stored XSS attack

$_GET[]:
    mode = “add”
    msg = “uh oh<script>alert( ‘XSS’ )</script>”
    topicID = 42
    poster = “Attacker”

Attacker’s input
Stored XSS attack

$_GET[]:
   mode = “add”
   msg = “uh
    oh<script>alert(‘XSS’)</script>”
   topicID = 42
   poster = “Attacker”

Attacker’s input

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

Victim’s input

addMessageForTopic()

PHP application

displayAllMessagesForTopic()

echo()

Message: uh oh

Database

The page at file://localhost says: XSS
Ardilla's Architecture:

1. **PHP Source Code**
   - Inputs to **Input Generator**
   - Taint sets to **Taint Propagator**

2. **Taint Propagator**
   - Propagates taint sets to **Attack Generator/Checker**
   - Transfers information to **Database with taint-tracking**

3. **Attack Generator/Checker**
   - Generates attacks based on taint sets

4. **Database with taint-tracking**
   - Stores propagated taint information

5. **Malicious inputs**
   - Used for testing and validation
Goal: Create a set of concrete inputs ($_GET[] & $_POST[]) based on concolic execution
if ($_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"

$_GET[]:
  mode = "display"
  msg = "1"
  topicID = 1
  poster = "1"
Example: SQL injection attack

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

   ```php
   SELECT msg FROM messages WHERE topicID='$_GET[topicID]'
   ```

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

   ```php
   $_GET[]:
   mode = "display"
   msg = "1"
   topicID = 1
   poster = "1"
   ```
**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**
- Assignments: $my_poster = $_GET[“poster”]
- String concatenation: $full_n = $first_n . $last_n
- PHP built-in functions: $z = foo($x, $y)
- Database operations (for stored XSS)
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’ } */
   }
   ```
**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:** Use a string constraint solver
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

$_GET[]:
    mode = “display”
    topicID = 1

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

    ```
    $_GET[]:  
    mode = "display"  
    topicID = 1
    ```

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

Avoids false positives from input sanitizing and filtering
Example: SQL injection attack

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

### Vulnerability Kind

<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

---

### Name

<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>

---

Main limitation: input generator
Comparison with other approaches

**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
Summary

• Automatically create SQLI and XSS attacks

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

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